

# MySQL Magazine

Winter 2007/2008 Issue 3

Happy  
Holidays

Welcome to MySQL Magazine. This is the magazine for the community of MySQL database administrators and developers who use MySQL on a daily basis to provide some of the best infrastructure available on the Internet.

What will you find here? News, articles on both administration and development, feedback from the users. Simply anything relating to MySQL!!

## MySQL in the News

The MySQL Users Conference (<http://en.oreilly.com/mysql2008/public/content/home>) is coming! The conference is April 14th - 17th in Santa Clara, California. Early registration (which saves you 200 USD) closes February the 26th. Come join over 1600 other MySQL users and hear people like Jay Pipes, Brian Aker and Baron Schwartz. There will be tutorials and sessions covering many, many topics.

## Book Worm

Recently I was asked to review the **Expert MySQL** book written by Dr. Charles Bell. I am glad that I was offered the opportunity as it is a quality book. It is definitely not for the novice though! Some of the topics covered include how queries travel "through" the MySQL server, building a storage engine and how to work with the source code.

The book is well written. It is obvious that Dr. Bell knows his subject in great detail. If you are wanting to move to the next level and work on in the depths of the code of the MySQL Server this is a great place to jump off.

Apress (<http://www.apress.com>), who publishes the book, kindly agreed to allow the magazine to reprint a small portion of the book. Turn to page 10 to take a look.

# Contents

Winter 2007/2008

6	InnoDB Performance Optimizations
10	Excerpt from Expert MySQL
13	Index Optimizatoin in MySQL - Part II

## Regular Features

1	News
1	Book Worm
3	Coding Corner - How to Call a Stored Procedure with an Out Parameter
16	log-bin

# How to Call a Stored Procedure with an Out Parameter

by Peter Brawley  
<http://www.artfulsoftware.com>

Stored procedures in MySQL 5 and 6 accept IN, OUT and INOUT parameters. The parameters work as documented in the mysql client. Do they work when called via MySQL language APIs?

MySQL Connector/NET supports OUT parameters, and the manual describes how to use them. That's so also for Connector/J. But the C API, Connector/ODBC 3.51, Connector/ODBC 5.1, and Connector/PHP have no syntax for fetching the value of an OUT or INOUT parameter. Mark Matthews threw some light on the issue in a 3 March 2006 bug note:

"Since there's no support in the protocol for an RPC call or binding directionality of parameters using server-side prepared statements for stored procedures, clients run into the issue that there is no standard or straightforward way to deal with OUTPUT or INOUT parameters, especially when a user wants to use a literal as the IN value of an INOUT parameter.

"We can work around it in the clients, but it means yet more parsing code (different in each connector), and the parsing code will begin to get complex as we'll have to support a subset of SQL."

Apparently only the .NET and JDBC connectors solve the problem, via client-side workarounds under the hood. If you read further down Mark's note, you see a hint that MySQL 6 might introduce proper server-side support for OUT parameters. Until then, what are we to do when a C, ODBC or PHP application needs an OUT parameter value?

One idea is to pass the name of a user variable in the OUT parameter slot and fetch its value with a subsequent query. In late 2005, Hasani Blackwell posted such a workaround in the MySQL C/C++ forum.

That leaves PHP and ODBC. Connector/NET beats Connector/ODBC in many respects, so it's fair to say that needing OUT parameter values is just another reason to move a project from ODBC to .NET.

Then what about PHP? We start with the mysql PHP API. Given the need to call PROCEDURE myproc(IN i int, OUT j int), we enable multiple queries per function call by calling `mysql_connect()` with `flags = 65536 (CLIENT_MULTI_STATEMENTS)`, and address the OUT parameter from the PHP script as ...

```
$result = mysql_query( "CALL myproc(1,@myvar);SELECT @myvar" );
```

But when we run this, we find that `mysql_fetch_row()` ignores the result of `SELECT @myvar`. Dead end.

Then can we put `SELECT @myvar` in a second query call?

```
$res1 = mysql_query( "CALL myproc(1,@myvar)" );  
$res2 = mysql_query( "SELECT @myvar" );
```

The PHP parser accepts the second call, but the call does not execute because after the mysql API executes a stored procedure call, it refuses to execute any more queries! You have to close the current connection and open a new one--where, of course, @myvar no longer exists. Another dead end.

We have a clear answer to whether OUT parameter values can be fetched via mysql PHP API. They cannot.

What about the mysqli API? Like the C API, it is also missing a syntax for fetching OUT parameter values directly, but at least it continues to execute queries after a stored procedure call. And it has a function, multi\_query(), which accepts multiple queries in a semicolon-separated string argument. So we can write ...

```
$mysqli = new mysqli( "HOST", "USR", "PWD", "DBNAME" );
$res = $mysqli->multi_query( "CALL PROCNAME(param,@outparam); SELECT @outparam" );
```

There is a syntax for retrieving results from all those queries: iterate calls to store\_result(), fetch\_row(), next\_result() and more\_results() until next\_result() returns FALSE.

Beware that MySQL makes a bit of a liar out of the PHP manual page for store\_result(): it says the function "transfers the result set from the last query ...". In fact next\_result() starts with the first query called by multi\_query() and proceeds to the last:

```
if( $res ) {
    $results = 0;
    do {
        if ( $result = $mysqli->store_result() ) {
            printf( "<b>Result #%u</b>:<br/>", ++$results );
            while( $row = $result->fetch_row() ) {
                foreach( $row as $cell ) echo $cell, "&nbsp;";
            }
            $result->close();
            if( $mysqli->more_results() ) echo "<br/>";
        }
    } while( $mysqli->next_result() );
}
$mysqli->close();
```

As a test of this workaround, assume table names(id int, lastname char(20), firstname(char(20))), a bit of data in the table, and this stored procedure:

```
CREATE procedure t(IN i, OUT j)
BEGIN
    SELECT * FROM names WHERE id=i;
    SET j = 1;
END;
```

If we execute ...

```
$mysqli->multi_query( "CALL t(1,@myvar); SELECT @myvar" );
```

then the above code produces this result:

```
Result #1:
2 Smith Fran
Result #2:
1
```

This is not elegant. Direct access to OUT parameter values will be much better if it ever comes. But at least this works, and if you use stored procedures with PHP, it illustrates why you pretty much have to use the mysqli API.

### **About the author**

Peter Brawley is president of Artful Software Development and co-author of **Get It Done With MySQL 5**.

# InnoDB Performance Optimizations

by Peter Zaitsev

When I am interviewing people for job openings I like to ask them a basic question - if you have a server with sixteen gigabytes of RAM which will be dedicated to MySQL with a large InnoDB database using a typical Web workload what settings would you would adjust? Most people fail to come up with anything reasonable. So I decided to publish the answers I would like to hear. In addition, I will extend it with the basics of hardware, operating system and application optimization.

I call this InnoDB performance optimization basics so these are general guidelines which will work well for a wide range of applications. Keep in mind that the optimal settings depend on the workload.

## Hardware

If you have a large InnoDB database than the amount of RAM is very important. Sixteen to thirty-two gigabytes is very cost-efficient these days. From the standpoint of CPUs, two dual-core CPUs work very well. With just two quad-core CPUs scalability issues can be observed on many workloads. You will find that this depends on the application a great deal. The third area of hardware is the I/O subsystem. It is best to use directly attached storage with plenty of spindles and a RAID array with battery backed up (BBU) cache is optimal. Typically you can get six to eight hard drives in the standard server case and this is often enough. Also take notice of the new 2.5" SAS hard drives. They are small but often faster than larger drives. RAID10 works well for data storage with a more write intensive load. For situations where it is more read-intensive and you still want some redundancy RAID 5 can work well. Be careful with RAID 5 though when it comes to random writes.

## Operating System

When considering the operating system you will run MySQL on it is important to run a 64-bit operating system. We still see people running 32-bit Linux on 64-bit capable servers with plenty of memory. Do not do this. If you are using Linux for your operating system you should use Linux Volume Manager (LVM) for your database directory. This allows for more efficient backup. The EXT3 file system provides acceptable performance in most cases. If you are running into particular roadblocks with your filesystem try XFS. You can use noatime and nodiratime options if you're using `innodb_file_per_table` and a lot of tables though the benefit of these settings is minor. Also make sure that your operating system will not swap MySQL out of memory.

## InnoDB Settings

The most important ones are:

**`innodb_buffer_pool_size`** 70-80% of memory is a safe bet. I set it to 12G on 16GB box.

**`innodb_log_file_size`** - This depends on your recovery speed needs but 256M seems to be a good balance between reasonable recovery time and good performance

**`innodb_log_buffer_size=4M`** 4M is good for most cases unless you're piping large blobs to InnoDB in this case increase it a bit.

**`innodb_flush_logs_at_trx_commit=2`** If you're not concern about ACID and can loose transactions for the last second or two in case of a full operating system crash than set this value. It can have a dramatic effect on throughput, especially on a lot of short write transactions.

**innodb\_thread\_concurrency=8** Even with current InnoDB scalability fixes having limited concurrency helps. The actual number may be higher or lower depending on your application and default which is eight is a decent start.

**innodb\_flush\_method=O\_DIRECT** Avoid double buffering and reduce swap pressure, in most cases this setting improves performance. Be careful if you do not have battery backed up RAID cache as your write I/O may suffer.

**innodb\_file\_per\_table** - If you do not have too many tables use this option. This will keep you from having an uncontrolled InnoDB main tablespace growth which you can not reclaim. This option was added in MySQL 4.1 and is now stable enough to use.

Also determine if your application can run in READ-COMMITTED isolation mode. If it is possible than set it to be default by adding `transaction-isolation=READ-COMMITTED` to your `my.cnf` file. This option has some performance benefits, especially with locking in 5.0 and even more with MySQL 5.1 and row level replication.

### **Application Tuning for InnoDB**

Especially when coming from a MyISAM background there will be some changes you would like to do with your application. First make sure you're using transactions when doing updates, both for sake of consistency and to get better performance. Next if your application has any writes be prepared to handle deadlocks which may happen. Third, you would like to review your table structure and see how you can get advantage of InnoDB properties - clustering by primary key, having primary keys in all indexes (so keep primary key short), fast lookups by primary keys (try to use it in joins), and large unpacked indexes (try to be easy on indexes).

With these basic InnoDB performance tunings you will be better than the majority of InnoDB users who use MySQL with the default settings, running it on hardware without battery backed up cache and no operating system changes. In addition, they make no changes to the application which was written for MyISAM tables in mind.

### **InnoDB Buffer Pool**

The InnoDB buffer pool setting is by far the most important option for InnoDB performance and it must be set correctly. I have seen a number of clients who have extreme problems by leaving the buffer pool setting at the default value of eight megabytes. If you have a dedicated MySQL server and you are only using InnoDB tables you will want to use all the memory you do not utilize for other needs to the InnoDB buffer pool.

Using 70-80% of total memory is a good baseline but you might want to adjust this somewhat. Of course you should not stick just to 50GB if you have 64G of memory - values of 56-60G would likely make more sense, and depending other settings 12G or 14G may well be good choice for 16GB server, though I would take care at values close to 14G as there is not much room left for other things.

This assumes your database is large so you need a large buffer pool. If not you should set the buffer pool a bit larger than your database size will be enough. You also should account for growth of course. You need buffer pool a bit (say 10%) larger than your data (total size of InnoDB table spaces) because it does not only contain data pages - it also contain adaptive hash indexes, insert buffer, locks which also take some time. Though it is not as critical - for most workloads if you will have your InnoDB buffer pool 10% less than your database size you would not loose much anyway.

You also may choose to set buffer pool as if your database size is already larger than amount of memory you have - so you do not forget to readjust it later. This is also valid approach as if it is Dedicated MySQL Server you may not have a good use for that memory anyway.

Another thing you should take into account is that Innodb allocates more memory in structures related to buffer pool than you specify for it - I just checked on our set of boxes which use twelve gigabyte buffer pool (with very small additional pool and log buffer) and total amount of memory allocated by Innodb is typically a bit over thirteen gigabyte.

After you have decided on a database size you need to check if there are any restrictions on Innodb Buffer Size you can use. Typically you would see restriction applying only on 32-bit systems but these can be still spotted in the wild, especially running on Windows. The restriction would normally apply to the total amount of memory the process can allocate so make sure to leave space for other MySQL needs while factoring this in.

The next step would be to decide how much memory you are going to need for other purposes. These needs are operating system needs - your system processes, page tables, socket buffers etc. They all need memory. I would set this to 256 megabytes for small sized boxes up to 5% of the total memory on the big boxes. It can be even less than that depending on the application. Besides operating system needs you also have MySQL needs - these include the MySQL buffers: query cache, key\_buffer, mysql threads, temporary tables, and per thread sort buffer which can be allocated. There are also other settings like innodb additional memory pool (which can grow larger than the memory you allocated for it - especially in case you have large amount of tables).

I could tell you some numbers, for example sum up all your global buffers plus add 1MB for each connection you're planning to have but in reality the number can vary significantly depending on the load. Idle connections for example will consume significantly less memory than connections doing work with huge temporary tables and running complex queries. It is usually much better to simply check it. Start MySQL with a ten gigabyte Innodb buffer pool for example and see how large RSS and VSZ get in "ps" output on Unix-based systems. If it gets to twelve gigabytes than you need two gigabytes for other stuff and you can increase it a bit to be on the safe size and scale appropriately.

The third memory consumer would be the operating system cache. You want to bypass cache for your Innodb tables but there are other things you need the operating system cache for - MyISAM tables (mysql database, temporary tables etc) will need it. Also the binary logs or relay logs. It is best to cache Innodb transactional logs - otherwise the operating system will need to do reads to serve writes to these log files as I/O to the log files is not aligned to the page boundary. Finally, you likely have some system scripts/processes running on the system which will also need some cache. The number can be a lot different depending on system workload but generally I'd see values from 200MB to 1GB as good estimates for this number.

### **Eliminate Double Buffering**

This is again very important for buffer pool size choice. You do not want the operating system to cache what Innodb is caching already. Innodb cache is more efficient compared to the operating system cache because there is no copying, due to adaptive hash indexes, ability to buffer writes and number of other factors so you just want to make your operating system to give a way to Innodb. Note it is not enough to block the operating system from swapping - as I already mentioned the operating system cache will be needed for other things and if you will not make Innodb to bypass operating system buffering Innodb table space I/O will wipe out cache because it typically makes most of the I/O on the system. On Windows you do not need to do anything. On Linux, FreeBSD and Solaris you need to set `innodb_flush_method=O_DIRECT`. On other operating systems you may be able to select it on

operating system level but make sure to do it. There is a small niche case when it hurts - when you do not have a RAID controller with a Battery Backup Unit (BBU) and your workload is very write intensive but there are always exceptions.

## **Make Your Operating System Happy**

The other challenge you may have is in making your operating system happy and avoiding swapping out MySQL process or other important processes to make room for file caching. The operating system may find it unfair that there is a MySQL process which consumes 95% of memory and the cache is just a couple of percent. Some people try to solve the problem by disabling the swap file but this can hurt another way - the operating system may kill the MySQL process when the operating system is running out of memory (or it thinks it is running out of memory). This may happen in the case of an unexpected connection spike. Also, not all kernels work well with the swap file disabled. For some people having no swap enabled works. Even so, they usually play things on the safe side, having enough “free” memory allocated as cache and buffers. Kevin Burton wrote a good post about his experiments.

Depending on the operating system you may want to do different virtual memory adjustments. You may want to make MySQL use Large Pages for allocating the Innodb buffer pool and a few other buffers. This might offer some performance benefits. Tuning your virtual memory to be less eager to swap by **echo 0 > /proc/sys/vm/swappiness** is another helpful change though it does not always save you from swapping. Some specific kernel versions may have other settings to play with. Finally you can try locking MySQL in memory by using memlock. Just be careful because if you have a memory usage spike there is a possibility that the mysqld daemon could be killed by the operating system instead of temporarily swapping a few things out.

There are two things to keep in mind about operating system swapping. First, looking at “swap used” is not really helpful because you do not know what is stored in the swap space. There are portions of both the MySQL process and other processes which you do not need during normal operation so it is acceptable to have them in the swap space. Make sure, however, that swapping is not happening continually. This can be checked with the vmstat command. Take a look at the “si/so” columns. They should be zero or close to it on Linux. Several swaps per minute would not hurt but if you’re doing 100+ pages per second of swap I/O you are in trouble.

Second, many people think - who cares if some of the buffer pool is swapped out, I would have one I/O to fetch page if I would have small buffer pool and now I have one I/O to fetch page data swapped out. This is very wrong thinking. First operating systems would have to swap even clean page from Buffer Pool while Innodb can simply discard that pages in case of memory pressure. But what is more important Innodb algorithms are finely tuned with consideration what is in memory and what is on disk, for example when Innodb tries to avoid holding latches while doing I/O while there could be locks set while accessing pages in the buffer pool. If page turns out to be in swap rather than memory you will have another threads piling up waiting on the same lock till I/O completion, while they may well have all data they need to proceed in the innodb buffer pool.

Note: I only described Innodb Buffer Pool selection for dedicated Innodb system. If you have a fair amount of MyISAM, Archive, PBXT, Falcon or other storage engines then you will get into complex balancing game besides considering all these factors.

## **About the author**

Peter Zaitsev is the co-founder of Percona consulting (<http://www.percona.com>) and works in all areas MySQL optimization. His blog is at <http://mysqlperformanceblog.com> and is a great source of performance tuning information.

# Excerpt from Expert MySQL

Following is an excerpt from the excellent book **Expert MySQL** by Dr. Charles Bell (Apress, 978-1-59059-741-5). It is from the section on how queries are optimized in a database server. The section is located on pages 36 through the top part of page 39.

## Query Optimizer

Some mistakenly believe that the query optimizer performs all of the steps outlined in the query execution phases. As you will see, query optimization is just one of the steps that the query takes on the way to be executed. The following paragraphs describe the query optimizer in detail and illustrate the role of the optimizer in the course of the query execution.

Query optimization is the part of the query compilation process that translates a data manipulation statement in a high-level, nonprocedural language, such as SQL, into a more detailed, procedural sequence of operators, called a query plan. Query optimizers usually select a plan by estimating the cost of many alternative plans and then choosing the least expensive among them (the one that executes fastest).

Database systems that use a plan-based approach to query optimization assume that many plans can be used to produce any given query. Although this is true, not all plans are equivalent in the number of resources (or cost) needed to execute the query, nor are all plans executed in the same amount of time. The goal then is to discover the plan that has the least cost and/or runs in the least amount of time. The distinction of either resource usage or cost usage is a trade-off often encountered when designing systems for embedded integration or running on a small platform (with low resource availability) versus the need for higher throughput (or time).

Figure 2-3 depicts a plan-based query processing strategy where the query follows the path of the arrows. The SQL command is passed to the query parser, where it is parsed and validated and then translated into an internal representation, usually based on a relational algebra expression or a query tree as described earlier. The query is then passed to the query optimizer, which examines all of the algebraic expressions that are equivalent, generating a different plan for each combination. The optimizer then chooses the plan with the least cost and passes the query to the code generator, which translates the query into an executable form, either as directly executable or as interpretative code. The query processor then executes the query and returns a single row in the result set at a time.

This is a common implementation scheme and is typical of most database systems. However, the machines that the database systems run on have improved over time. It is no longer the case that the query plans have diverse execution costs. In fact, most query plans have been shown to execute with approximately the same cost. This realization has led some database system implementers to adopt a query optimizer that focuses on optimizing the query using some well-known good rules (called heuristics) or practices for query optimization. Some database systems use hybrids of optimization techniques that are based on one form while maintaining aspects of other techniques during execution

The four primary means of performing query optimizations are

- Cost-based optimization
- Heuristic optimization
- Semantic optimization
- Parametric optimization

Though no optimization technique can guarantee the best execution plan, the goal of all these methods is to generate an efficient execution for the query that guarantees correct results.

A cost-based optimizer generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost based on the metrics (or statistics) gathered about the relations and operations needed to execute the query. For a complex query, many equivalent plans are possible. The goal of cost-based optimization is to arrange the query execution and table access utilizing indexes and statistics gathered from past queries. Systems such as Microsoft SQL Server and Oracle use cost-based optimizers.

Heuristic optimizers use rules concerning how to shape the query into the most optimal form prior to choosing alternative implementations. The application of heuristics, or rules, can eliminate queries that are likely to be inefficient. Using heuristics as a basis to form the query plan ensures that the query plan is most likely (but not always) optimized prior to evaluation. The goal of heuristic optimization is to apply rules that ensure "good" practices for query execution. Systems that use heuristic optimizers included Ingres and various academic variants. These systems typically use heuristic optimization as a means of avoiding the really bad plans rather than as a primary means of optimization.

The goal of semantic optimization is to form query execution plans that use the semantics, or topography, of the database and the relationships and indexes within to form queries that ensure the best practice available for executing a query in the given database. Though not yet implemented in commercial database systems as the primary optimization technique, semantic optimization is currently the focus of considerable research. Semantic optimization operates on the premise that the optimizer has a basic understanding of the actual database schema. When a query is submitted, the optimizer uses its knowledge of system constraints to simplify or to ignore a particular query if it is guaranteed to return an empty result set. This technique holds great promise for providing even more improvements to query processing efficiency in future RDBSs.

Parametric query optimization combines the application of heuristic methods with cost-based optimization. The resulting query optimizer provides a means of producing a smaller set of effective query plans from which cost can be estimated, and thus the lowest-cost plan of the set can be executed.

An example of a database system that uses a hybrid optimizer is MySQL. The query optimizer in MySQL is designed around a select-project-join strategy, which combines a cost-based and heuristic optimizer that uses known optimal mechanisms, thus resulting in fewer alternatives from which cost-based optimization can choose the minimal execution path. This strategy ensures an overall "good" execution plan, but does not guarantee to generate the best plan. This strategy has proven to work well for a vast variety of queries running in different environments. The internal representation of MySQL has been shown to perform well enough to rival the execution speeds of the largest of the production database systems.

An example of a database system that uses a cost-based optimizer is Microsoft's SQL Server. The query optimizer in SQL Server is designed around a classic cost-based optimizer that translates the query statements into a procedure that can execute efficiently and return the desired results. The optimizer uses information, or statistics, collected from values recorded into past queries and the characteristics of the data in the database to create alternative procedures that represent the same query. The statistics are applied to each procedure to predict which one can be executed more efficiently. Once the most efficient procedure is identified, execution begins and results are returned to the client.

Optimization of queries can be complicated by using unbound parameters, such as a user predicated. For example, an unbound parameter is created when a query within a stored procedure accepts a parameter from a user when the stored procedure is executed. In this case, query optimization may not be possible, or it may not generate the lowest cost unless some knowledge of the predicate is obtained prior to execution. If very few records satisfy the predicate, even a basic index is far superior to the file scan. The opposite is true if many records qualify. If the selectivity is not known when optimization is performed because the predicate is unbound, the choice among the alternative plans should be delayed until execution.

The problem of selectivity can be overcome by building optimizers that can adopt the predicate as an open variable and perform query plan planning by generating all possible query plans that are likely to occur based on historical query execution and by utilizing the statistics from the cost-based optimizer. The statistics include the frequency distribution for the predicate's attribute.

# Index Optimization in MySQL - Part II

by Keith Murphy

This article is a continuation of last issues article on the types of indexes available for MySQL. We talked about "normal" indexes, unique and primary indexes, covering indexes and duplicate indexes. If you need to refer to it the issue is available for download at <http://www.mysqlzine.net>.

This article will center around showing how indexes can directly affect the performance of your application. I built a simple database storing hypothetical data about some students grades. To make it modestly interesting I stored the data for 50,000 students. There is a total of three tests stored in the databases. The grades were randomly generated.

First some information about the database and tables:

```
mysql> show tables;
```

```
+-----+
| Tables_in_mysql_query_optimization |
+-----+
| grades                               |
| students                             |
| tests                                |
+-----+
```

```
mysql> describe students;
```

```
+-----+-----+-----+-----+-----+-----+
| Field      | Type   | Null  | Key  | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| student_id | int(11) | NO    | PRI  | NULL    | auto_increment |
| name       | char(40) | YES   |      | NULL    |              |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> describe grades;
```

```
+-----+-----+-----+-----+-----+-----+
| Field      | Type           | Null  | Key  | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| test_id    | int(11)        | YES   |      | NULL    |              |
| student_id | int(11)        | YES   |      | NULL    |              |
| score      | decimal(5,2)   | YES   |      | NULL    |              |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql> describe tests;
```

```
+-----+-----+-----+-----+-----+-----+
| Field  | Type   | Null  | Key  | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| test_id | int(11) | NO    | PRI  | NULL    | auto_increment |
| total   | int(11) | YES   |      | NULL    |              |
| name    | char(40) | YES   |      | NULL    |              |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql>
```

As you can see both students and tests have a primary key on student\_id and test\_id. The grades table does not have an index. Doing simple queries returns results quickly:

```
mysql> select sql_no_cache avg(score) from grades where test_id=1;
+-----+
| avg(score) |
+-----+
| 50.010551 |
+-----+
1 row in set (0.03 sec)
```

```
mysql>
```

Note: adding sql\_no\_cache means that the query will not be pulled from the query cache. This will give you consistent results with multiple runs of the same query.

However, once you start doing joins things get more complicated. I tried to run the next query and finally ended up killing it because it wasn't finishing:

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

After canceling the query (after 20 minutes of running) I ran EXPLAIN:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | s | index | NULL | PRIMARY | 4 | NULL | 50000 | Using index |
| 1 | SIMPLE | g | ALL | NULL | NULL | NULL | NULL | 120000 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 ses)
```

Now I add an index to grades table on the student\_id:

```
mysql> alter table grades add index student_id (student_id);
Query OK, 120000 rows affected (0.57 sec)
Records: 120000 Duplicates: 0 Warnings: 0
```

```
mysql>
```

Now I rerun the query:

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

```
+-----+
| count(*) |
+-----+
| 10000 |
+-----+
1 row in set (2.19 sec)
```

```
mysql>
```

It took just over two seconds. That is quite an improvement over killing the query after 20 minutes. Running EXPLAIN tells why:

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

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	s	index	NULL	PRIMARY	4	NULL	50000	Using index
1	SIMPLE	g	ref	student_id	student_id	5	s.student_id	2	Using where

```
2 rows in set (0.00 sec)
```

You can see that the number of rows examined dropped from 120,000 in the grades table (for each of the 50,000 students) to two rows (for each of the 50,000 students). That is quite a difference and the reason for the dramatic drop in execution time.

I hope that these two articles have helped you to understand how indexes work with your databases and how they can speed your query execution time. Just remember, indexes are great but don't index more than necessary. Every index cost overhead when your table is updated and you don't want to have an index just because you have a column. Remember that EXPLAIN is your friend.

### About the author

Keith Murphy is the DBA at iContact. He gets to play all day with fun toys devoping a better environment for the data of iContacts users. He tries to write on a reasonable schedule all the latest happenings at his blog at <http://blog.paragon-cs.com>.

The views and opinions expressed in this article are the author's own and may not reflect the views and opinions of iContact, Inc.

# log-bin

By the time you read this, there is a strong possibility that MySQL AB has released an “official” new general release version of MySQL Server. The latest release of the 5.1 series (5.1.22) was designated a RC (release candidate). Unless something comes up in final testing, the 5.1 branch will soon become the official production version of the server.

While 5.1.x does not bring as many new features to the DBA's plate as the 5.0.x servers did, it is still a momentous occasion. It is good to see our favorite database (and the company behind it) continuing to prosper. As we look down the road to future releases it is exciting to see the changes taking place. The MySQL database server will continue to be a market leader in the years to come.

As the new year is now here, take the time to reflect on 2007. Look for opportunities in 2008 to contribute to the Open Source community and to the MySQL community in particular. Join a MySQL users group if you have one in your area. Start one if you don't. Promote MySQL whenever you have a chance. Make it a great new year!!

This issue marks the completion of the first year of MySQL Magazine. I didn't really know what to expect when I got started on this little project but I have been pleasantly surprised with what has happened so far. Soon I am going to make some changes that I hope improve the magazine in the future. Stick around. It will be worth it!

Feel free to write me at [bmurphy@paragon-cs.com](mailto:bmurphy@paragon-cs.com).

Thanks,

Keith