

MySQL Magazine

Fall 2007 Issue 2



MySQL -
powering
the future

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 most recent news was the MySQL camp held in Brooklyn, New York, at Polytechnic University on August 22-23. It was a great little “un-conference” and a chance to get together with friends, learn some new things, have some good food and network.

There are already plans in the works for the next camp. It is planned for early 2008 in the Washington DC area. The website for the camp is <http://www.mysqlcamp.org>.

Book Worm

Not to long ago I picked up **MySQL Clustering** book by Alex Davies and Harrison Fisk. It is the “authoritative” book on clustering from MySQL. It is the only book currently available that is exclusively about MySQL clustering.

The good points of the book include it is fairly well written with well laid-out chapters exploring how you set up a cluster and administer it. In addition, it has a number of perl scripts used for various tasks. These scripts alone justify the cost of the book.

Now, the negative aspects. It is very thin at 202 pages. It also is dated. MySQL cluster is changing rapidly and this book covers version 5.0 while the latest changes in 5.1 are not covered in any details. Maybe there just isn't that much in-depth knowledge about MySQL clustering, but I am looking forward to the first “bible” on clustering as this aspect of MySQL continues to mature.

Contents

Fall 2007

3	MySQL Maintenance script -a multi-threaded tool for database maintenance
10	MySQL Table Checksum How-To
14	Optimizing MySQL with Rails - Part Two
18	Index Optimizatoin in MySQL

Regular Features

1	News
1	Book Worm
7	Coding Corner - Top N per Group
21	log-bin

MySQL Maintenance script

a multi-threaded tool for database maintenance

By Dan Buettner
drbuettner@gmail.com

MySQL_maintenance.pl is a perl script that uses multiple threads to check, optionally repair, and optionally optimize the tables in your MySQL databases - and do it in record time compared to a serial solution.

I worked with a content management and publishing system at a newspaper for several years that revolved around massive Sybase databases, to the point of storing huge Encapsulated PostScript graphics files in BLOBs. The system worked quite well, but as the core business of the newspaper happened in those databases, as a DBA I was understandably always cautious to make sure the databases were healthy. The CMS vendor supplied a nifty maintenance script for Sybase that would perform database consistency check (DBCC) routines on designated tables, and ran multiple threads to take advantage of our multi-processor Sun servers (and try to meet the demands of an increasingly around-the-clock news operation). We managed to squeeze quite a bit of maintenance into our little window every night, keeping the servers and disk arrays humming.

I started using MySQL for personal and work projects, and something I found lacking as I developed more solutions on MySQL was a tool like the data consistency checking and table optimization I was doing with Sybase. There just wasn't anything out there. Sure, it was easy to write a shell script to list tables and check then optimize each one, but I wanted something configurable, that could be intelligent about the check results, and more importantly, that would run as quickly as possible to minimize service interruptions.

I wrote my own simple, serial perl maintenance tool for MySQL in 2002 for my employer, and later wrote a new one from scratch as a multi-threaded solution. I've used it to maintain the health of MySQL databases at the newspaper as well as at the market research company I'm with now.

Why multi-threaded?

Speed. Modern servers can do multiple things at once, and a solution where you check one table after another in series is, frankly, wasting time: in any kind of production database, minimizing the impact of maintenance is probably a significant concern. If you have fast disks and multiple processing cores, it's quite likely that you could cut your maintenance time significantly by running a multi-threaded maintenance tool.

A note on perl threads: threading in perl is still technically labeled as not for production use. Honestly, I've been using threading in mission-critical perl scripts for three years, and have never had a problem with those scripts I would attribute to the use of threads.

The operation of the script

You run the script specifying, among other things, N worker threads. It creates N+1 connections to the database, one for each worker thread as well as one for a control thread. The control thread creates the workers and also creates a queue of work for them to do. The control thread lists databases and tables, and queues table check operations for each one. The worker threads begin popping work items off the queue: table checks to start with. When a table check is complete, if a problem was

found and repairs were specified, a repair operation for that table is put into the queue. If no problem is found, or a repair is successful, and table optimization was specified, an optimize operation for that table is put into the queue.

When a worker thread finished its current task, it gets the next one; when it finds the work queue empty, it exits. Once all worker threads have exited, the control thread prints results, sends the optional email summary, and exits.

The script's output, to STDOUT and in the optional email, lists the time and results of each operation, as well as providing some basic statistics about the tables it checked (number of records, size of data, size of indices).

How you might use it

For small databases, time required is generally not a huge issue, as MySQL can check and optimize small tables really quickly. When you're managing a truly massive set of databases, it's sometimes not possible to check the entire database at once no matter how fast your servers are or how many threads you run. An approach I have used in the past is to identify my largest tables and schedule my maintenance scripts such that some those are excluded most days of the week, in a rotating scheme; this lets me check other tables every day while still getting checks in on the larger tables a couple of times each week.

How many threads to run at once will vary from one situation to the next, and will depend on factors such as the size of your tables, the speed of your disks, and the number and speed of your CPUs. A good starting point for most small database servers, those with 2-4 processing cores, would be to run 3 or 4 worker threads and experiment to find your optimum number by increasing the number of worker threads by 1 until you observe the total runtime starting to increase, then reduce number of threads back down.

Options for the script

Running it with no arguments, or with --help, will get you the help screen:

```
#####  
MySQL_maintenance.pl  
By Dan Buettner - drbuettner@gmail.com
```

Multi-threaded table maintenance tool for MySQL database installations.

Options are as follows:

```
--host=<host>      MySQL host address (name or IP)  
--user=<username>  MySQL user to log in as - needs proper db privileges.  
--password=<password>  MySQL password  
--check-type=<type>  Check type (fast, quick, medium, extended). Default: quick  
--table-types=<type>  Comma-separated table types to check (default InnoDB,MyISAM)  
--optimize        Optimize tables (default: no)  
--repair          Repair table if error found during check (default: no)  
--threads=n       Number of worker threads (default 3). Total threads = n+1  
--exclude-dbs=<list>  Comma-separated list of database names/patterns to not check  
--exclude-tables=<list>  Comma-separated list of table names/patterns to not check  
--include-dbs=<list>  Comma-separated list of database names/patterns to check  
--include-tables=<list>  Comma-separated list of table names/patterns to check  
--smtp-host=<host>  Hostname or IP address of SMTP server to send mail through  
--email-from=<email>  Email address email should appear to be "from"  
--email-password=<pass>  Password to use in conjunction with --email-from for SMTP AUTH  
--email-to=<email>  Comma-separated addresses for email to go to  
--subject=xyz      Subject line of message, default "MySQL maintenance on <host>"
```

Example:

```
./MySQL_maintenance.pl --host=mysql.domain.com --user=user --password=pass \  
--check-type=extended --repair --optimize --threads=5 --exclude-dbs=test,test1  
#####
```

Hopefully, that is comprehensive and helpful enough in terms of the options available. At a minimum, you need to supply a hostname, a username, and a password for it to get past the help screen and run with default check level and number of threads.

Not specifying an SMTP host will mean no email summary is sent.

Database privileges required

The MySQL user this script connects as does not have to be root, but it does need to have the following (long) list of privileges in order to do its job: RELOAD, SHOW DATABASES, LOCK TABLES, SELECT, INSERT. Honestly, I usually use a user I have granted ALL to, but that is not strictly necessary.

Perl modules required

This script does require the installation of several modules. These have usually been fairly easy for me with CPAN on Mac OS X and Solaris, yum on Fedora Linux, and ActiveState's ppm perl package manager on Windows.

Here is the list of modules it requires: DBI (and DBD::mysql to go with it); Time::HiRes; Net::SMTP; threads; threads::shared; Thread::Queue::Any; Getopt::Long

MySQL versions supported

I've used this script or an earlier version of it on MySQL servers from 4.0.x through 5.0.x. Since the script really only directs MySQL to use its own CHECK/REPAIR/OPTIMIZE commands, it should in theory be compatible with earlier and later versions. If you do run into a compatibility problem, please let me know the specifics and I'll see what I can do about it.

A note on installation and use

It is not necessary to run the perl script on the same machine you run MySQL on; the perl script is relatively lightweight as it drives MySQL to do the heavy lifting. Running it on a client machine with network access to the MySQL server should work fine, if you make sure you have granted remote access to the MySQL user account you're using with this script.

For those of you less familiar with perl, I have included the traditional "shebang" line at the top of the script (something like '#!/usr/bin/perl'). For Windows users, you can replace that with just '#!perl'; for those of you with perl in other locations, adjust it as needed, or remove it and call it from cron like so: /path/to/perl /path/to/MySQL_maintenance.pl ...

Where to get it

Download the perl script here: http://dbuettner.dyndns.org/blog/?page_id=88

Please send me feedback!

I have relied on this script to keep my databases running well, and have added features as I found the need, but I recognize that it is not (yet) going to be perfect for every situation. If you find it doesn't work well for you for some reason, I'd like to hear why - and how you think I might be able to improve it so that it would. I hope this will be a valuable tool in people's DBA toolboxes. I can be reached at dr-buettner@gmail.com

About the author

Dan Buettner works for Outsell, Inc., a San Francisco area market research company tracking the information industry, though he lives in Des Moines, Iowa. In a past life he worked at The Des Moines Register, a midsize newspaper, as an IT manager, newspaper production specialist, software developer, and Sybase and MySQL DBA. He has been using MySQL since version 3.x in 1999.

Top N per Group

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

A table has multiple rows per key value, and you need to retrieve the first or earliest two rows per key.

If the groups are fairly small, this can be done efficiently with a self-join and counts. For example the following table (based on a tip by Rudy Limebac at <http://searchoracle.techtarget.com/>) has three small data groups:

```
DROP TABLE IF EXISTS test;
CREATE TABLE test (
  id INT,
  entrydate DATE
);
```

```
INSERT INTO test VALUES
( 1, '2007-5-01'),
( 1, '2007-5-02'),
( 1, '2007-5-03'),
( 1, '2007-5-04'),
( 1, '2007-5-05'),
( 1, '2007-5-06'),
( 2, '2007-6-01'),
( 2, '2007-6-02'),
( 2, '2007-6-03'),
( 2, '2007-6-04'),
( 3, '2007-7-01'),
( 3, '2007-7-02'),
( 3, '2007-7-03');
```

The first two rows per ID are the rows which, for a given ID, have two or fewer rows with earlier dates. If we use an inequality join with the COUNT(*) function to find the earlier rows per ID ...

```
SELECT t1.id, t1.entrydate, count(*) AS earlier
FROM test AS t1
JOIN test AS t2 ON t1.id=t2.id AND t1.entrydate >= t2.entrydate
GROUP BY t1.id, t1.entrydate
```

```
+-----+-----+-----+
| id | entrydate | earlier |
+-----+-----+-----+
| 1 | 2007-05-01 | 1 |
| 1 | 2007-05-02 | 2 |
| 1 | 2007-05-03 | 3 |
| 1 | 2007-05-04 | 4 |
| 1 | 2007-05-05 | 5 |
| 1 | 2007-05-06 | 6 |
| 2 | 2007-06-01 | 1 |
| 2 | 2007-06-02 | 2 |
| 2 | 2007-06-03 | 3 |
| 2 | 2007-06-04 | 4 |
| 3 | 2007-07-01 | 1 |
| 3 | 2007-07-02 | 2 |
| 3 | 2007-07-03 | 3 |
+-----+-----+-----+
```

... then we get our result immediately by removing rows where the 'earlier' count exceeds 2:

```
SELECT t1.id, t1.entrydate, count(*) AS earlier
FROM test AS t1
JOIN test AS t2 ON t1.id=t2.id AND t1.entrydate >= t2.entrydate
GROUP BY t1.id, t1.entrydate
HAVING earlier <= 2;
```

```
+-----+-----+-----+
| id | entrydate | earlier |
+-----+-----+-----+
| 1 | 2007-05-01 | 1 |
| 1 | 2007-05-02 | 2 |
| 2 | 2007-06-01 | 1 |
| 2 | 2007-06-02 | 2 |
| 3 | 2007-07-01 | 1 |
| 3 | 2007-07-02 | 2 |
+-----+-----+-----+
```

This works beautifully with smallish aggregates. But the query algorithm compares every within-group row to every other within-group row. As the size N of a group increases, execution time increases by N*N. If the query takes one minute for groups of 1,000, it will take 16 minutes for groups of 4,000, and more than four hours for groups for 16,000. *The solution does not scale.*

What to do? Forget GROUP BY! Manually assemble the desired query results in a temporary table from simple indexed queries, in this case, two rows per ID>:

```
DROP TEMPORARY TABLE IF EXISTS earlier;
CREATE TEMPORARY TABLE earlier( id INT, entrydate DATE);
INSERT INTO earlier
  SELECT id,entrydate FROM test WHERE id=1 ORDER BY entrydate LIMIT 2;
INSERT INTO earlier
  SELECT id,entrydate FROM test WHERE id=2 ORDER BY entrydate LIMIT 2;
INSERT INTO earlier
  SELECT id,entrydate FROM test WHERE id=3 ORDER BY entrydate LIMIT 2;
```

You need one INSERT statement per grouping value. To print the result, just query the earlier table:

```
SELECT * FROM earlier
ORDER BY id, entrydate;
```

```
+-----+-----+
| id | entrydate |
+-----+-----+
| 1 | 2007-05-01 |
| 1 | 2007-05-02 |
| 2 | 2007-06-01 |
| 2 | 2007-06-02 |
| 3 | 2007-07-01 |
| 3 | 2007-07-02 |
+-----+-----+
```

```
DROP TEMPORARY TABLE earlier;
```

Most useful reports run again and again. If that's the case for yours, automate it in a stored procedure: using a cursor and a prepared statement, auto-generate an INSERT statement for every grouping value, and return the result:

```
DROP PROCEDURE IF EXISTS listearliers;
DELIMITER |
CREATE PROCEDURE listearliers()
BEGIN
  DECLARE curdone, vid INT DEFAULT 0;
  DECLARE idcur CURSOR FOR SELECT DISTINCT id FROM test;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET curdone = 1;
  DROP TEMPORARY TABLE IF EXISTS earliers;
  CREATE TEMPORARY TABLE earliers( id INT, entrydate DATE);
  SET @sql = 'INSERT INTO earliers SELECT id,entrydate FROM test WHERE id=?
  ORDER BY entrydate LIMIT 2';
  OPEN idcur;
  REPEAT
  FETCH idcur INTO vid;
  IF NOT curdone THEN
  BEGIN
    SET @vid = vid;
    PREPARE stmt FROM @sql;
    EXECUTE stmt USING @vid;
    DROP PREPARE stmt;
  END;
  END IF;
  UNTIL curdone END REPEAT;
  CLOSE idcur;
  SELECT * FROM earliers ORDER BY id,entrydate;
END;
|
DELIMITER ;
CALL listearliers();
```

About the author

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

MySQL Table Checksum How-To

by Baron Schwartz

Have you ever wondered if your slaves have the same data as their masters? Have you ever needed to see if a table or tables have the same data across many MySQL servers simultaneously? You probably have, especially if you use replication. If you use your replication slave for backups, keep reading unless you don't mind your backups being silently corrupt!

MySQL Table Checksum solves these problems efficiently. This article explains how it works and shows how to use it. It also shows you how to repair slave servers that have different data from their masters.

MySQL Table Checksum is part of the MySQL Toolkit project, available from SourceForge at <http://mysqltoolkit.sourceforge.net/>

How It Works

MySQL Table Checksum is a wrapper script around several kinds of table checksumming algorithms. You may already be familiar with MySQL's own CHECKSUM TABLE command, which is built into the server and returns an integer checksum of the table's contents. MySQL Table Checksum can use this command, but there are limitations: it only works in some MySQL versions and can't be used in an INSERT/SELECT statement. We'll see later why that's important.

I say it's a wrapper script because auto-generating checksum queries is only a small part of what it does for you. It handles a lot of administrative work, such as locking and consistency. These are necessary for comparing table contents across servers.

Aside from using MySQL's CHECKSUM TABLE command, it can generate two kinds of checksum queries, using cryptographic hash functions, that can be used in regular SELECTs. The first is a cumulative, order-dependent query that works by checksumming each row, concatenating it to a user variable, then setting the user variable to the checksum of the result. After "accumulating" all the row checksums into the user variable this way, the variable has a checksum of the table's contents.

The benefit to this approach is it takes very little memory (just enough to process each row and hold the user variable) and is very fast. The downside is the resulting checksum is order-dependent. That is, two tables with the same data but with differently ordered rows will appear to have different data, according to this checksum.

If this is an issue for you, MySQL Table Checksum also offers an order-independent checksum, using BIT_XOR. This query checksums the whole table as one aggregate, and splits the checksum into smaller parts that can be converted into BIGINT and BIT_XOR-ed together. This reduces the entire table to a few BIGINTs, which can then be smashed back together into a single value. (Even though cryptographic hash functions return alphanumeric data, it's really just a number in base 16, so it's easy to convert to an ordinary number for mathematical functions). This method is also very efficient, and has the advantage that BIT_XOR is order-independent over the whole group. This is cryptographically strong and doesn't suffer from any artifacts other order-independent aggregate functions, such as SUM(), exhibit.

In addition to simply checksumming rows, the queries have some extra tidbits added to make the calculations more efficient, handle NULL-able columns, and so on.

If you want to examine the queries MySQL Table Checksum writes, it's easy to do, but I warn you they might not be easy to understand! They are delicately tweaked to work around lots of little special cases. Hopefully the above explanation will help if you have any questions.

What the Wrapper Script Does

MySQL Table Checksum does a lot more than just inspecting table structures and writing appropriate queries to checksum them. It uses these checksum queries as building blocks for much larger tasks. The two major ways it operates are checksumming a bunch of servers simultaneously, and verifying replication integrity. I'll show you how to do each of these.

First, let's see how to do fast, non-blocking, parallel checksums across many servers at once. Let's say we just want to check that the same users and privileges exist on each server -- we'll checksum the mysql database. Here's a command, and its slightly abbreviated output:

```
mysql-table-checksum -d mysql server1 server2
```

DATABASE	TABLE	HOST	COUNT	CHECKSUM
mysql	columns_priv	server2	0	NULL
mysql	columns_priv	server1	0	NULL
mysql	db	server1	14	8df4f3998d6ba6f65eb05802a6cbcb38bd7855ec
mysql	db	server2	14	8df4f3998d6ba6f65eb05802a6cbcb38bd7855ec
mysql	tables_priv	server1	17	3fcb0d0ce4be6713edcb4ec0afb5f5b7e796a190
mysql	tables_priv	server2	17	3fcb0d0ce4be6713edcb4ec0afb5f5b7e796a190
mysql	user	server1	10	49f1e210045f3653946090d9b08736be018ae378
mysql	user	server2	10	49f1e210045f3653946090d9b08736be018ae378

As you can see, the tables are identical, so the privileges are the same. Take a closer look at the first two rows of output; server2's output came before server1. This is because the queries are truly simultaneous. MySQL Table Checksum forks off multiple processes to handle the servers, and they might not finish in the order they started. By the way, there's a helper script, called MySQL Checksum Filter, which will filter out every table that's the same on all servers, so you can easily see only tables that differ.

You can really do this on dozens or hundreds of servers at once if you need to. The machine on which you run MySQL Table Checksum does virtually no work, and there is almost no network traffic. It's extremely efficient.

Verifying Replication Consistency

The second use case, which is what I use most of the time, is to checksum a replication master and slaves, to ensure the slaves have the same data as the master. If you think your slaves have the same data but have never checked, try it. Almost everyone I've talked to who's checked has found, to their surprise, that replication bugs and other problems cause slaves to differ silently and invisibly from their masters. I see everything from extra rows that don't get deleted on the slave, to `TIMESTAMP` columns that have different values, to just outright different data.

Checksumming slaves against their masters is a little tricky. One way to do it is to lock each table on the master and slave, let the slave catch up to the master, checksum the tables, and then release the locks. MySQL Table Checksum can do this, if you want -- read the documentation on the website I linked above. But it's not the best way to do it, for several reasons. The most obvious is blocking updates because of the table locks.

A better way to do it is to let replication handle it natively like this:

```
INSERT INTO <checksum> SELECT <checksum query> FROM <each table>;
```

If you run this statement on the master, it will propagate through replication to the slave and run there, too. No locking is needed, and replication takes care of consistency. This is why the cryptographic hash checksums are necessary; they can be used in INSERT/SELECT statements, whereas CHECKSUM TABLE can't.

Once the slave has finished running the replicated checksum query, you can go to the slave and compare the contents of the "checksum" table against the master. If they match, the slave has the same data.

Of course, MySQL Table Checksum takes care of all this for you, very conveniently. Here's how.

First, we need to create the table that'll hold the results of the checksums:

```
CREATE TABLE test.checksum (  
    db      char(64) NOT NULL,  
    tbl     char(64) NOT NULL,  
    chunk   int     NOT NULL,  
    boundaries char(64) NOT NULL,  
    this_crc char(40) NOT NULL,  
    this_cnt int     NOT NULL,  
    master_crc char(40) NULL,  
    master_cnt int     NULL,  
    ts      timestamp NOT NULL,  
    PRIMARY KEY (db, tbl, chunk)  
);
```

This table will end up holding the results of the INSERT/SELECT, as well as the master's checksum and count values inserted literally, so they can easily be compared on the slave. Next, just run MySQL Table Checksum on the master with the --replicate argument. Here's the absolute minimum you need to run:

```
mysql-table-checksum --replicate=test.checksum master_host
```

However, I suggest adding some extra arguments to avoid slowing things down too much. There are a lot of options you can tweak, but the ones I use the most often are to checksum tables in small pieces and sleep between each piece. I also skip certain databases and tables; some of them might only exist on the master, for example, and would break replication on the slave. Here's what I typically run:

```
mysql-table-checksum --replicate=test.checksum --emptyrepltbl --ignoredb scratch,mysql --chunksize 100000 --sleep-coef=2  
master_host
```

In order, these arguments do the following:

- Store the checksums into the test.checksum table
- Empty that table before beginning
- Don't do the scratch or mysql databases
- Do tables in chunks of about 100,000 rows
- After every chunk, sleep twice as long as the chunk took to run

After it's finished running and the slaves have caught up, I run the following command to check the results:

```
mysql-table-checksum --replicate=test.checksum --recursecheck master_host
```

This uses SHOW SLAVE HOSTS to discover all the slaves and recurse through them, running an ordinary SQL query against test.checksum to discover tables that differ from the master. It outputs any differences it finds.

What if You Find Differences?

If you find differences, you're in good company! Most people find replication isn't working as well as they thought. But what can you do about it?

The traditional advice is to re-initialize the slave. If you cringe at this thought, there's still hope for you. First of all, you can probably repair just the tables, and even just the parts of tables, that have differences. Take a look at the "boundaries" column in the test.checksum table. If you used a chunking strategy, as I showed above, you can probably just repair the chunks that are bad, and that column shows the WHERE clause you can use.

You can use mysqldump to select the offending rows from the master and repair the slave. You can even do the whole table, if you must. But this has its hazards! Anytime you change the slave directly, you're subverting replication and possibly causing further problems.

A better method is to change rows on the master to their present value. This will not affect the rows on the master, but the statements will go through replication and fix the slave. You can use statements like REPLACE and INSERT IGNORE to do this.

It's sometimes pretty hard to figure out what statements you need to write, though. Sometimes your slave will have missing rows, extra rows, and different rows -- all on the same table. Another tool in the toolkit is MySQL Table Sync, which is like rsync for MySQL tables. Give it a shot; it works through replication and finds differences very efficiently. I use it to repair tables that have drifted out of sync on my slave servers. It's a work in progress, but it's complete enough to be very useful.

Conclusion

I hope this article has shown you how to compare MySQL tables and what to do about it when they're different. If you have more questions, please read the online documentation. It's very complete and explains things in much more detail than I can in this short article. If you have questions it doesn't answer, file a bug at the SourceForge project and I'll update the documentation.

If you have any troubles with these or other tools in the toolkit, please file bugs or use the mailing list at SourceForge.

About the author

Baron Schwartz works for the Rimm-Kaufman Group, a search marketing agency in Charlottesville, Virginia. He writes frequently about MySQL on his blog (<http://www.xaprb.com>). He is the author of the innotop MySQL and InnoDB monitor, and MySQL Toolkit, a collection of command-line tools.

Optimizing MySQL with Rails - Part Two

By Dan Buettner
drbuettner@gmail.com

Beware sample data and development laptops

Use real data, and a realistic amount of data, in your database when doing performance testing. Typically, development databases have a set of sample data, often small, often outdated, and that's generally fine for writing code that works – but don't rely on that small, stale dataset when it comes time to test and improve database performance. You want to test MySQL's query optimizer against what it will be working on in the real world, so grab a snapshot of your production database and use it when evaluating ways to increase performance. The `mysqldump` utility is very handy for this – you can dump a table from your live database to your test database with a one-liner:

```
mysqldump -h live.domain.com -u user -ppass live_db --tables employees | mysql -h test.domain.com -u user -ppass -D test_db
```

Doing this will also help ensure that you're testing against tables without any different or extra indices you may have created in development or in earlier testing and forgotten about, as well as with any changes that were made in production.

In the same vein, performance testing is best done with MySQL server settings and hardware as close as possible to your production environment – while you can optimize SQL based on EXPLAIN output just fine on your development laptop, you may find that queries against large tables or complex multi-table queries are simply faster on a production database server because more indices and more data fit into the memory caches, disks are faster, etc., and you could end up chasing a problem query on your laptop that is not a problem in production.

Databases evolve over time, usually growing, and what works early on may be less than optimal later, so revisit your queries and indices from time to time.

MySQL's query cache

MySQL from version 4.1 forward offers the “query cache”, which is essentially a hash structure that stores specific queries and their results. This allows for potentially huge performance increases in situations where queries are frequently repeated and data is static or changes infrequently. Many, if not most, Web sites fit this pattern and might benefit greatly from the query cache. There is some performance penalty associated with enabling it – estimates vary on the severity – but the potential return is incredible.

Bear in mind that the query cache is only effective for identical queries that are repeated, and where the underlying data does not change. Data changes invalidate any related queries in the cache.

One situation where the query cache cannot help is when using SQL functions such as `NOW()`, since `NOW()` obviously changes all the time. If you have a page that loads a list of songs released “today” with a query like:

```
:conditions => "published_on = DATE(NOW())"
```

the query cache will not be helpful; instead, consider establishing today's date programmatically to generate a query that could live in the query cache all day:

```
:conditions => ["published_on = ?", DateTime.now.strftime("%Y-%m-%d")]
```

Rails' query cache

Rails 2.0 promises to include a query cache of its own, similar in concept and effect to MySQL's query cache; this feature will be on by default for Rails 2.0 users.

Datatype mixing

Be sure that your datatypes for columns are appropriate, and that they are consistent for columns you use to join tables. Otherwise, MySQL will have to cast the data to a different type before searching or joining – which is expensive on its own, and negates the possibility of using an index.

Some examples of poor datatype choices that can result in performance hits:

An integer of one size (say INT) in table t1 joined on an integer of another size (say BIGINT) in table t2

An unsigned INT in table t1 joined on a signed INT in table t2

An integer stored in a VARCHAR in table t1 joined on a proper integer in table t2

A date or datetime value stored in a VARCHAR

Move BLOBs to separate tables to lighten the load

Regardless of how simple your data structure, when you ask for a record, MySQL has to read the data from disk or memory and send it over the network or through a socket, and Ruby has to instantiate an object and put the database result into it. If you have BLOB fields (TEXT or BINARY datatypes), that can be a lot of data flowing between your database and your web app, and chances are you don't need most BLOB fields as often as you need the metadata. Break out your BLOB into a separate table and class with a has_one/belongs_to relationship to the original class, and only load when needed.

Use eager loading to reduce number of queries executed

You can specify in your model file that associated objects should be loaded at the same time as the original object, to reduce repetitive queries. In a has_many or has_and_belongs_to_many declaration, for example, the :include parameter does this:

```
class WordPress::BlogPost < ActiveRecord::Base
  has_and_belongs_to_many :categories, :class_name => 'BlogPostCategory', :join_table => "wp_post2cat", :include
=> 'segments', :foreign_key => 'post_id', :association_foreign_key => 'category_id', :order => 'cat_name'
```

Here we've linked WordPress posts to items in a table of our own creation, segments. Without the :include => 'segments' parameter, once a blog post and its WordPress categories are loaded, a separate SELECT is issued for each segment linked to a WP category. With the :include parameter, a single statement is issued for the post and categories and segments, reducing communication overhead, query parsing time, and taking advantage of the power of a relational database: they live to join tables, and do it exceedingly well.

Beware the use of functions on indexed columns

Using SQL functions when querying on indexed columns will frequently prevent MySQL from taking advantage of the index, so try to find ways to avoid this.

Example: you have an indexed date column `birth_date` tracking people's birthdates and want to find everyone born in a given year. It's tempting and easy to use something like

```
:conditions => ["YEAR(birth_date) = ?", year]
```

However, the index will not be used, at least with MySQL today. Knowing that every year starts on January 1 and ends on December 31, however, you can rewrite this query so that it can use the index:

```
:conditions => ["birth_date >= ? AND birth_date <= ?", year.to_s + '-01-01', year.to_s + '-12-31']
```

Cache Rails partials

One of the fastest ways to speed up Rails-MySQL interaction is to not hit the database at all, at least for elements that change infrequently.

```
<% cache(:controller=>'content_items', :part=>'leftnav') do -%>
  <%= render(:partial => "shared/sectionnav") %>
  <%= render(:partial => "shared/segmentnav") %>
<% end -%>
```

A "gotcha" is that the partials are not re-created when the database changes. We have an automated publishing process to move content from review in our staging environment database into our production database, and part of that process involves calling a standard rake task to clear the cache when new content arrives:

```
rake tmp:cache:clear
```

Use the `:limit` argument in Rails whenever you can (especially with `:first`)

No matter how fast your database is, it takes time to locate, sort, and return result sets. It also takes time for a Rails app to digest those result sets, as it instantiates an object from each row of the results and stores them in an array.

A trick I discovered while optimizing some functions on our web site was to use both the `:first` and `:limit` arguments for improved performance, instead of `:first` by itself.

In a case where I want to locate the most recent article published, I might use a call like this:

```
Article.find :first, :order => 'pub_date DESC'
```

which will work quite well, especially if I don't have lots of articles. There is a hidden cost to this, though, in that Rails does not, as you might expect, put a "LIMIT 1" clause in the SQL it issues to the database. Instead, it retrieves all articles from the database in `pub_date` descending order, instantiates each, stores them in an array, and then returns the first object from that array to the caller. The cost can vary greatly, and was up to several seconds for a couple of our large tables - only a fraction of which was database time.

By simply adding a `:limit` argument to the call, like so:

```
Article.find :first, :order => 'pub_date DESC', :limit => 1
```

I tell Rails to add a “LIMIT 1” clause to the SQL it issues. This means Rails gets at most one row back from the database, eliminating the overhead described above. Also, in some cases, the database will need to do less work to satisfy the query. From the perspective of the calling method, the results are the same - it still gets an object back through the use of the `:first` argument. This means faster performance for your site with minimal changes to your code.

Next steps

We've looked at several options for increasing performance, options that can be very effective and should be tried before anyone throws hardware at a performance problem. As with any such problem, keeping a careful record of changes made and their effects will be helpful to avoid duplicated effort - and using more than one approach to improve performance is practically required these days, as there is no one magic bullet that will cause all your queries to return instantly.

For a site of considerable size, a single MySQL server simply may not cut the mustard. Scaling your hardware up (more RAM, faster disks, more CPUs) is of course a possibility, as is scaling out with a MySQL cluster or replication. MySQL has free whitepapers available on their site for scaleout and clustering, and there are a number of resources elsewhere on the Internet as well.

I hope this article has been useful, and if you have any comments or suggestions, please send me a note: drbuettner@gmail.com

About the author

Dan Buettner works for Outsell, Inc., a San Francisco area market research company tracking the information industry, though he lives in Des Moines, Iowa. In a past life he worked at The Des Moines Register, a midsize newspaper, as an IT manager, newspaper production specialist, software developer, and Sybase and MySQL DBA. He has been using MySQL since version 3.x in 1999, and Rails since Summer 2006. This article was finished while attending RailsConf 2007.

Index Optimization in MySQL

by Keith Murphy

When you are looking to optimize MySQL there are three parts you can examine: the underlying hardware and operating system of the server, the MySQL daemon and its various parameters itself and the queries that are being run. We will be examining the third part to this equation. Specifically we will be examining how indexes can affect your query performance.

Indexes are data structures used to speed up access to data in your database. If your database did not use indexes every time it went to find data it would have to do a "table scan" where it starts at the start of the table and reads through it until it finds the data it is seeking. This is almost always slower than using an index which points to where the data is located so the database can go straight to the right data.

Indexes are not a panacea. They do have a cost. Whenever you do a write to the database, any table that is indexed will be slower to update because of the overhead of the index itself. Even so, it is almost always best to have at least one index on a table (typically a primary key index). Beyond this, it will vary greatly depending on your table structure and your SQL queries.

There are three types of indexes used by MySQL.. When you create a table you can specify indexes using the following keywords:

INDEX - there are no restrictions on the data in the field

UNIQUE - every row in the field is required to be unique

PRIMARY KEY - a unique index where all key columns must be defined as NOT NULL. If they are not explicitly declared as NOT NULL, MySQL declares them so implicitly (and silently). A table can have only one PRIMARY KEY.

In addition to these three basic types of indexes there is a composite index (an index on multiple columns). This can be quite useful. Suppose that you have a table with a composite index on columns state, city, and zip. The index would sort on an order of state, city and zip code. One advantage of a composite index is that MySQL can use the index to sort state values, state and city values or state, city and zip code value. However, it could not sort based on zip code values alone. Nor could it sort based on city value alone..

An index can also be created on part of a column. For example, say you had a column holding the 10-digit number of an American phone number. You could create an index on the first three digits and use it when searching by zip code. It would save space compared to an index on the entire column. That is somewhat of a contrived example as you would most often want to index the entire column but don't forget about partial indexes. There are times when they prove very useful.

Now that you understand the basics of indexes we will discuss a couple of other fundamental concepts that are used when determining the optimum index usage.

Covering Indexes

Covering indexes can be very useful. A covering index is an index that encompasses every field that is returned by the SELECT statement. This reduces the number of reads required to return the data.

As an example, let's look at the following query: `SELECT last_name from customers where customer_id=20`

If `customer_id` is indexed only then MySQL will have to perform two reads to return the query results. The first read is of the index to find the row pointer and the second read is to actually read the row. If you have a covering index of `(customer_id,last_name)` you will only have a read of the index. All the necessary information is stored in the index.

Duplicate Indexes

When table has multiple indexes defined on the same columns it has duplicate indexes. Often the indexes simply have different names. An amateur mistake is to define the same index with different index key-types. An example of this would be indexes defined like this: PRIMARY KEY(id), INDEX id2(id). The logic is that you use the primary key to enforce uniqueness and use the INDEX id2(id) index in your query. This will slow down the performance of your queries. You can use the PRIMARY KEY index in your queries with no problem.

MySQL's laxness allows you to create multiple keys on the same column. MySQL manages these separately and each index will take up disk space and memory. Updates to the table will require more time. There really is no legitimate reason to have duplicate indexes in your table schema.

Tools for Indexes

How do you know what is going on with your indexes? There are some fundamental ways built into MySQL to determine index information. First let's create a simple table:

```
mysql> create table customers (customer_id int not null auto_increment,  
-> primary key(customer_id),  
-> last_name varchar(25));  
Query OK, 0 rows affected (0.03 sec)
```

If I hadn't created the table and it was in my database I could use the DESCRIBE command to begin my analysis:

```
mysql> describe customers;  
+-----+-----+-----+-----+-----+-----+  
| Field      | Type      | Null | Key  | Default | Extra      |  
+-----+-----+-----+-----+-----+-----+  
| customer_id | int(11)   |      | PRI  | NULL    | auto_increment |  
| last_name   | varchar(25) | YES  |      | NULL    |              |  
+-----+-----+-----+-----+-----+-----+  
2 rows in set (0.00 sec)
```

This shows that in this table we have one primary key.

If you have a complex indexing scheme the DESCRIBE command won't show enough information. You can use the 'SHOW INDEX FROM tablename' command to get better information:

```
mysql> show index from customers\G
***** 1. row *****
  Table: customers
  Non_unique: 0
  Key_name: PRIMARY
  Seq_in_index: 1
  Column_name: customer_id
  Collation: A
  Cardinality: 197138
  Sub_part: NULL
  Packed: NULL
  Null:
  Index_type: BTREE
  Comment:
  1 row in set (0.00 sec)
```

While we won't be able to go into much depth, the EXPLAIN command is very useful. The output of a simple query on the previous table:

```
mysql> explain select last_name from customers where customer_id=34450;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type | possible_keys | key      | key_len | ref  | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | customers | const | PRIMARY      | PRIMARY | 4       | const | 1    |       |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Some important columns here are “key” which shows that it is using a key and it is a primary key and the “rows” column showing that it returns one result row (as would be expected). The “possible_keys” column shows which potential indexes are available. Next time we will be using the EXPLAIN command quite a bit. I will show the output and explain what you need to know.

Coming next time

Next time we will take a look at how different indexing schemes affect benchmarks performed against sample data. Once you combine an understanding of what indexes are and the types of indexes available in MySQL along with the tools available to work with indexes and queries you can begin to build a strong foundation of how to speed up your application.

Resources

SQL Tuning by Dan Tow

MySQL Database Design and Tuning by Robert D. Schneider

<http://www.mysqlperformanceblog.com/>

About the author

Keith Murphy is the DBA at iContact (formally Broadwick). He gets to play all day with fun toys developing a better environment for the data of iContacts users. Currently he is especially fascinated with his latest toy - a NAS server which he will use to rule the world (or consolidate all his data ... whichever comes first). He tries to write on a reasonable schedule all the latest happenings at his blog at <http://www.paragon-cs.com/wordpress>.

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

I just want to say thanks to the people in the community. From the fabulous writers who contribute stories, to the people who took the time to read the first issue. I put out one email asking for writers and one email when I released the magazine. I have some great people who responded to write articles and we had over 1300 downloads of the first issue.

That is incredible!! As I said in the first issue, I thought there was a need for a MySQL magazine. It appears that I am right. I am so glad that I chose to work with people in the open source community. There are no more passionate people to work with. I am looking forward to this venture and seeing where it goes. Even more, I am excited about where MySQL is going. We have a solid product and it only continues to improve.

There has been much ink spilled about the recent announcements of MySQL regarding the (non)release of the Enterprise codebase. While technically it is still available (as it must be under the GPL license), it is essentially hidden.

There are several interesting points to this discussion in my opinion. First, I think that MySQL is trying to both improve their bottom line and their product. I don't think they are trying to be "evil" in any sense. Unfortunately, I also think that this won't really help improve the product. On the other hand, I don't think it will really hurt it either. Another point that was raised in the "blog-o-sphere", first by Matt Asay, is that the Open Source community tends to get upset when companies seemingly close things up (such as when Redhat did the whole Enterprise/Fedora split). And those same people get all excited when companies such as "When IBM, Adobe, Oracle, Novell, SAP, or other proprietary companies release even a modicum of code, they get universal plaudits. When 100-percent open-source companies like Red Hat and MySQL rejig the way they release code (while still releasing it and fully complying with open-source licenses), they get tarred and feathered." This came from Matt's posted on his CNET blog.

He thinks this is hypocritical. I think it is people being unrealistic. Companies operate to make money. They may operate, as MySQL does, using open source licensing with the idea that this will help them build a better product. I happen to think that it will help exactly that happen. Companies and people both make wrong decisions from time to time. Time will tell if this decision was right or wrong.

I guess I am just pragmatic about the whole thing. If MySQL AB really hoses things up there is always the ability to either split the code or use another product. That is not something I would want to do or even contemplate, but it is possible. That is what open source is all about - freedom of choice.

Feel free to write me at bmurphy@paragon-cs.com. Let me know what you think about this issue and I will gladly set up a "reader feedback" column for the magazine.

Thanks again,

Keith Murphy (bmurphy@paragon-cs.com)