

Top 20+ MySQL Best Practices

by Burak Guzel 25 Nov 2009 266 Comments



48



43



97

大多数Web应用的**主要瓶颈**

Database operations often tend to be the main bottleneck for most web applications today. It's not only the DBA's (database administrators) that have to worry about these performance issues. We as programmers need to do our part by structuring tables properly, writing optimized queries and better code. Here are some MySQL optimization techniques for programmers. 担心这些性能问题 下面是一些针对程序员的MySQL优化技术

1. Optimize Your Queries For the Query Cache

Most MySQL servers have query caching enabled. It's one of the most effective methods of improving performance, that is quietly handled by the database engine. When the same query is executed multiple times, the result is fetched from the cache, which is quite fast.

The main problem is, it is so easy and hidden from the programmer, most of us tend to ignore it. Some things we do can actually prevent the query cache from performing its task.

```
1 // query cache does NOT work
2 $r = mysql_query("SELECT username FROM user WHERE signup_date >= CURDATE()");
3
4 // query cache works!
5 $today = date("Y-m-d");
6 $r = mysql_query("SELECT username FROM user WHERE signup_date >= '$today'");
```

The reason query cache does not work in the first line is the usage of the CURDATE() function. This applies to all non-deterministic functions like NOW() and RAND() etc... Since the return result of the function can change, MySQL decides to disable query caching for that query. All we needed to do is to add an extra line of PHP before the query to prevent this from happening.

2. EXPLAIN Your SELECT Queries

Using the **EXPLAIN** keyword can give you insight on what MySQL is doing to execute

your query. This can help you spot the bottlenecks and other problems with your query or table structures.

The results of an EXPLAIN query will show you which indexes are being utilized, how the table is being scanned and sorted etc...

Take a SELECT query (preferably a complex one, with joins), and add the keyword EXPLAIN in front of it. You can just use phpmyadmin for this. It will show you the results in a nice table. For example, let's say I forgot to add an index to a column, which I perform joins on: 执行联接查询

Browser Structure SQL Search Insert Export Import Operations Emp

✓ Your SQL query has been executed successfully

```
EXPLAIN SELECT username, group_name
FROM users u
JOIN groups g ON ( u.group_id = g.id )
WHERE g.id
BETWEEN 1
AND 10
```

not using an index (key)

+ Options

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	u	ALL	NULL	NULL	NULL	NULL	7883	Using where
1	SIMPLE	g	eq_ref	PRIMARY	PRIMARY	4	test.u.group_id	1	

After adding the index to the group_id field:

Browser Structure SQL Search Insert Export Import Operations Emp

✓ Your SQL query has been executed successfully

```
EXPLAIN SELECT username, group_name
FROM users u
JOIN groups g ON ( u.group_id = g.id )
WHERE g.id
BETWEEN 1
AND 10
```

that's better!

+ Options

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	g	range	PRIMARY	PRIMARY	4	NULL	9	Using where
1	SIMPLE	u	ref	group_id	group_id	4	test.g.id	16	

Now instead of scanning 7883 rows, it will only scan 9 and 16 rows from the 2 tables. A good rule of thumb is to multiply all numbers under the "rows" column, and your query performance will be somewhat proportional to the resulting number.

3. LIMIT 1 When Getting a Unique Row

Sometimes when you are querying your tables, you already know you are looking for just one row. You might be fetching a unique record, or you might just be just checking the existence of any number of records that satisfy your WHERE clause.

In such cases, adding LIMIT 1 to your query can increase performance. This way the database engine will stop scanning for records after it finds just 1, instead of going thru the whole table or index.

应用场景：

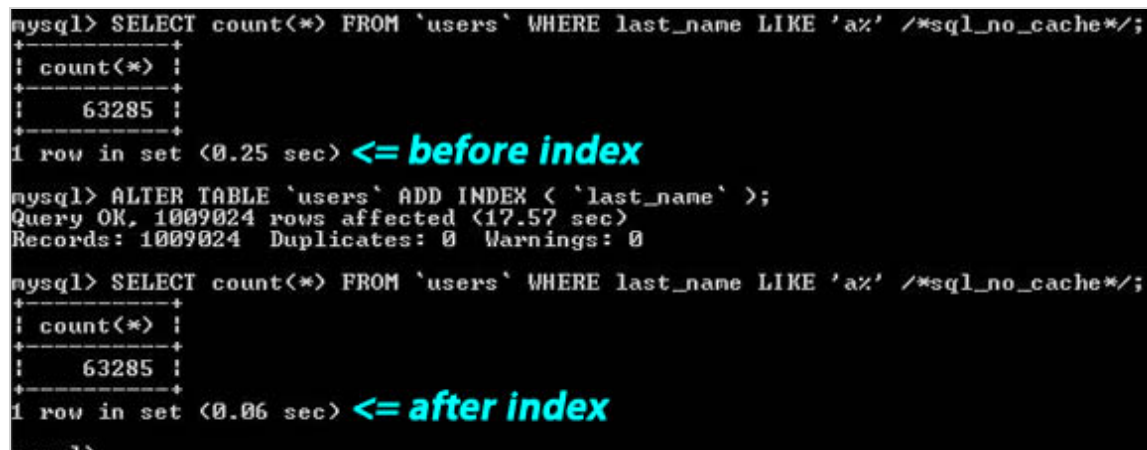
抓取一条记录

检查满足WHERE子句条件的记录是否存在

```
01 // do I have any users from Alabama?  检查记录是否存在
02
03 // what NOT to do:
04 $r = mysql_query("SELECT * FROM user WHERE state = 'Alabama'");
05 if (mysql_num_rows($r) > 0) {
06     // ...
07 }
08
09
10 // much better:
11 $r = mysql_query("SELECT 1 FROM user WHERE state = 'Alabama' LIMIT 1");
12 if (mysql_num_rows($r) > 0) {
13     // ...
14 }
```

4. Index the Search Fields

Indexes are not just for the primary keys or the unique keys. If there are any columns in your table that you will search by, you should almost always index them.



```
mysql> SELECT count(*) FROM `users` WHERE last_name LIKE 'az' /*sql_no_cache*/;
+-----+
| count(*) |
+-----+
| 63285 |
+-----+
1 row in set (0.25 sec) <= before index

mysql> ALTER TABLE `users` ADD INDEX (`last_name` );
Query OK, 1009024 rows affected (17.57 sec)
Records: 1009024 Duplicates: 0 Warnings: 0

mysql> SELECT count(*) FROM `users` WHERE last_name LIKE 'az' /*sql_no_cache*/;
+-----+
| count(*) |
+-----+
| 63285 |
+-----+
1 row in set (0.06 sec) <= after index
```

As you can see, this rule also applies on a partial string search like "last_name LIKE 'a%'. When searching from the beginning of the string, MySQL is able to utilize the index on that column.

You should also understand which kinds of searches can not use the regular indexes. For instance, when searching for a word (e.g. "WHERE post_content LIKE '%apple%'"), you will not see a benefit from a normal index. You will be better off using [mysql fulltext](#)

[search](#) or building your own indexing solution.

5. Index and Use Same Column Types for Joins

If your application contains many **JOIN** queries, you need to make sure that the columns you join by are indexed on both tables. This affects how MySQL internally optimizes the join operation.

Also, the columns that are joined, need to be the same type. For instance, if you join a DECIMAL column, to an INT column from another table, MySQL will be unable to use at least one of the indexes. Even the character encodings need to be the same type for string type columns.

```
1 // looking for companies in my state
2 $r = mysql_query("SELECT company_name FROM users
3     LEFT JOIN companies ON (users.state = companies.state)
4     WHERE users.id = $user_id");
5
6 // both state columns should be indexed
7 // and they both should be the same type and character encoding
8 // or MySQL might do full table scans
```

6. Do Not ORDER BY RAND()

This is one of those tricks that sound cool at first, and many rookie programmers fall for this trap. You may not realize what kind of terrible bottleneck you can create once you start using this in your queries.

If you really need random rows out of your results, there are much better ways of doing it. Granted it takes additional code, but you will prevent a bottleneck that gets exponentially worse as your data grows. The problem is, MySQL will have to perform RAND() operation (which takes processing power) for every single row in the table before sorting it and giving you just 1 row.

```
01 // what NOT to do:
02 $r = mysql_query("SELECT username FROM user ORDER BY RAND() LIMIT 1");
03
04
05 // much better:
06
07 $r = mysql_query("SELECT count(*) FROM user");
08 $d = mysql_fetch_row($r);
09 $rand = mt_rand(0, $d[0] - 1);
10
```

```
11 | $r = mysql_query("SELECT username FROM user LIMIT $rand, 1");
```

So you pick a random number less than the number of results and use that as the offset in your LIMIT clause.

7. Avoid SELECT *

The more data is read from the tables, the slower the query will become. It increases the time it takes for the disk operations. Also when the database server is separate from the web server, you will have longer network delays due to the data having to be transferred between the servers.

It is a good habit to always specify which columns you need when you are doing your SELECT's.

```
01 | // not preferred
02 | $r = mysql_query("SELECT * FROM user WHERE user_id = 1");
03 | $d = mysql_fetch_assoc($r);
04 | echo "Welcome {$d['username']}";
05 |
06 | // better:
07 | $r = mysql_query("SELECT username FROM user WHERE user_id = 1");
08 | $d = mysql_fetch_assoc($r);
09 | echo "Welcome {$d['username']}";
10 |
11 | // the differences are more significant with bigger result sets
```

8. Almost Always Have an id Field

In every table have an id column that is the PRIMARY KEY, AUTO INCREMENT and one of the flavors of INT. Also preferably UNSIGNED, since the value can not be negative.

Even if you have a users table that has a unique username field, do not make that your primary key. VARCHAR fields as primary keys are slower. And you will have a better structure in your code by referring to all users with their id's internally.

There are also behind the scenes operations done by the MySQL engine itself, that uses the primary key field internally. Which become even more important, the more complicated the database setup is. (clusters, partitioning etc...).

One possible exception to the rule are the "association tables", used for the

many-to-many type of associations between 2 tables. For example a "posts_tags" table that contains 2 columns: post_id, tag_id, that is used for the relations between two tables named "post" and "tags". These tables can have a PRIMARY key that contains both id fields.

9. Use ENUM over VARCHAR

ENUM type columns are very fast and compact. Internally they are stored like TINYINT, yet they can contain and display string values. This makes them a perfect candidate for certain fields.

If you have a field, which will contain only a few different kinds of values, use ENUM instead of VARCHAR. For example, it could be a column named "status", and only contain values such as "active", "inactive", "pending", "expired" etc...

There is even a way to get a "suggestion" from MySQL itself on how to restructure your table. When you do have a VARCHAR field, it can actually suggest you to change that column type to ENUM instead. This done using the PROCEDURE ANALYSE() call. Which brings us to:

10. Get Suggestions with PROCEDURE ANALYSE()

PROCEDURE ANALYSE() will let MySQL analyze the columns structures and the actual data in your table to come up with certain suggestions for you. It is only useful if there is actual data in your tables because that plays a big role in the decision making.

For example, if you created an INT field for your primary key, however do not have too many rows, it might suggest you to use a MEDIUMINT instead. Or if you are using a VARCHAR field, you might get a suggestion to convert it to ENUM, if there are only few unique values.

You can also run this by clicking the "Propose table structure" link in phpmyadmin, in one of your table views.

Std	Optimal_fieldtype
6.1138	MEDIUMINT(7) UNSIGNED NOT NULL
JLL	CHAR(32) NOT NULL
JLL	CHAR(32) NOT NULL
JLL	ENUM('active','expired','inactive','pending') NOT NULL
1.9902	SMALLINT(3) UNSIGNED NOT NULL

Keep in mind these are only suggestions. And if your table is going to grow bigger, they may not even be the right suggestions to follow. The decision is ultimately yours.

11. Use NOT NULL If You Can

Unless you have a very specific reason to use a NULL value, you should always set your columns as **NOT NULL**.

First of all, ask yourself if there is any difference between having an empty string value vs. a NULL value (for INT fields: 0 vs. NULL). If there is no reason to have both, you do not need a NULL field. (Did you know that Oracle considers NULL and empty string as being the same?)

NULL columns require additional space and they can add complexity to your comparison statements. Just avoid them when you can. However, I understand some people might have very specific reasons to have NULL values, which is not always a bad thing.

From MySQL docs:

"NULL columns require additional space in the row to record whether their values are NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte."

12. Prepared Statements

There are multiple benefits to using **prepared statements**, both for performance and security reasons.

Prepared Statements will filter the variables you bind to them by default, which is great for protecting your application against SQL injection attacks. You can of course filter your variables manually too, but those methods are more prone to human error and forgetfulness by the programmer. This is less of an issue when using some kind of framework or ORM.

Since our focus is on performance, I should also mention the benefits in that area. These benefits are more significant when the same query is being used multiple times in your application. You can assign different values to the same prepared statement, yet MySQL will only have to parse it once.

Also latest versions of MySQL transmits prepared statements in a native binary form, which are more efficient and can also help reduce network delays.

There was a time when many programmers used to avoid prepared statements on purpose, for a single important reason. They were not being cached by the MySQL query cache. But since sometime around version 5.1, query caching is supported too.

To use prepared statements in PHP you check out the [mysqli extension](#) or use a database abstraction layer like [PDO](#).

```
01 // create a prepared statement
02 if ($stmt = $mysqli->prepare("SELECT username FROM user WHERE state=?")) {
03
04     // bind parameters
05     $stmt->bind_param("s", $state);
06
07     // execute
08     $stmt->execute();
09
10     // bind result variables
11     $stmt->bind_result($username);
12
13     // fetch value
14     $stmt->fetch();
15
16     printf("%s is from %s\n", $username, $state);
17
18     $stmt->close();
19 }
```

13. Unbuffered Queries

Normally when you perform a query from a script, it will wait for the execution of that query to finish before it can continue. You can change that by using **unbuffered queries.**

There is a great explanation in the PHP docs for the [mysql_unbuffered_query\(\)](#) function:

"mysql_unbuffered_query() sends the SQL query query to MySQL without automatically fetching and buffering the result rows as mysql_query() does. This saves a considerable amount of memory with SQL queries that produce large result sets, and you can start working on the result set immediately after the first row has been retrieved as you don't have to wait until the complete SQL query has been performed."

However, it comes with certain limitations. You have to either read all the rows or call [mysql_free_result\(\)](#) before you can perform another query. Also you are not allowed to use [mysql_num_rows\(\)](#) or [mysql_data_seek\(\)](#) on the result set.

14. Store IP Addresses as UNSIGNED INT

Many programmers will create a `VARCHAR(15)` field without realizing they can actually store IP addresses as integer values. With an INT you go down to only 4 bytes of space, and have a fixed size field instead.

You have to make sure your column is an UNSIGNED INT, because IP Addresses use the whole range of a 32 bit unsigned integer.

In your queries you can use the [INET_ATON\(\)](#) to convert an IP to an integer, and [INET_NTOA\(\)](#) for vice versa. There are also similar functions in PHP called [ip2long\(\)](#) and [long2ip\(\)](#).

```
1 $r = "UPDATE users SET ip = INET_ATON('".$_SERVER['REMOTE_ADDR']. "') WHERE user_id = $user_id";
```

15. Fixed-length (Static) Tables are Faster

When every single column in a table is "fixed-length", the table is also considered "static" or "fixed-length". Examples of column types that are NOT fixed-length are: `VARCHAR`, `TEXT`, `BLOB`. If you include even just 1 of these types of columns, the table ceases to be fixed-length and has to be handled differently by the MySQL engine.

Fixed-length tables can improve performance because it is faster for MySQL engine to seek through the records. When it wants to read a specific row in a table, it can quickly calculate the position of it. If the row size is not fixed, every time it needs to do a seek, it has to consult the primary key index.

They are also easier to cache and easier to reconstruct after a crash. But they also can take more space. For instance, if you convert a VARCHAR(20) field to a CHAR(20) field, it will always take 20 bytes of space regardless of what is it in.

By using "Vertical Partitioning" techniques, you can separate the variable-length columns to a separate table. Which brings us to:

16. Vertical Partitioning

Vertical Partitioning is the act of splitting your table structure in a vertical manner for optimization reasons.

Example 1: You might have a users table that contains home addresses, that do not get read often. You can choose to split your table and store the address info on a separate table. This way your main users table will shrink in size. As you know, smaller tables perform faster.

Example 2: You have a "last_login" field in your table. It updates every time a user logs in to the website. But every update on a table causes the query cache for that table to be flushed. You can put that field into another table to keep updates to your users table to a minimum.

But you also need to make sure you don't constantly need to join these 2 tables after the partitioning or you might actually suffer performance decline.

17. Split the Big DELETE or INSERT Queries

If you need to perform a big DELETE or INSERT query on a live website, you need to be careful not to disturb the web traffic. When a big query like that is performed, it can lock your tables and bring your web application to a halt.

Apache runs many parallel processes/threads. Therefore it works most efficiently when scripts finish executing as soon as possible, so the servers do not experience too many

open connections and processes at once that consume resources, especially the memory.

If you end up locking your tables for any extended period of time (like 30 seconds or more), on a high traffic web site, you will cause a process and query pileup, which might take a long time to clear or even crash your web server.

If you have some kind of maintenance script that needs to delete large numbers of rows, just use the LIMIT clause to do it in smaller batches to avoid this congestion.

```
1 while (1) {
2     mysql_query("DELETE FROM logs WHERE log_date <= '2009-10-01' LIMIT 10000");
3     if (mysql_affected_rows() == 0) {
4         // done deleting
5         break;
6     }
7     // you can even pause a bit
8     usleep(50000);
9 }
```

18. Smaller Columns Are Faster

With database engines, disk is perhaps the most significant bottleneck. Keeping things smaller and more compact is usually helpful in terms of performance, to reduce the amount of disk transfer.

MySQL docs have a list of [Storage Requirements](#) for all data types.

If a table is expected to have very few rows, there is no reason to make the primary key an INT, instead of MEDIUMINT, SMALLINT or even in some cases TINYINT. If you do not need the time component, use DATE instead of DATETIME.

Just make sure you leave reasonable room to grow or you might end up like [Slashdot](#).

19. Choose the Right Storage Engine

The two main storage engines in MySQL are MyISAM and InnoDB. Each have their own pros and cons.

MyISAM is good for read-heavy applications, but it doesn't scale very well when there are a lot of writes. Even if you are updating one field of one row, the whole table gets locked,

and no other process can even read from it until that query is finished. MyISAM is very fast at calculating SELECT COUNT(*) types of queries.

InnoDB tends to be a more complicated storage engine and can be slower than MyISAM for most small applications. But it supports row-based locking, which scales better. It also supports some more advanced features such as transactions.

- [MyISAM Storage Engine](#)
- [InnoDB Storage Engine](#)

20. Use an Object Relational Mapper

By using an **ORM (Object Relational Mapper)**, you can gain certain performance benefits. Everything an ORM can do, can be coded manually too. But this can mean too much extra work and require a high level of expertise.

ORM's are great for "Lazy Loading". It means that they can fetch values only as they are needed. But you need to be careful with them or you can end up creating too many mini-queries that can reduce performance.

ORM's can also batch your queries into transactions, which operate much faster than sending individual queries to the database.

Currently my favorite ORM for PHP is [Doctrine](#). I wrote an article on how to [install Doctrine with CodeIgniter](#).

21. Be Careful with Persistent Connections

Persistent Connections are meant to reduce the overhead of recreating connections to MySQL. When a persistent connection is created, it will stay open even after the script finishes running. Since Apache reuses it's child processes, next time the process runs for a new script, it will reuse the same MySQL connection.

- [mysql_pconnect\(\) in PHP](#)

It sounds great in theory. But from my personal experience (and many others), this features turns out to be not worth the trouble. You can have serious problems with connection limits, memory issues and so on.

Apache runs extremely parallel, and creates many child processes. This is the main reason that persistent connections do not work very well in this environment. Before you consider using the `mysql_pconnect()` function, consult your system admin.