THE COMPREHENSIVE LAMP GUIDE - II

MySQL is the most widely used Relational Database Management System (RDBMS), and is the "M" in the LAMP stack. One of the main uses for MySQL is with Web applications. To avoid the database turning into the bottleneck in Web applications, let us set up a MySQL server and tune it for maximum performance. At the time of writing, the latest (stable) version of MySQL is 5.5.14, and we'll be using that here.

Though MySQL provides binaries for various Linux distributions and even FreeBSD, we're not interested; we need performance, and so we'll be compiling our own. As in the previous article, I do not claim that optimisations will definitely increase performance; sometimes, they can cause degradation as well, because it depends more on hardware, CPU cache and DRAM.

MySQL, unlike the traditional `./configure` build methods, uses CMake, an advanced and easy-to-use build system with a nice UI called `ccmake` (command-line) or `cmake-gui` (the GUI, if you run X). However, we'll specify options at the command line itself. Along with CMake, you'll also need a C++ compiler installed. This usually comes with GCC in most distributions, but sometimes is in a separate package (e.g., Fedora has `gcc` and `gcc-c++`).

To download the MySQL source tarball, navigate to the downloads page on the MySQL website, and in the platform drop-down list, choose "Source Code". When the download option is displayed, make a note of the MD5 sum in a text file. After the download is over, to ensure you have the original file and not a tampered version with security holes, verify the file checksum as follows.

Copy the saved MD5 sum to the clipboard, and paste it at the shell prompt, to assign to a variable, as shown; then run the following commands:

```
$ ORG_SUM=&amp;lt;Paste the MD5 sum here&amp;gt;
$ FLE_SUM=$(md5sum mysql-5.5.14.tar.gz | awk -F ' +' '{ print $1 }')
$ [ "$ORG_SUM" == "$FLE_SUM" ] &amp;amp;&amp;amp; echo verification success || echo verification failed
```

If your download was not okay, redownload it, else read on.

Source code configuration

CMake accepts options on the command line with the -D switch; for example, cmake -D<OPTION NAME>=<OPTION VALUE>. Significant options while configuring the code-base are:

- CMAKE_INSTALL_PREFIX: The location where the MySQL server will be installed.
- COMMUNITY_BUILD: Enables/disables the community features in MySQL.
- ENABLED_PROFILING: Enables/disables query profiling code.
- MYSQL_DATADIR: This is the default directory where data will be stored. It can be set in the configuration file as well, after installation.
- WITH_INNOBASE_STORAGE_ENGINE: Enables/disables InnoDB, a transactional storage engine.
- WITH_DEBUG: Enables/disables debugging capability; this increases the binary size, but can be very useful while getting support from forums, etc, in case your installation fails for some reason. A related option is ENABLE_DEBUG_SYNC.
- WITH_PARTITION_STORAGE_ENGINE: Enables/disables the partition storage engine.
- WITH_SSL: Enables/disables SSL support in the server and client. Enable this if you see any need for replication in the future; unencrypted replication is a risk.
- CMAKE_CXX_FLAGS: Sets the C++ compiler flags -O3 -march=native -mtune=native -msse -msse2 -mmmx (provided you have a CPU with a decent cache size — else use -Os instead of -O3).
- CMAKE_C_FLAGS: Sets the C compiler flags. MySQL has no C code, so this is not required. You might want to set it to the same value as CXXFLAGS, for sanity.
- DISABLE_SHARED: Builds MySQL statically, with all plugins compiled in. This is not preferred; it will unnecessarily consume memory for unwanted features/plugins.

I have described the most important options, all of which are enabled by default. If you want a look at many more, use ccmake or cmake-gui, or refer to the MySQL documentation here.

Server configuration

MySQL's single configuration file, my.cnf, is located in CMAKE_INSTALL_PREFIX/etc orMYSQL_DATADIR (deprecated). You can also pass the location of the configuration file to mysqldwhile starting it. Some of the configuration options in my.cnf in the [mysqld] section are:

- bind-address: The address on which the server will listen for connections. If you're only going to connect from localhost, don't use this; instead, use the skip-networking option, as a security measure.

- user: The account as which mysqld will run; again, a security measure. This user must have R/W access to MYSQL_DATADIR.
- datadir: By default, the compile-time path in MYSQL_DATADIR. Change it if you want to relocate the data directory.
- character-set-server: Keep as UTF8 (Unicode), unless you have special reason to change it.
- character-set-client: Same as the previous option, but this will tell the client to use the specified character set.
- default-character-set: The default set to use while creating tables.
- default-storage-engine: This default (MyISAM) will be used if an engine is not specified in the CREATE statement.
- skip-innodb: Disable the InnoDB storage engine. If you statically compiled MySQL, you need to use ignore-builtin-innodb.
- key_buffer: The MyISAM key buffer; should be kept as large as possible (of course, sparing DRAM for other applications and services). The larger it is, the better the performance. On a dedicated MySQL server, it is usually at least a quarter of the total memory (and not more than half). It should be enough to hold all the MyISAM indexes (.MYI files) in memory. This can be checked via variables, which can be obtained by the SHOW VARIABLES command at the MySQL command prompt, or using tools like phpMyAdmin. Obtain these variable values, and divide key_read by key_read_requests; the value should be < 0.01. And if you divide key_writeby key_write_requests, the value should be < 1.
- table-cache: Every time MySQL opens a table, it saves it in cache, to speed up access. The variable opened_tables in server status will tell you what you need to set this to; some use cases have seen values as high as 20,000. Keep opened_tables as low as possible.
- read_buffer_size: The default is 128K, and usually this is set to 1M or 2M. The performance depends on CPU cache, disk speed and other factors; you should run a test in your environment, as described here.
- query_cache_type: The value 0 means the query cache is disabled; 1 implies that the query cache is enabled for all queries that don't have SQL_NO_CACHE specified in the query; and 2 means that no query is cached unless SQL_CACHE is specified in the query.
- query_cache_size: The larger the better, but also check the available DRAM.

- ♣ query_cache_limit: The maximum size of the result set that will be stored in the query cache. This should be large if you have queries producing large result sets and have a lot of DRAM to spare. Watch the variables %Qcache% in the server status to see how well the query cache is being used.
- ♣ tmp_table_size: MySQL sometimes needs to create temporary tables automatically, depending on the query. When the result set grows larger than the size specified here, the temporary table is converted to a disk-based temporary table, which results in a loss in performance. Keep this size as large as possible, considering the available DRAM and other applications running on the site. In the server status, observe created_tmp_disk_tables.
- ♣ max-connections: The maximum number of simultaneous connections. This is limited by CPU power and the available DRAM.
- ♣ sort_buffer: This should be set to a high value if your queries involve a lot of sorting. This seems to have some negative effect on performance as well, so do a few test cases before and after changing the value.
- ♣ thread_cache: If you have a lot of very-short-duration connections, increase this value till thehreads_created value stops increasing in the server status, to reduce CPU usage.
- ♣ long_query_time: This is the time a query will be allowed to run before it is considered slow and logged in slowlog. Keep this value small — say 5s; the default value is 10s.
- ♣ slow_query_log_file: This is the path to the slow query log file. MySQL 5.5 has a new option,log-queries-not-using-indexes. If specified, it will send only slow queries that are not using indexes to the slow query log.

Tips for designing efficient tables and queries

Although I have talked a lot about options and optimisations, the actual optimisation of MySQL is done by properly designing tables and the database. Also, it depends on how efficient your queries are. Generally, one query can be executed in multiple ways, and each way has its own advantages and disadvantages. We need to choose a query that suits our environment and isn't costly on the performance side. Here are some points:

- ♣ Keep the design simple; split your columns across tables. Columns you don't access often should be put in a different table, to avoid memory wastage when the table is cached. You can link data across tables using foreign keys, which are supported only in InnoDB, and not in MyISAM.

- Use MyISAM if you are going to read a lot and write less — and InnoDB otherwise. There are some absolutely astounding results in this aspect, and you should switch engines after proper testing.
- The columns frequently used in WHERE statements and in filtering results should be indexed. Indexes point to the data location on the disk. If indexes are not used, MySQL has to scan the whole table to find the required row, which is very costly in terms of performance.
- Your tables should conform to normal form rules, have normalised data, and use efficient column types.
- Keep queries simple and short. Longer queries take more time to execute. Use application-level caching.
- Turn on slow query logging in my.cnf by setting long_query_time and slow_query_log_file.
- Use EXPLAIN and ANALYZE to see what a slow query does, and how long it takes. This can give you a lot of hints for optimising it.
- Run OPTMIZE TABLE frequently. But be aware: this locks the table, so run it only when MySQL traffic is low. This basically rearranges disk files, reclaiming disk space consumed by deleted rows.

Since MySQL uses a lot of disk, actual performance is pretty much limited by disk performance, rotation speed, etc. Obviously, you need a high-speed disk. Another optimisation tip would be to use a filesystem that supports compression on the MySQL data partition, to reduce disk seeks; this is currently supported in ZFS (FreeBSD) and Btrfs. The latter is under development as of now, and not recommended for production use; try it only after proper backups.

Using RAID striping with a stripe size of 1 MB or so can help a lot, since the data is split across two disks, and reading from two disks is obviously faster than reading from one disk sequentially.

There's another possibility for speeding up disk performance, which is not actually tested: using SquashFS and AUFS. I have never tried it for a MySQL database directory but have used it on my system, and it does speed up things at the cost of CPU usage. What you basically do is compress /usr into a squashfs file — say, /squashed/usr.sqfs. Then, you mount usr.sqfs at/squashed/usr/ro (loop, readonly).
Using AUFS, you can mount /usr and specify read (/squashed/usr/ro) and write (/squashed/usr/rw) locations as its branches. The only problem with this method is that you have to re-squash whenever

large changes are made, in order to ensure performance. This is probably not feasible with MySQL in cases where the database will have a lot of writes.

More information on this squashing method can be obtained from this Gentoo Linux forum thread.

Please note that I'm not responsible for any loss. Backup before trying any tricks. :-

Source : http://www.opensourceforu.com/2011/08/comprehensive-lamp-guide-part-2-mysql/