PERCONA

MySQL Performance Tuning

Strategies, best practices, and tips from Percona MySQL experts

Contents

MySQL Performance Tuning

Strategies, best practices, and tips from Percona MySQL experts

Welcome to MySQL Performance Tuning, an essential compilation of the most sought-after MySQL wisdom and insights from the Percona blog. This ebook is not intended to be an exhaustive manual on MySQL performance; it serves as an expert companion piece, providing a curated selection of the greatest hits penned by Percona's MySQL professionals, who have decades of experience solving complex database performance issues. Inside, we will cover the critical aspects of MySQL performance optimization, guided by seasoned experts who share their invaluable experiences and tested solutions.

Each chapter in this comprehensive collection focuses on a unique aspect of MySQL optimization, ranging from query optimization and memory/resource allocation to partitioning and schema design. It is designed to provide you with an array of strategies, tips, and best practices that have been proven to enhance the performance of MySQL databases. Whether you're a database administrator, developer, or system architect, this compilation provides a unique opportunity to benefit from the combined experience of Percona's MySQL experts.

MySQL Performance Tuning will help you make informed decisions that ensure your databases are running at their best. Get ready to elevate your MySQL databases to new levels of efficiency and performance!

Query optimization

Our first chapter is all about query optimization, a key tactic for boosting your MySQL database's efficiency and responsiveness to enhance performance. Drawing upon our experts' extensive knowledge and experience, we explore various approaches to query optimization, each offering a unique perspective on how to streamline and refine database queries.

While this chapter should not be seen as the definitive guide to query optimization, it aims to provide you with some expert practices that can transform your database operations.

Whether you're a database administrator looking to reduce query times, a developer seeking to optimize application performance, or simply an enthusiast eager to learn more about the intricacies of MySQL, it will equip you with the knowledge to elevate your database's functionality.

How to Find and Tune a Slow SQL Query

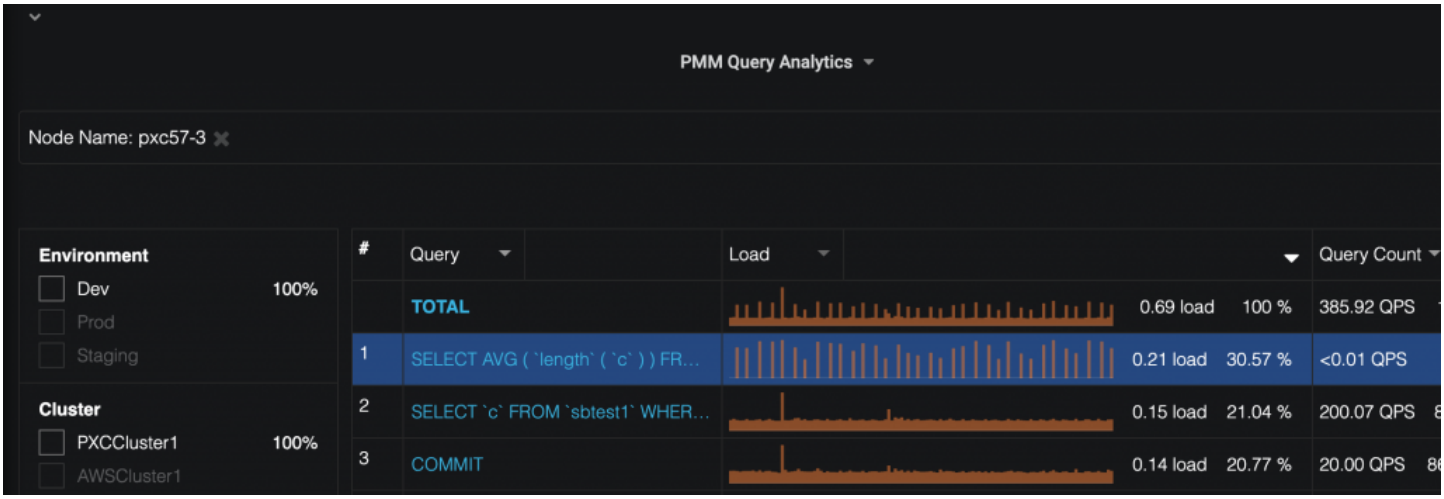
By Mike Benshoof

Michael joined Percona in 2012 as a US-based consultant and is currently a Technical Account Manager. Prior to joining Percona, Michael spent several years in a DevOps role, maintaining a SaaS application specializing in social networking. His experiences include application development and scaling, systems administration, along with database administration and design.

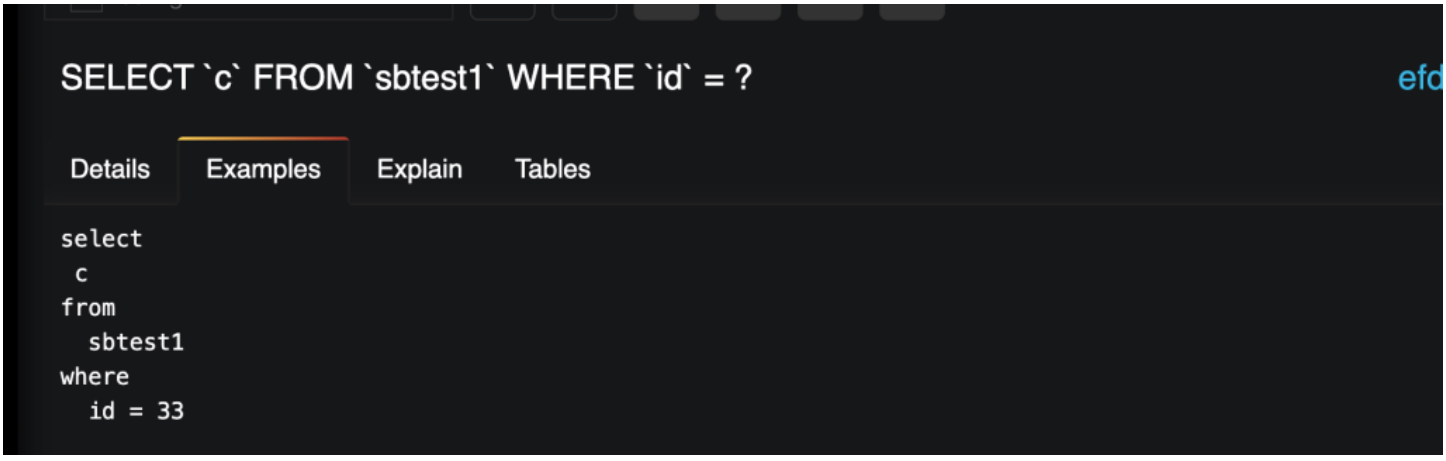
One of the most common support tickets we get at Percona is the infamous “database is running slower” ticket. While this can be caused by a multitude of factors, it is more often than not caused by a bad query. While everyone always hopes to recover through some quick config tuning, the real fix is to identify and fix the problem query. Sure, we can generally alleviate some pain by throwing more resources at the server. But this is almost always a short-term bandaid and not the proper fix.

With Percona Monitoring and Management

So how do we find the queries causing problems and fix them? If you have Percona Monitoring and Management (PMM) installed, the identification process is swift. With Query Analytics enabled (QAN) in PMM, you can simply look at the table to identify the top query:



When you click on the query in the table, you should see some statistics about that query and also (in most cases), an example:



View a Demo of Percona Monitoring and Management

Without Percona Monitoring and Management

Now, let's assume that you don't have PMM installed yet (I'm sure that is being worked on as you read this). To find the problem queries, you'll need to do some manual collection and processing that PMM does for you. The following is the best process for collecting and aggregating the top queries:

1. Set long_query_time = 0 (in some cases, you may need to rate limit to not flood the log)
2. Enable the slow log and collect for some time (slow_query_log = 1)
3. Stop collection and process the log with pt-query-digest
4. Begin reviewing the top queries in times of resource usage

Note - you can also use the performance schema to identify queries, but setting that up is outside the scope here. Here is a good reference on how to use P_S to find suboptimal queries.

When looking for bad queries, one of the top indicators is a large discrepancy between rows_examined and rows_sent. In cases of suboptimal queries, the rows examined will be very large compared with a small number of rows sent.

Once you have identified your query, it is time to start the optimization process. The odds are that the queries at the top of your list (either in PMM or the digest report) lack indices. Indexes allow the optimizer to target the rows you need rather than scanning everything and discarding non-matching values. Let's take the following sample query as an example:

```
SELECT *
FROM user
WHERE username = "admin1"
ORDER BY last_login DESC;
```

This looks like a straightforward query that should be pretty simple. However, it is showing up as a resource hog and is bogging down the server. Here is how it showed up in the pt-query-digest output:

```
# Profile
# Rank Query ID           Response time Calls R/Call V/M   Item
# ==== ================== ============= ===== ====== ===== ===========
#    1 0xA873BB85EEF9B3B9  0.4011 98.7%     2 0.2005  0.40 SELECT user
# MISC 0xMISC              0.0053  1.3%     7 0.0008   0.0 <7 ITEMS>

# Query 1: 0.18 QPS, 0.04x concurrency, ID 0xA873BB85EEF9B3B9 at byte 3391
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.40
# Time range: 2018-08-30T21:38:38 to 2018-08-30T21:38:49
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         22       2
# Exec time     98   401ms    54us   401ms   201ms   401ms   284ms   201ms
# Lock time     21   305us       0   305us   152us   305us   215us   152us
# Rows sent      6       1       0       1    0.50       1    0.71    0.50
# Rows examine  99 624.94k       0 624.94k 312.47k 624.94k 441.90k 312.47k
# Rows affecte   0       0       0       0       0       0       0       0
# Bytes sent    37     449      33     416  224.50     416  270.82  224.50
# Query size    47     142      71      71      71      71       0      71
# String:
# Databases    plive_2017
# Hosts        localhost
# Last errno   0
# Users        root
# Query_time distribution
#   1us
#  10us  ################################################################
# 100us
#   1ms
```

```
#  10ms
# 100ms  ###############################################################
#    1s
#  10s+
# Tables
#    SHOW TABLE STATUS FROM `plive_2017` LIKE 'user'G
#    SHOW CREATE TABLE `plive_2017`.`user`G
# EXPLAIN /*!50100 PARTITIONS*/
SELECT *  FROM user  WHERE username = "admin1" ORDER BY last_login DESCG
```

We can see right away the high number of rows examined vs. the rows sent, as highlighted above. So now that we've identified the problem query let's start optimizing it. Step one in optimizing the query would be to run an EXPLAIN plan:

```
mysql> EXPLAIN SELECT *  FROM user  WHERE username = "admin1" ORDER BY last_login DESCG
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 635310
     filtered: 10.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)
```

The EXPLAIN output is the first clue that this query is not properly indexed. The type: ALL indicates that the entire table is being scanned to find a single record. This will often lead to I/O pressure on the system if your dataset exceeds memory. The Using filesort indicates that once it goes through the entire table to find your rows, it has to then sort them (a common symptom of CPU spikes).

Limiting rows examined

One thing that is critical to understand is that query tuning is an iterative process. You won't always get it right the first time and data access patterns may change over time. In terms of optimization, the first thing we want to do is get this query using an index and not using a full scan. For this, we want to look at the WHERE clause: where username = "admin1".

With this column theoretically being selective, an index on username would be a good start. Let's add the index and re-run the query:

```
mysql> ALTER TABLE user ADD INDEX idx_name (username);
Query OK, 0 rows affected (6.94 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT *  FROM user  WHERE username = "admin1" ORDER BY last_login DESCG
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
   partitions: NULL
         type: ref
possible_keys: idx_name
          key: idx_name
      key_len: 131
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using index condition; Using filesort
1 row in set, 1 warning (0.01 sec)
```

Optimizing sorts

So we are halfway there! The type: ref indicates we are now using an index, and you can see the rows dropped from 635k down to one. This example isn't the best as this finds one row, but the next thing we want to address is the filesort. For this, we'll need to change our username index to be a composite index (multiple columns). The rule of thumb for a composite index is to work your way from the most selective to the least selective columns, and then if you need sorting, keep that as the last field. Given that premise, let's modify the index we just added to include the last_login field:

```
mysql> ALTER TABLE user DROP INDEX idx_name, ADD INDEX idx_name_login (username, last_login);
Query OK, 0 rows affected (7.88 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT *  FROM user  WHERE username = "admin1" ORDER BY last_login DESCG
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
   partitions: NULL
         type: ref
possible_keys: idx_name_login
          key: idx_name_login
      key_len: 131
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

And there we have it! Even if this query scanned more than one row, it would read them in sorted order, so the extra CPU needed for the sorting is eliminated. To show this, let's do this same index on a non-unique column (I left email as non-unique for this demo):

```
mysql> select count(1) from user where email = "SGCRGCTOPGLNGR@RLCDLWD.com";
+----------+
| count(1) |
+----------+
|       64 |
+----------+
1 row in set (0.23 sec)

mysql> ALTER TABLE user ADD INDEX idx_email (email, last_login);
Query OK, 0 rows affected (8.08 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT *  FROM user  WHERE email = "SGCRGCTOPGLNGR@RLCDLWD.com" ORDER BY last_
login DESCG
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
   partitions: NULL
         type: ref
possible_keys: idx_email
          key: idx_email
      key_len: 131
          ref: const
         rows: 64
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

In summary, the general process to tune a SQL query follows this process:

1. Identify the query (either manually or with a tool like PMM)
2. Check the EXPLAIN plan of the query
3. Review the table definition
4. Create indexes
   a. Start with columns in the WHERE clause
   b. For composite indexes, start with the most selective column and work to the least selective column
5. Ensure sorted columns are at the end of the composite index
6. Review the updated explain plan and revise as needed
7. Continue to review the server to identify changes in access patterns that require new indexing

While query optimization can seem daunting, using a process can make it much easier to achieve. Naturally, optimizing complex queries isn't trivial like the above example, but is definitely possible when broken down. And remember that Percona engineers are always available to help you when you get stuck! Happy optimizing!

MySQL Query Performance Troubleshooting: Resource-Based Approach

By Peter Zaitsev

Peter is an entrepreneur and co-founder of Percona. As one of the leading experts in open source strategy and database optimization, Peter has used his technical vision and entrepreneurial skills to grow Percona from a two-person store into one of the most respected open source companies in the business with over 350 employees. He is the co-author of the book "High Performance MySQL: Optimization, Backup and Replication," one of the most popular books on MySQL performance.

When I speak about MySQL performance troubleshooting (or frankly any other database), I tend to speak about four primary resources which typically end up being a bottleneck and limiting system performance: CPU, Memory, Disk, and Network.

It would be great if when seeing what resource is a bottleneck, we could also easily see what queries contribute the most to its usage and optimize or eliminate them. Unfortunately, it is not as easy as it may seem.

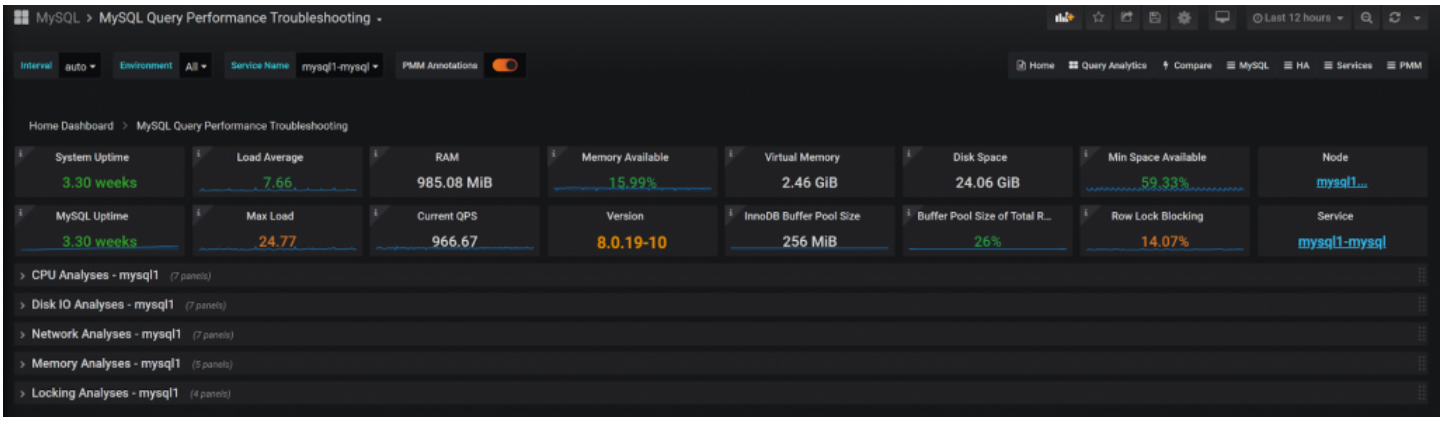
First, MySQL does not really provide very good instrumentation in those terms, and it is not easy to get information on how much CPU usage, Disk IO, or Memory a given query caused. Second, direct attribution is not even possible in a lot of cases. For example, disk writes from flushing data from the InnoDB buffer pool in the background aggregates writes from multiple queries, or disk what technically is Disk IO - temporary files, etc. may, in the end, cause major memory load due to caching.

While exact science on this matter is hard to impossible with MySQL, I think we can get close enough for it to be useful... and this is what I'll try to do as explained in the rest of this article.

This is work in progress so your feedback is very much appreciated!

I have created the dashboard for Percona Monitoring and Management (PMM) v2, which combines the System Performance Metrics with MySQL Performance Metrics and Query Execution information to estimate the top queries for each class.

In addition to the top resources mentioned, I also add locking, which while is not really a "physical" resource but still often a cause of database performance problems.



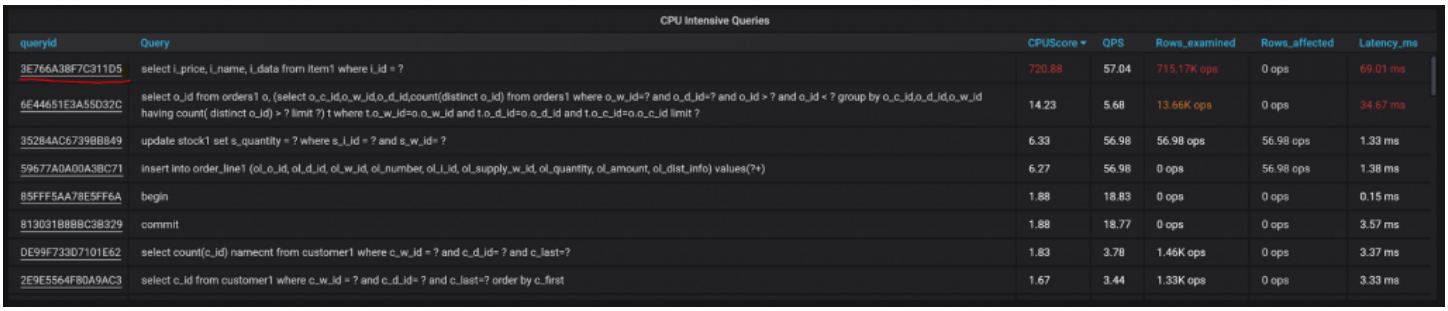
CPU analysis

In the first part, we see the performance graphs; the first line is system metrics which show utilization and saturation (as in Brendan Gregg's USE Method). Additionally, to overall CPU, I also included the information about max core utilization, which is rather important for MySQL where you will often have a bottleneck caused one CPU core saturated with a single thread workload rather than running out of CPU capacity for multi-core CPU.

The second row contains the data which is relevant for high-level CPU usage in MySQL: Connections, Queries, and Handlers. A large number of new connections, especially TLS, can cause significant CPU usage. A large number of queries, even trivial ones, can load up CPU, and finally, there are MySQL "Handlers" which correspond to row-level operations. When there is a lot of data being crunched, CPU usage tends to be high.

The purpose of those graphs is to easily allow you to find a problem spot when resource utilization was high. You can zoom in to it by selecting a given interval on the graph and see query activity for that particular activity.

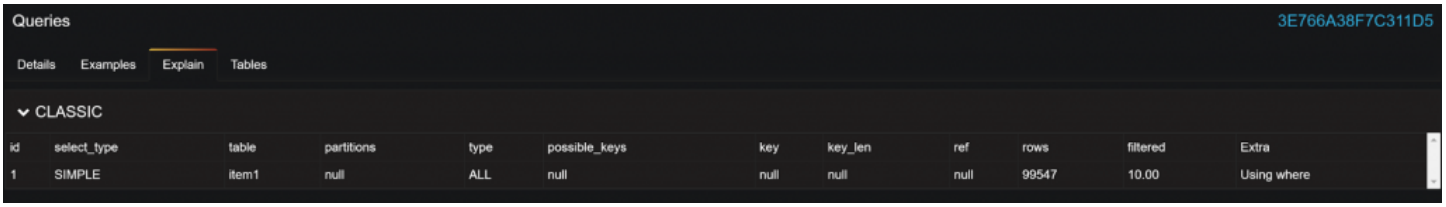
The next section includes the mentioned CPU intensive queries:



You can see queries are sorted by CPU Score - the metric computed based on the number of rows a query examines and changes (there is no direct CPU usage data easily available). We also have additional columns that typically are of interest for CPU intensive queries: number of rows this query kind crunches per second, the number of such queries, query avg latency, etc.

If you click on QueryID, Query Analytics will open with a focus on this particular query and let you see more details:

This particular query examines 12K rows for each row send; not particularly healthy. If we check out EXPLAIN, we can see it is using FULL table scan because index is missing.



Disk IO analysis

For Disk IO, we're also looking at Disk Utilization and Saturation as Disk IO errors would simply cause a database crash.

We're looking at both system-level Disk IO Bandwidth as well as "Load" - how many IO requests, on average, are in flight, as well as Disk IO Latency, which is compared to the medium-term average. If something is wrong with storage, or if it is overloaded, latency tends to spike compared to the norm.

We're also looking at the most important disk consumers on MySQL Level - InnoDB Data IO, Log IO, and Fsyncs. As you pick the period you're looking to analyze, you can look at the queries scored by IOScore.

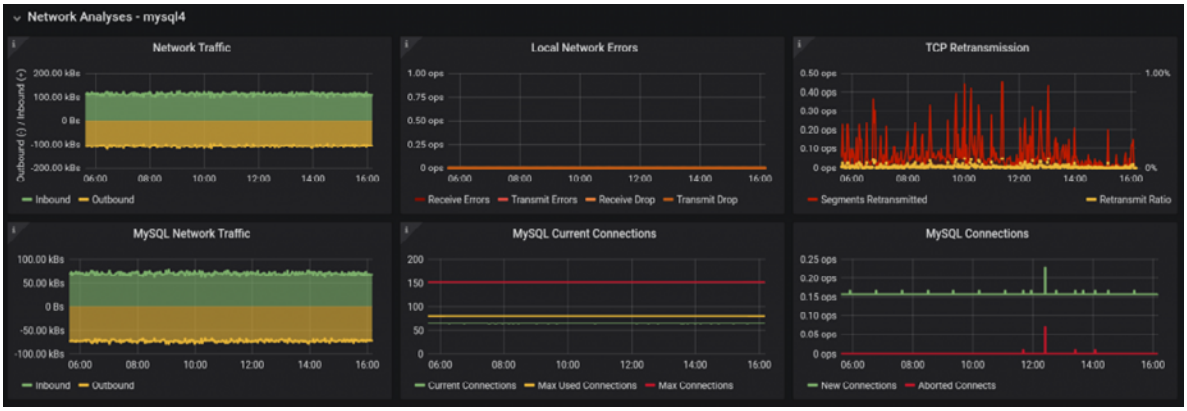
IO Intensive Queries

queryid	Query	IOScore	QPS	InnoDB_Reads	Read_IO_Load	Read_Wait_pct	Latency_ms
3E766A38F7C311D5	select i_price, i_name, i_data from item1 where i_id = ?	4.27 K	57.59 ops	348.42 kBs	0.04	0.90%	68.95 ms
38D881F83C541478	select count(distinct (s_i_id)) from order_line1, stock1 where ol_w_id = ? and ol_d_id = ? and ol_o_id < ? and ol_o_id >= ? and s_w_id= ? and s_i_id=ol_i_id and s_quantity < ?	3.93 K	0.58 ops	569.05 kBs	0.03	77.89%	61.55 ms
455FB77DFC41ADE1	delete from order_line1 where ol_w_id=? and ol_d_id=? and ol_o_id=?	1.50 K	5.74 ops	95.96 kBs	0.01	36.41%	6.26 ms
E973F56CA3B8E4F7	update customer1 set c_balance = c_balance + ?, c_delivery_cnt = c_delivery_cnt + ? where c_id = ? and c_d_id = ? and c_w_id = ?	1.21 K	5.86 ops	72.56 kBs	0.01	49.95%	3.65 ms
6BB7CACF1F236F48	select s_quantity, s_data, s_dist_10 s_dist from stock1 where s_i_id = ? and s_w_id= ? for update	746.58	5.86 ops	58.30 kBs	0.01	50.06%	2.15 ms
9CF7C5A18C94BC03	select s_quantity s_data s_dist_08 s_dist from stock1 where s_i_id = ? and s_w_id= ? for update	745.41	5.80	57.67 kBs	0.01	50.09%	2.17 ms

You can also see the columns most relevant for IO intensive queries - how many reads from the disk query are required, how much load is generated, as well what portion of response time was waiting for disk IO versus total query execution time.

The IO Score (same as all scores) is not super scientific; I just came up with something which looked like it works reasonably well, but the great thing is you can easily customize the scoring to be more meaningful for your environment:

Network analysis



You now get the drill; we see the network usage on the system, the most common errors which can cause the problem, as well as MySQL network-level data traffic, number of concurrent connections, and new connections created (or failed connection attempts).

And queries with network-specific information:

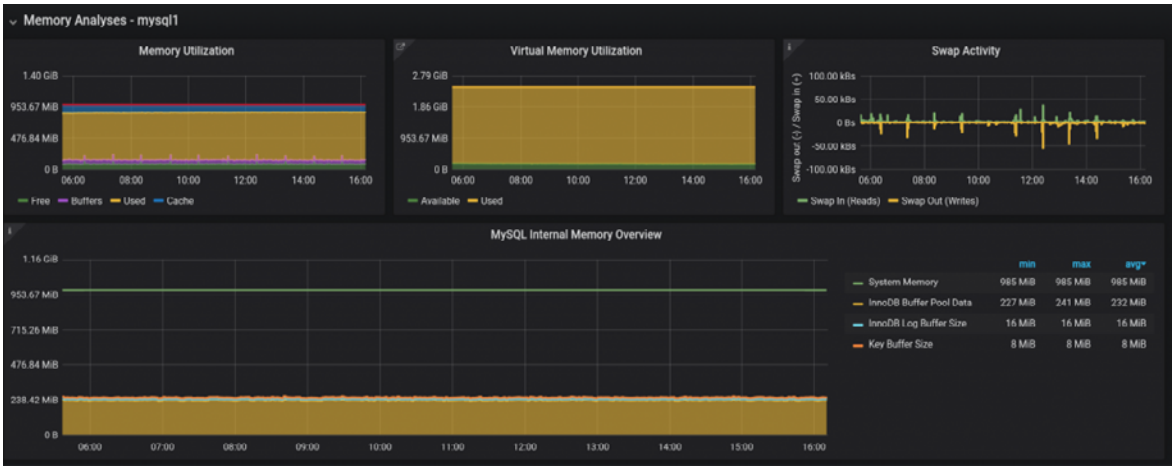
Network Intensive Queries

queryid	Query	NetScore	QPS	Bytes_Sent	Sent_Per_Query	Latency_ms
3E766A38F7C311D5	select i_price, i_name, i_data from item1 where i_id = ?	73.39	57.67 ops	14.34 kBs	248.64 B	68.93 ms
35284AC6739BB849	update stock1 set s_quantity = ? where s_i_id = ? and s_w_id= ?	61.99	57.61 ops	3.00 kBs	52.00 B	1.32 ms
59677A0A00A3BC71	insert into order_line1 (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) values(?+)	59.63	57.61 ops	633.73 Bs	11.00 B	1.37 ms
85FFF5AA78E5FF6A	begin	19.69	19.02 ops	209.23 Bs	11.00 B	0.15 ms
813031B8B8C3B329	commit	19.63	18.96 ops	208.61 Bs	11.00 B	3.55 ms
EA13E5DEEF9A840C	select c_first, c_middle, c_last, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim, c_discount,	12.69	5.79	6.76 kBs	1.14 KiB	1.48 ms

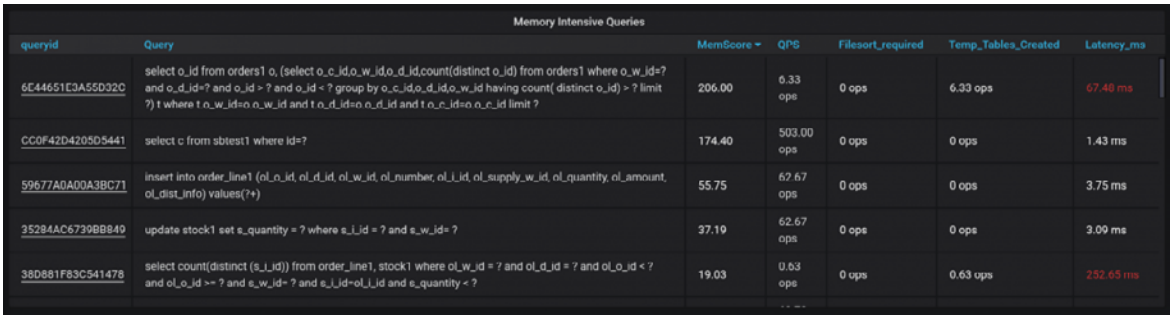
This query generates network bandwidth utilization, how large the average query result set is, etc.

Memory analysis

At the system level, physical memory and virtual memory are resources you need to be tracking as well as swap activity, as serious swapping is a performance killer. For MySQL, I currently only show the most important memory buffers as configured; it would be much better to integrate with memory instrumentation in MySQL 5.7 performance schema, which I have not done yet…



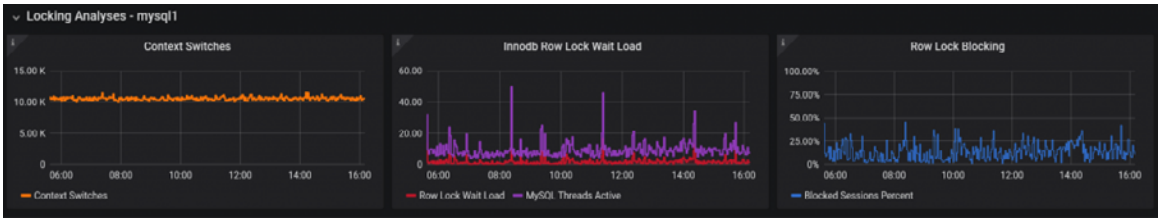
And when we can get queries showing which are heavy memory users:



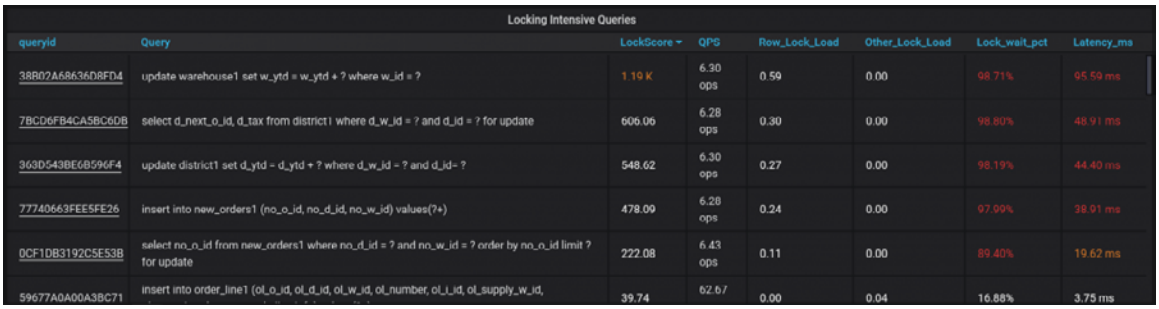
I'm using indirect measures here such as the use of temporary tables and filesort to guesstimate memory-intensive queries.

Locking analysis

Locking is a database phenomenon so it does not show on the system level in some very particular way. It may show itself as an increased number of context switches, which is why I include the graph here. The other two graphs show us the load (or the number of average active sessions which were blocked waiting for row-level locks), as well as what portion of the total running session it is, basically to show how much of overall response time is due to locks rather than Disk IO, CPU, etc.



And here we have the queries:



When it comes to locking in MySQL, you typically have to take care of two kinds of locks. There are InnoDB row locks, and there are other locks - tables locks, DML locks, etc., which I show here as row locks and other locks, highlighting how much load each of these generates.

We can also see what percent of the particular query response time was taken by waiting on lock, which can give a lot of ideas for optimization.

Note: Queries that spent a lot of time waiting on locks may be blocked by other instances of the same queries... or might be blocked by entirely different types of queries, which is much harder to catch. Still, this view will be helpful to troubleshoot many simple locking cases.

Using Slow Query Log to Find High Load Spots in MySQL

by Uday Varagani

Uday worked at Percona as Technical Support Specialist, assisting customers in troubleshooting MySQL database issues.

pt-query-digest is one of the most commonly used tools for query auditing in MySQL. By default, pt-query-digest reports the top ten queries consuming the most amount of time inside MySQL. A query that takes more time than the set threshold for completion is considered slow, but it's not always true that tuning such queries makes them faster. Sometimes, when resources on the server are busy, it will impact every other operation on the server, and so will impact queries too. In such cases, you will see the proportion of slow queries going up. That can also include queries that work fine in general.

This article explains a small trick to identify such spots using pt-query-digest and the slow query log. pt-query-digest is a component of Percona Toolkit, open source software that is free to download and use.

Overview of the MySQL slow query log

The MySQL slow query log is a feature of MySQL that logs information about queries that take longer than a specified threshold to execute. It is a useful tool for identifying performance issues in MySQL databases.

When enabled, the slow query log records information about each slow query, including query text, execution time, and query timestamp. This information can be used to analyze and optimize database performance.

Once enabled, you can analyze the log file using various tools, such as the MySQL command-line tool, Percona Toolkit, or Percona Monitoring and Management (PMM).

How to enable the slow query log in MySQL

1. Open your MySQL configuration file.
2. Find the [mysqld] section in the configuration file.
3. Add the following lines to the [mysqld] section:

```
slow_query_log = 1
slow_query_log_file = /path/to/your/log/file.log
long_query_time = 5
```

Here, "slow_query_log" enables the slow query log, "slow_query_log_file" specifies the location and name of the file, and "long_query_time" specifies the number of seconds (example, you can enter your own number) that a query must take to be considered "slow" and logged.

4. Save the configuration file and restart the MySQL server.

After enabling the slow query log, MySQL will log all queries that take longer than the "long_query_time" value you entered.

To verify that the slow query log is working correctly, run a slow query to test if the slow query log is working. If the query takes longer than the specified time (in this example, five seconds), check the slow query log file to verify that it contains the slow query.

Slow query log parameters

To enable the MySQL slow query log, you need to modify the MySQL server configuration file (usually my.cnf or my.ini) and set the following parameters:

- `slow_query_log`: This specifies whether the slow query log is enabled or disabled. The default value is OFF. To enable the slow query log, set it to ON.
- `slow_query_log_file`: This parameter specifies the name and location of the file where slow queries will be logged. The default value is hostname-slow.log and it is stored in the data directory of the MySQL server.
- `long_query_time`: This specifies the amount of time a query must take to be considered "slow" and logged in the slow query log. The default value is 10 seconds. You can adjust this value.
- `log_queries_not_using_indexes`: This parameter specifies whether to log queries that are not using indexes. If you want to log queries that are not using indexes, set it to ON as the default value is OFF.
- `log_slow_admin_statements`: This parameter specifies whether to log administrative statements that take a long time to execute. The default value is OFF. If you want to log slow administrative statements, set it to ON.

Slow query log contents

The slow query log provides information that helps identify queries that are taking too long and can be optimized. The types of slow query log statements include the following:

- **Query timestamp:** Date and time the query was executed.
- **Query execution time:** The time taken to execute the query.
- **Query ID:** A unique identifier assigned to each query to match queries with corresponding log entries.
- **Query user:** The user name of the person who executed the query.
- **Query database:** The database name on which the query was executed.
- **Query text:** The SQL text of the executed query.
- **Query result:** How many rows were returned by the query.
- **Query error message:** If the query fails to execute, this is the error message associated with it.

How to read the MySQL slow query log

By analyzing the MySQL slow query log, you can identify queries that are taking too long to execute and causing performance issues. In order to read the slow query log, follow these steps:

- Locate and open the slow query log file, usually located in the MySQL data directory.
- Each log entry coincides with a query that exceeded the threshold originally set in the MySQL configuration. The entry contains such information as the query timestamp, query execution time, and the user who executed the query. This information can then be used to identify the specific queries causing performance issues.
- After identifying the slow queries, you can optimize them by adding indexes, rewriting queries, or tuning the MySQL configuration. Using tools such as “explain” can help in analyzing the execution plan of the queries to identify areas that require optimization.

Some sample data

Let’s have a look at sample data in Percona Server for MySQL 5.7. The slow query log is configured to capture queries longer than ten seconds with no limit on the logging rate, which is generally considered for throttling the IO that comes while writing slow queries to the log file.

```
mysql> show variables like ‘log_slow_rate%’ ;
+---------------------+---------+
| Variable_name       | Value    |
+---------------------+---------+
| log_slow_rate_limit | 1       |  --> Log all queries
| log_slow_rate_type  | session |
+---------------------+---------+
2 rows in set (0.00 sec)

mysql> show variables like ‘long_query_time’ ;
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 |  --> 10 seconds
```

When I run pt-query-digest, I see in the summary report that 80% of the queries have come from just three query patterns.

```
1  # Profile
2  # Rank Query ID                           Response time    Calls R/Call   V/M
3  # ==== ================================== ================ ===== ======== =====
4  #    1 0x7B92A64478A4499516F46891... 13446.3083 56.1%   102 131.8266  3.83 SELECT performance_schema.
events_statements_history
5  #    2 0x752E6264A9E73B741D3DC04F...  4185.0857 17.5%    30 139.5029  0.00 SELECT table1
6  #    3 0xAFB5110D2C576F3700EE3F7B...  1688.7549  7.0%    13 129.9042  8.20 SELECT table2
7  #    4 0x6CE1C4E763245AF56911E983...  1401.7309  5.8%    12 116.8109 13.45 SELECT table4
8  #    5 0x85325FDF75CD6F1C91DFBB85...   989.5446  4.1%    15  65.9696 55.42 SELECT tbl1 tbl2 tbl3     tbl4
9  #    6 0xB30E9CB844F2F14648B182D0...   420.2127  1.8%    4 105.0532 12.91 SELECT tbl5
10 #    7 0x7F7C6EE1D23493B5D6234382...   382.1407  1.6%    12  31.8451 70.36 INSERT UPDATE tbl6
11 #    8 0xBC1EE70ABAE1D17CD8F177D7...   320.5010  1.3%    6  53.4168 67.01 REPLACE tbl7
12 #   10 0xA2A385D3A76D492144DD219B...   183.9891  0.8%    18  10.2216  0.00 UPDATE tbl8
13 #      MISC 0xMISC                      948.6902  4.0%    14  67.7636   0.0 <10 ITEMS>
```

Query #1 is generated by the qan-agent from PMM and runs approximately once a minute. These results will be handed over to PMM Server. Similarly, queries #2 & #3 are pretty simple. I mean, they scan just one row and will return either zero or one row. They also use indexing, which makes me think that this is not because of something just within MySQL. I wanted to know if I could find any common aspect of all these occurrences.

Let's take a closer look at the queries recorded in the slow query log.

```
# grep -B3 DIGEST mysql-slow_Oct2nd_4th.log
....
....
# User@Host: ztrend[ztrend] @ localhost []  Id: 6431601021
# Query_time: 139.279651  Lock_time: 64.502959 Rows_sent: 0  Rows_examined: 0
SET timestamp=1538524947;
SELECT DIGEST, CURRENT_SCHEMA, SQL_TEXT FROM performance_schema.events_statements_history;
# User@Host: ztrend[ztrend] @ localhost []  Id: 6431601029
# Query_time: 139.282594  Lock_time: 83.140413 Rows_sent: 0  Rows_examined: 0
SET timestamp=1538524947;
SELECT DIGEST, CURRENT_SCHEMA, SQL_TEXT FROM performance_schema.events_statements_history;
# User@Host: ztrend[ztrend] @ localhost []  Id: 6431601031
# Query_time: 139.314228  Lock_time: 96.679563 Rows_sent: 0  Rows_examined: 0
SET timestamp=1538524947;
SELECT DIGEST, CURRENT_SCHEMA, SQL_TEXT FROM performance_schema.events_statements_history;
....
....
```

Now you can see two things.

- All of them have the same Unix timestamp.
- They all spent over 70% of their execution time waiting for some lock.

Analyzing MySQL slow query log from pt-query-digest

Now I want to check if I can group the count of queries based on their time of execution. If there are multiple queries at a given time captured into the slow query log, time will be printed for the first query but not all. Fortunately, in this case, I can rely on the Unix timestamp to compute the counts. The timestamp gets captured for every query. Luckily, without a long struggle, a combination of grep and awk utilities displayed what I wanted to display.

```
# grep -A1 Query_time mysql-slow_Oct2nd_4th.log | grep SET | awk -F "=" '{ print $2 }' | uniq -c
2   1538450797;
1   1538524822;
3   1538524846;
7   1538524857;
167 1538524947;   ---> 72% of queries have happened at this timestamp.
1   1538551813;
3   1538551815;
6   1538602215;
1   1538617599;
33  1538631015;
1   1538631016;
1   1538631017;
```

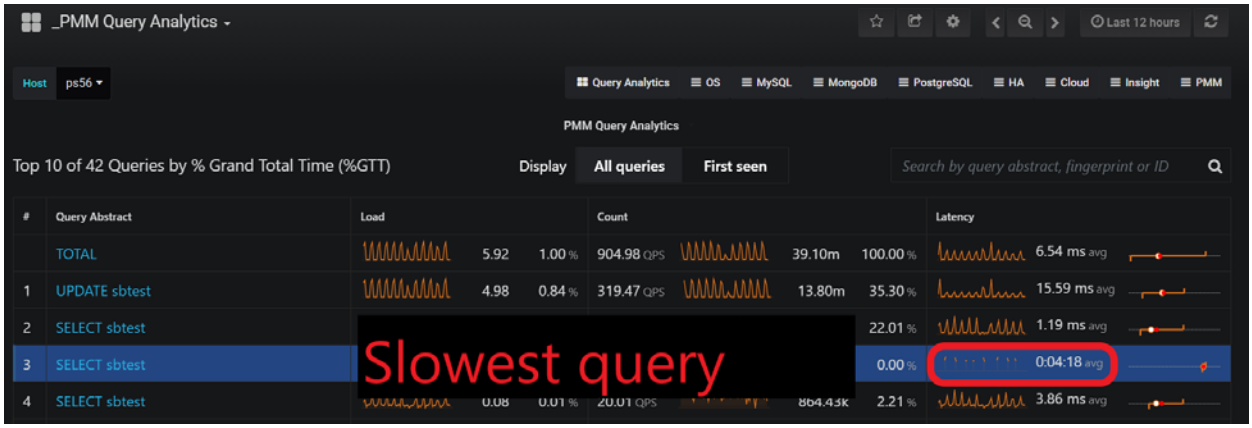
You can use the command below to check the regular date time format of a given timestamp. So, Oct 3, 05:32 is when there was something wrong on the server:

```
# date -d @1538524947
Wed Oct 3 05:32:27 IST 2018
```

Query tuning can be carried out alongside this, but identifying such spots helps avoid spending time on query tuning where badly written queries are not the problem. Having said that, from this point, further troubleshooting may take different sub-paths such as checking log files at that particular time, looking at CPU reports, reviewing past pt-stalk reports if set up to run in the background, and dmesg, etc. This approach is useful for identifying at what time (or time range) MySQL was more stressed just using a slow query log when no robust monitoring tools, like Percona Monitoring and Management (PMM), are deployed.

Using PMM to monitor queries

If you have PMM, you can review Query Analytics to see the topmost slow queries, along with details like execution counts, load, etc. Below is a sample screen copy for your reference:



NOTE: If you use Percona Server for MySQL, a slow query log can report time in microseconds. It also supports extended logging of other statistics about query execution. These provide extra power to see the insights of query processing. You can see more information about these options here.

MySQL Key Performance Indicators (KPI) with PMM

By Kedar Vaijanapurkar

Kedar has been a Tier 2 MySQL DBA at Percona since October 2021 and has a solid foundation in MySQL technologies. With a passion for excellence and continuous improvement, he's committed to honing his skills to deliver unparalleled database management solutions.

As a MySQL database administrator, keeping a close eye on the performance of your MySQL server is crucial to ensure optimal database operations. A monitoring tool like Percona Monitoring and Management (PMM) is a popular choice among open source options for effectively monitoring MySQL performance.

However, simply deploying a monitoring tool is not enough; you need to know which Key Performance Indicators (KPIs) to monitor to gain insights into your MySQL server's health and performance.

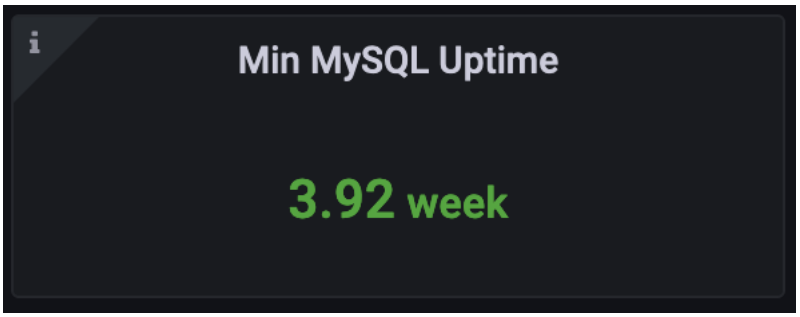
In this article, we will explore various MySQL KPIs that are basic and essential to track using monitoring tools like PMM. We will also discuss related configuration variables to consider that can impact these KPIs, helping you gain a comprehensive understanding of your MySQL server's performance and efficiency.

Let's dive in and learn how (and what) to effectively monitor MySQL performance, along with examples from PMM, by understanding the critical KPIs to watch for.

Database uptime and availability

Monitoring database uptime and availability is crucial as it directly impacts the availability of critical data and the performance of applications or websites that rely on the MySQL database.

PMM monitors the MySQL uptime:




```
show global
status like
'uptime';
```

Indicates the amount of time (seconds) the MySQL server has been running since the last restart.

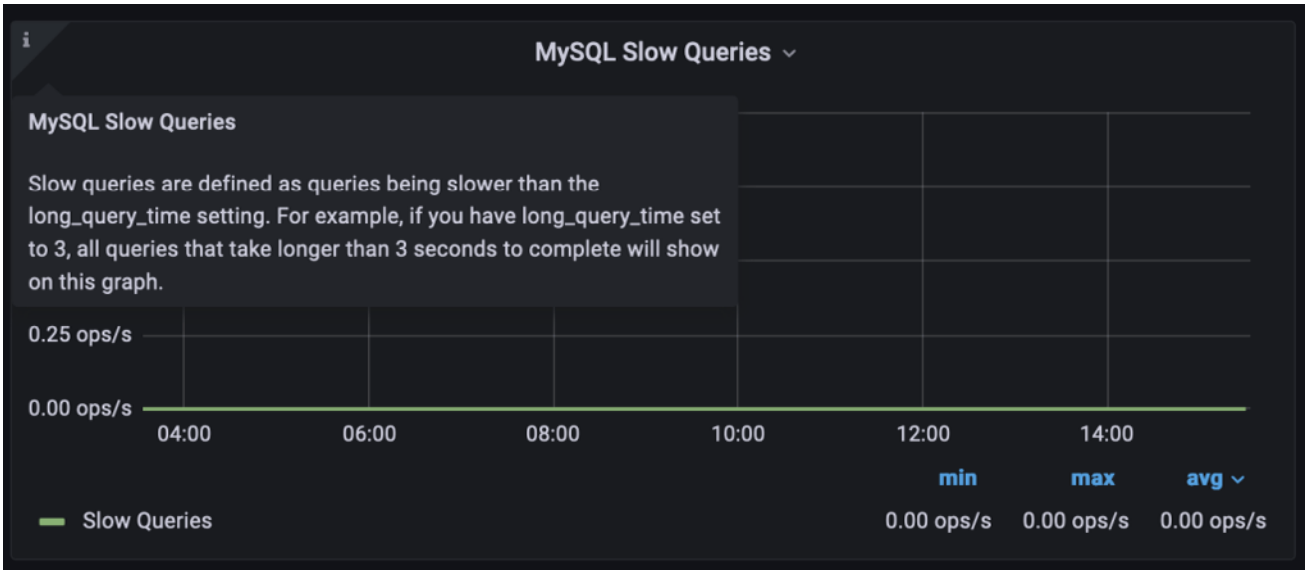
Query performance

Query performance is a key performance indicator (KPI) in MySQL, as it measures the efficiency and speed of query execution. This includes metrics such as query execution time, the number of queries executed per second, and the utilization of query cache and adaptive hash index.

Too many slow queries, inefficient queries, or long-running queries can indicate potential performance issues that may negatively impact the database's performance and why monitoring query performance is crucial.

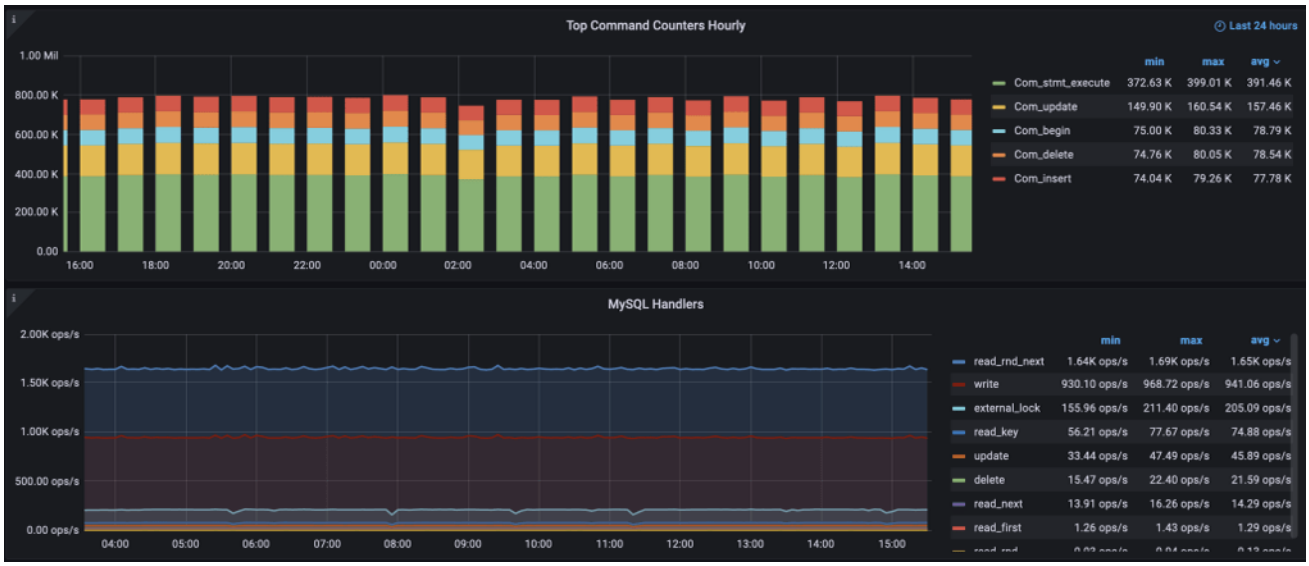
On PMM, we have the following panels showing the gist of query execution and summarizing the pattern. This is not an exhaustive list but an example of what we can watch for.

Number of slow queries recorded



Select types, sorts, locks, and total questions against a database

Command counters and handlers used by queries give an overall traffic summary



Along with this, PMM also comes with Query Analytics giving much detailed information about queries getting executed.

Query Analytics

You should watch out for status variables for these, for example:

```
show global status
like '%sort%';
show global status
like '%slow%';
```

Some of the configuration variables to note for:

- slow_query_log: Enables / Disables slow query log.
- long_query_time: Defines the threshold for query execution time, and queries taking longer than this threshold are considered slow and logged for further analysis
- innodb_buffer_pool_size: Sets the size of the InnoDB buffer pool.
- query cache: Disable (query_cache_size: 0, query_cache_type:OFF)
- innodb_adaptive_hash_index: Check adaptive hash index usage to determine its efficiency.

Indexing efficiency

Monitoring indexing efficiency in MySQL involves analyzing query performance, using EXPLAIN statements, utilizing performance monitoring tools, reviewing error logs, performing regular index maintenance, and benchmarking/testing. This KPI is also directly related to Query Performance and helps improve it.

There are multiple tables MySQL internal system manages that come in handy, identifying the inefficient indexes, to name a few: sys.schema_unused_indexes, and information_schema.index_statistics.

The pt-duplicate-key-checker is another Percona Toolkit utility to eliminate duplicate indexes and improve indexing efficiency.

Connection usage

Connection usage is a critical key performance indicator (KPI) in MySQL, as it involves tracking the number of concurrent connections to the MySQL server and ensuring that it does not exceed the allowed limit. Improper configuration of connection settings can have catastrophic effects on the production database, resulting in it becoming inaccessible during connection spikes or even experiencing out-of-memory (OoM) kills of the MySQL daemon.

By monitoring and managing connection usage, you can proactively identify and address potential issues

such as connection spikes, resource exhaustion, or improper configuration that may impact the availability and performance of the MySQL database.

PMM captures the MySQL connection matrix

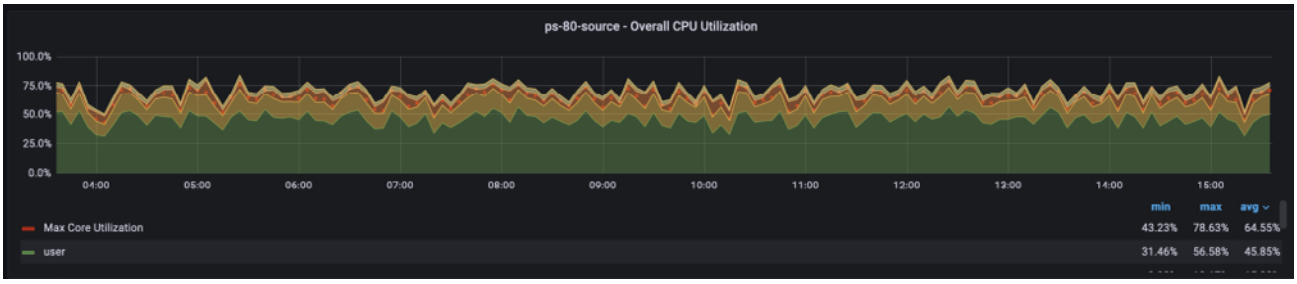
It is important to provide appropriate max_connections and also monitor max_used_connections, max_used_connections_time to review the history of max usage to estimate the traffic.

Implementing appropriate connection pooling or choosing appropriate connection settings can help optimize resource utilization and reduce downtime or performance degradation. Hint Hint, ProxySQL helps.

CPU and memory usage

Monitor CPU and memory utilization of the MySQL server to ensure efficient resource utilization. It is advisable to have a dedicated production MySQL Server that can independently claim the system resources as needed. That said, it should also be monitored for usage, which will exhibit the traffic pressuring them.

PMM dashboard – CPU utilization and memory details



The most effective memory configuration variable, innodb_buffer_pool_size, sets the size of the InnoDB buffer pool.

Another related variable, innodb_buffer_pool_instances, determines the number of buffer pool instances for the InnoDB storage engine, which can improve the performance of multi-core systems by reducing contention on the buffer pool latch.

That said, CPU or memory usage is not only limited to these two variable configurations, and further analysis is required to track what's causing the usage spikes. The point we're making here is they are critical for MySQL Performance.

Disk space usage

Monitor the disk space usage of MySQL data files, log files, and temporary files.
You may review the fragmented tables, binary-logs, and duplicate or redundant indexes to reclaim the space. As a best practice, It is advisable to have different mounts for MySQL data and log files with specific system configurations.

PMM – Disk details, which includes disk usage as well as disk performance charts

Replication lag

Monitoring replication lag is important as it can affect the consistency and reliability of data across multiple database instances in a replication setup.

Replication lag can occur due to various factors such as network latency, system resource limitations, complex transactions, or heavy write loads on the primary/master database. If replication lag is too high, it can result in stale or outdated data on the replica/slave databases, leading to data inconsistencies and potential application issues.

PMM – MySQL Replication Summary dashboard

Related configuration or status variables to consider:

- Seconds_Behind_Master: In SHOW REPLICA STATUS command, this value indicates the replication lag in seconds.

One of the possible improvements in lag would be utilizing parallel replication.

Backup and recovery metrics

Backup and recovery metrics are key performance indicators (KPIs) for MySQL that provide insights into the reliability, efficiency, and effectiveness of backup and recovery processes. They include backup success rate, backup duration, recovery time objective (RTO), recovery point objective (RPO), and backup storage utilization. Monitoring these metrics helps ensure data protection, minimize downtime, and ensure business continuity.

You should not only monitor the backup mount for disk space and backup log but also regularly test the restores and log to match RPO and RTO objectives.

Error rates

The MySQL error log contains information about errors, warnings, and other issues that occur within the MySQL database. By monitoring the error log, you can quickly identify and resolve any problems that may arise, such as incorrect queries, missing or corrupt data, or database server configuration issues.

- error_log: Specifies the location of the MySQL error log.

Conclusion

By monitoring these KPIs, such as database uptime and availability, query performance, indexing efficiency, connection usage, CPU and memory usage, disk space usage, replication lag, backup and recovery metrics, and error rates, you can gain valuable insights into your MySQL server's health and performance.

Note that the specific configuration variables and their optimal values may vary depending on the MySQL version, system hardware, workload, and other factors. It's important to thoroughly understand the MySQL configuration variables and their impacts before making any changes and to carefully monitor the effects of configuration changes to ensure they improve the desired KPIs.

How Percona Monitoring and Management Helps You Find Out Why Your MySQL Server Is Stalling

By Sveta Smirnova

Sveta joined Percona in 2015. Her main professional interests are problem solving, working with tricky issues, bugs, finding patterns that can solve typical issues quicker and teaching others how to deal with MySQL issues, bugs and gotchas effectively. Before joining Percona, Sveta worked as a Support Engineer in the MySQL Bugs Analysis Support Group at MySQL AB-Sun-Oracle.

She is the author of the book "MySQL Troubleshooting" and JSON UDF functions for MySQL.

In this article, I will demonstrate how to use Percona Monitoring and Management to find out the reason why the MySQL server is stalling. I will use only one typical situation for the MySQL server stall in this example, but the same dashboards, graphs, and principles will help you in all other cases.

Nobody wants it but database servers may stop handling connections at some point. As a result, the application will slow down and then will stop responding.

It is always better to know about the stall from a monitoring instrument rather than from your own customers.

PMM is a great help in this case. If you look at its graphs and notice that many of them started showing unusual behavior, you need to react. In the case of stalls, you will see that either some activity went to 0 or, otherwise, it increased to high numbers. In both cases, it does not change.

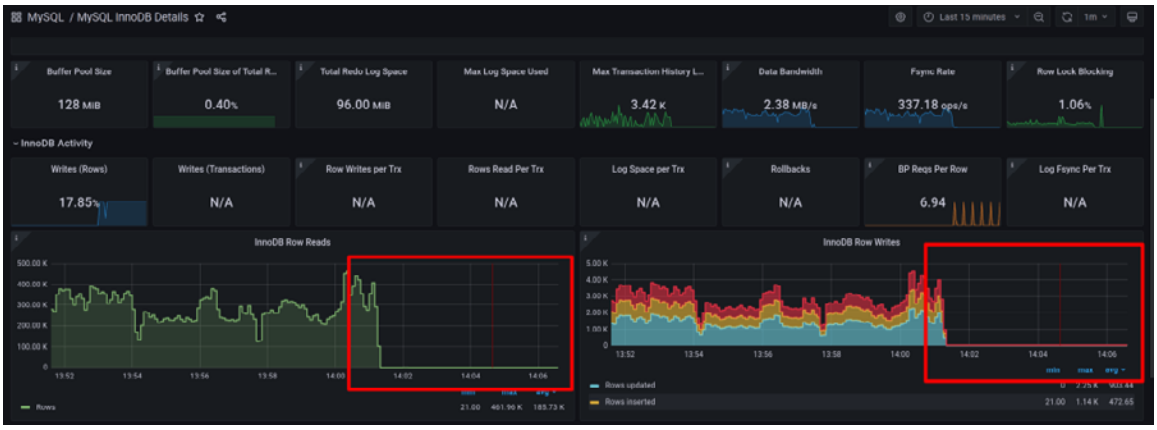
Let's review the dashboard "MySQL Instance Summary" and its graph "MySQL Client Thread Activity" during normal operation:

As you see, the number of active threads fluctuates and this is normal for any healthy application: even if all connections request data, MySQL puts some threads into idle states while they need to wait while the storage engine prepares the data for them. Or, if the client application processes retrieved data.

The next screenshot was taken when the server was stalling:

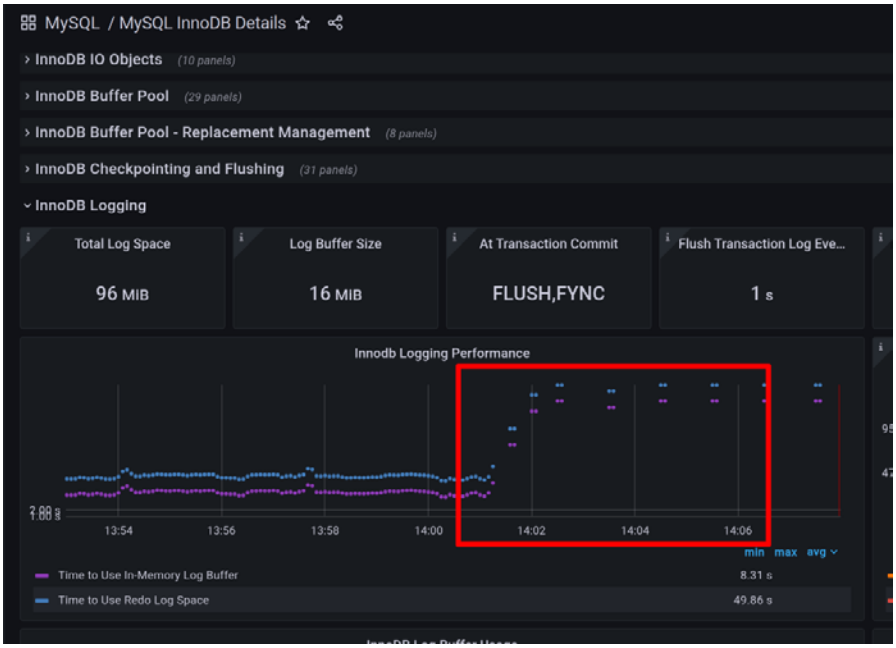
In this picture, you see that the number of active threads is near maximum. At the same time the number of "MySQL Temporary Objects" lowered down to zero. This by itself shows that something unusual happened. But to understand the picture better let's examine storage engine graphs.

I, like most MySQL users, used InnoDB for this example. Therefore the next step for figuring out what is going on would be to examine graphs in the "MySQL InnoDB Details" dashboard.

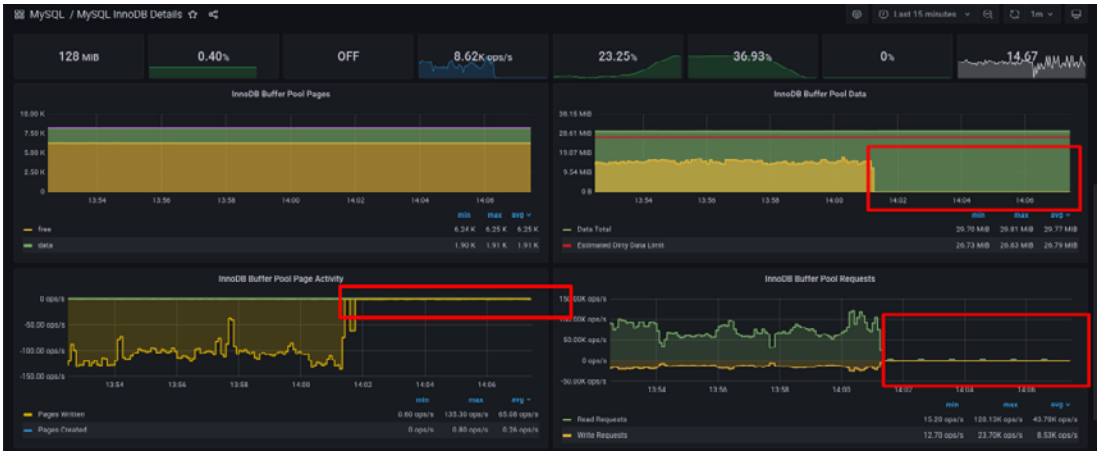


First, we are seeing that the number of rows that InnoDB reads per second went down to zero, as well as the number of rows written. This means that something prevents InnoDB from performing its operations.

More importantly, we see that all I/O operations were stopped. This is unusual even on a server that does not handle any user connection: InnoDB always performs background operations and is never completely idle.

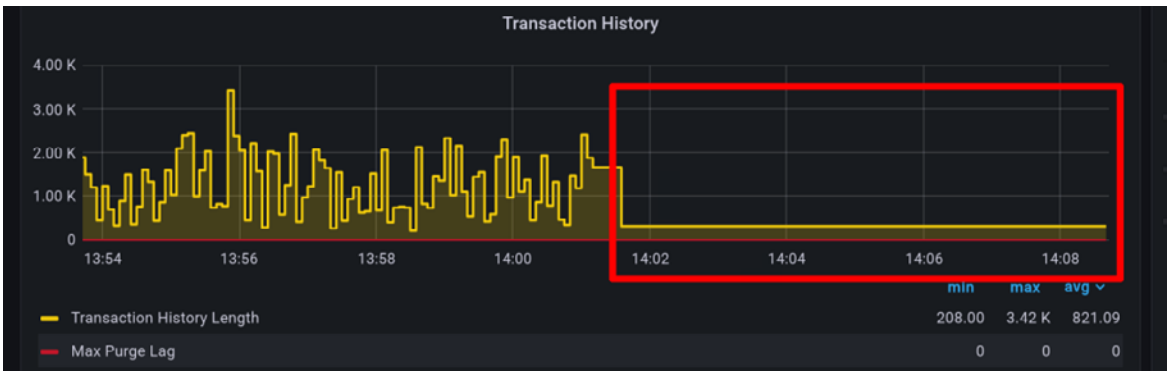


You may see this in the "InnoDB Logging Performance" graph: InnoDB still uses log files but only for background operations.



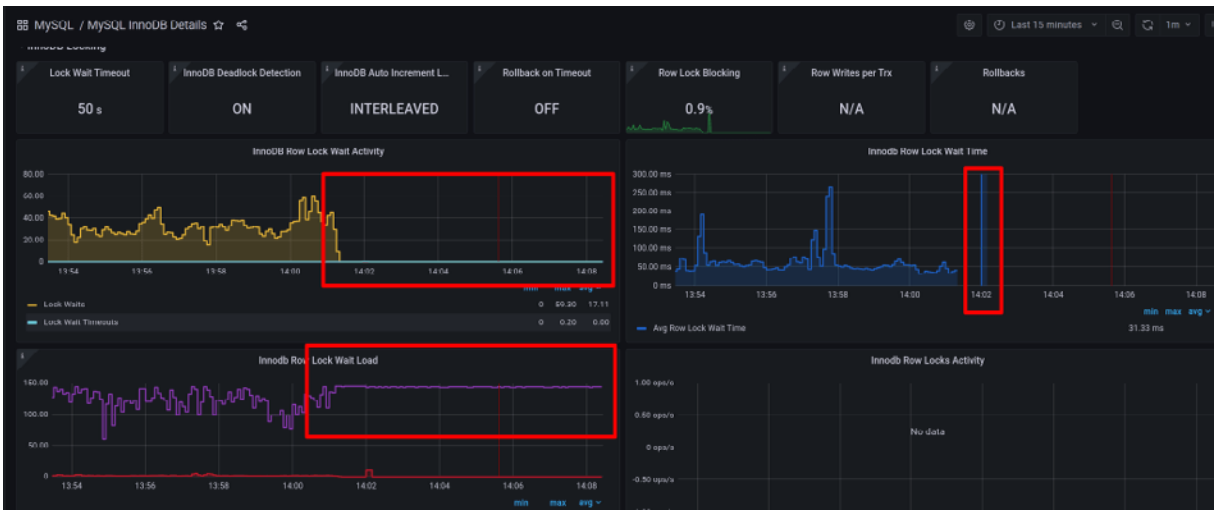
InnoDB Buffer Pool activity is also stopped. What is interesting here is that the number of dirty pages went down to zero. This is visible on the "InnoDB Buffer Pool Data" graph: dirty pages are colored in yellow. This actually shows that InnoDB was able to flush all dirty pages from the buffer pool when InnoDB stopped processing user queries.

At this point we can make the first conclusion that our stall was caused by some external lock, preventing MySQL and InnoDB from handling user requests.



The *"Transaction History"* graph confirms this guess: there are no new transactions and InnoDB was able to flush all transactions that were waiting in the queue before the stall happened.

We can conclude that we are NOT experiencing hardware issues.



This group shows why we experience the stall. As you can see in the "InnoDB Row Lock Wait Time" graph, the wait time was raised to its maximum value around 14:02, then lowered to zero. There is no row lock waits registered during the stall time.

This means that at some point all InnoDB transactions were waiting for a row lock, then failed with a timeout. Still, they have to wait for something. Since there are no hardware issues and InnoDB functions healthy in the background, this means that all threads are waiting for a global MDL lock, created by the server.

If we have **Query Analytics (QAN)** enabled we can find such a command easily.

For the selected time frame we can see that many queries were running until a certain time when a query with id 2 was issued, then other queries stopped running and restarted a few minutes later. The query with id 2 is FLUSH TABLES WITH READ LOCK which prevents any write activity once the tables are flushed.

This is the command that caused a full server stall.

Once we know the reason for the stall we can perform actions to prevent similar issues in the future.

Conclusion

PMM is a great tool that helps not only to identify if your database server is stalling but also to figure out what was the reason for the stall. I used only one scenario here. But you may use the same dashboards and graphs to find out other reasons for the stall, such as DoS attack, hardware failure, a high number of IO operations, caused by poorly optimized queries, and many more.

Proper configuration

This next chapter turns our focus to proper configuration, essential for ensuring the optimal performance of your MySQL database. Our experts discuss how to go about validating configuration settings and configuring your OS to match the specific needs of your application and hardware so you can significantly reduce bottlenecks, prevent downtime, and secure data against threats.

Well-configured databases are more resilient to high traffic volumes and complex queries, ensuring that your applications remain responsive and stable under varying loads. Investing the necessary time in configuring your MySQL database correctly is required for optimal performance and long-term success in your database management strategy.

Easily Validate Configuration Settings in MySQL 8

By Brian Sumpter

Brian joined Percona in 2016 as a Technical Account Manager after a successful tenure in the corporate enterprise sector. His professional background encompasses experience in managing sizable deployments of MySQL and Percona XtraDB Cluster.

In past versions of MySQL, there was often an 'upgrade dance' that had to be performed in starting up a newly upgraded MySQL instance with the previous version configuration file. In some cases a few deprecated options might no longer be supported in the newer server version, triggering an error and a subsequent shutdown moments after starting. The same thing can happen even outside of upgrade scenarios if a configuration change was made with a mistake or typo in the variable name or value.

As of MySQL 8.0.16 and later, there is now a **'validate-config'** option to quickly test and validate server configuration options without having to start the server. Once used, if no issues are found with the configuration file, the server will exit with an exit code of zero (0). If a problem is found, the server will exit with an error code of one (1) for the first occurrence of anything that is determined to be invalid.

Validating a single option

For example, consider the following server option (old_passwords) which was removed in MySQL 8:

```
./mysqld --old_passwords=1 --validate-config 2021-03-25T17:56:49.932782Z 0 [ERROR] [MY-000067]
[Server] unknown variable 'old_passwords=1'. 2021-03-25T17:56:49.932862Z 0 [ERROR] [MY-010119]
[Server] Aborting
```

Note the server exited with an error code of one (1) and is showing the error for the invalid option.

Validating a configuration file

It is also possible to validate an entire my.cnf configuration file to check all options:

```
./mysqld --defaults-file=/home/sandbox/my.cnf --validate-config 2021-03-25T18:03:41.938734Z 0
[ERROR] [MY-000067] [Server] unknown variable 'old_passwords=1'. 2021-03-25T18:03:41.938865Z 0
[ERROR] [MY-010119] [Server] Aborting
```

Note that the server exited on the **first occurrence** of an invalid value. Any remaining errors in the configuration file will need to be found after correcting the first error and running the **validate-config** option again. So in this example, I've now removed the **'old_passwords=1'** option in the configuration file, and I need to run **validate-config** again to see if there are any other errors remaining:

```
./mysqld --defaults-file=/home/sandbox/my.cnf --validate-config 2021-03-25T18:08:28.612912Z
0 [ERROR] [MY-000067] [Server] unknown variable 'query_cache_size=41984'.
2021-03-25T18:08:28.612980Z 0 [ERROR] [MY-010119] [Server] Aborting
```

Indeed, there is yet another option that was removed from MySQL 8 in the configuration file, so after running the validate again (after fixing the first issue), we've now identified a second problem. After removing the **query_cache_size** option from the my.cnf file and running **validate-config** again, we finally get a clean bill of health:

```
./mysqld --defaults-file=/home/sandbox/my.cnf --validate-config
```

Change validation verbosity

By default, the validate-config option will only report error messages. If you are also interested in seeing any warnings or informational messages, you can change the log_error_verbosity with a value that is greater than one (1):

```
./mysqld --defaults-file=/home/sandbox/my.cnf --log-error-verbosity=2 --validate-config
2021-03-25T18:20:01.380727Z 0 [Warning] [MY-000076] [Server] option 'read_only': boolean value
'y' was not recognized. Set to OFF.
```

Now we are seeing warning level messages, and the server is exiting with an error code of zero (0) as there are technically no errors, only warnings. Taking this further, I've added the query_cache_size option from above back in the my.cnf file, and if we run the validate again we see both errors and warnings this time, while the server exits with an error code of one (1) as there really was an error:

```
./mysqld --defaults-file=/home/sandbox/my.cnf --log-error-verbosity=2 --validate-config
2021-03-25T18:23:11.364288Z 0 [Warning] [MY-000076] [Server] option 'read_only': boolean value
'y' was not recognized. Set to OFF. 2021-03-25T18:23:11.372729Z 0 [ERROR] [MY-000067] [Server]
unknown variable 'old_passwords=1'. 2021-03-25T18:23:11.372795Z 0 [ERROR] [MY-010119] [Server]
Aborting
```

Older version alternative

During my testing of the **validate_config** feature, a colleague pointed out that there is a way to replicate this validation on older MySQL versions by using a combination of 'help' and 'verbose' options as below:

```
$ mysqld --defaults-file=/tmp/my.cnf --verbose --help 1>/dev/null; echo $? 0   $ echo default_
table_encryption=OFF >> /tmp/my.cnf   $ mysqld --defaults-file=/tmp/my.cnf --verbose --help
1>/dev/null; echo $? 2021-03-26T08:43:04.987265Z 0 [ERROR] unknown variable 'default_table_
encryption=OFF' 2021-03-26T08:43:04.990898Z 0 [ERROR] Aborting 1   $ mysqld --version mysqld  Ver
5.7.33-36 for Linux on x86_64 (Percona Server (GPL), Release 36, Revision 7e403c5)
```

Closing thoughts

While not perfect, the validate-config feature goes a long way towards making upgrades and configuration changes easier to manage. It is now possible to know with some certainty that your configuration files or options are valid prior to restarting the server and finding an issue that ends up keeping your node from starting normally leading to unexpected downtime.

Don’t Start Using Your MySQL Server Until You’ve Configured Your OS

By Denis Subbota

Denis has been with Percona since March 2021 and is a MySQL DBA I in Percona Managed Services. With a passion for IT technology, he excels in database management and optimization, bringing valuable expertise to his current role.

Whenever you install your favorite MySQL server on a freshly created Ubuntu instance, you start by updating the configuration for MySQL, such as configuring buffer pool, changing the default datadir director, and disabling one of the most outstanding features - query cache. It’s a nice thing to do, but first things first. Let’s review the best practices we usually follow in Managed Services before using your MySQL server in production and stage env, even for home play purposes.

Memory

Our usual recommendation is to use specific memory parameters, which we suggest to ensure optimal performance.

- To prevent out-of-memory (OOM) episodes, the OOM Score has to be set to -800.
- vm.swappiness = 1
- Disable Transparent Huge Pages
- Install and enable jemalloc. Let's briefly go through each setting to understand why adjustments are needed. Afterward, we will see how to configure these settings on your OS.

OOM

The OOM killer checks oom_score_adj to adjust its final calculated score. This file is present in /proc/$pid/oom_score_adj. You can add a sizable negative score to this file to ensure that OOM killer is less likely to pick up and terminate your process. The oom_score_adj can vary from -1000 to 1000. If you assign -1000 to it, it can use 100% memory and avoid getting terminated by OOM killer. On the other hand, if you assign 1000 to it, the Linux kernel will keep killing the process even when it uses minimal memory.

Swappiness

Swappiness is a Linux kernel parameter determining how aggressively the Linux virtual machine swaps pages between memory and the swap space on the system's disk. The default value of vm.swappiness is 60, representing the percentage of free memory before activating the swap. Lower values reduce swapping and keep more memory pages in physical memory. Changing the value directly influences the performance of the Linux system. These values are defined as:

- 0: swap is disabled
- 1: Minimum amount of swapping without disabling it entirely
- 10: recommended value to improve performance when sufficient memory exists in a system
- 100: aggressive swapping

Transparent Huge Pages and Jemalloc

When it comes to Transparent Huge Pages (THP), they can take up more memory. The kernel's memory allocation function allocates the requested page size, and sometimes more, rounded up to fit within the available memory. In other words, even if your application requires a small amount of memory, it will still be allocated at least a full page.

Additionally, pages must be contiguous in memory, which applies to 'huge pages.' This means that if the server cannot find a full page available in a row, it will defragment the memory before allocating it. This can negatively impact performance and cause delays.

InnoDB is built on a B*-Tree of indices, meaning that its workload will usually have sparse rather than contiguous-memory access, and, as such, it will likely noticeably perform worse with THP.

If you use jemalloc in conjunction with THP, the server may run out of memory over time because unused memory cannot be freed. Therefore, disabling Transparent Huge Pages for database servers is advisable to avoid this situation.

Using jemalloc instead of glibc memory allocator for MySQL results in less memory fragmentation and more efficient resource management. This is especially true when Transparent Huge Pages are disabled.

Action steps for memory settings

Before we change what needs to be adjusted, we need to know the current situation on our DB instance. By the way, I assume you installed the pt-toolkit and your favorite MySQL server to make your life easier. If you haven't, please install it (Percona Toolkit documentation).

```
echo " -- THP check";cat /sys/kernel/mm/transparent_hugepage/enabled;  cat /sys/kernel/mm/
transparent_hugepage/defrag;echo " --Swappiness";sysctl vm.swappiness ; cat /etc/sysctl.conf |
grep -i swap;echo " -- OOM for MySQL";cat /proc/$(pidof mysqld)/oom_score ; cat /proc/$(pidof
mysqld)/oom_score_adj;echo " -- jemalloc"; sudo pt-mysql-summary | grep -A5 -i "memory management"
; sudo grep -i jem /proc/$(pidof mysqld)/maps
```

We want to see something like below, but I am sure we are not.

```
> echo " -- THP check";cat /sys/kernel/mm/transparent_hugepage/enabled;  cat /sys/kernel/mm/
transparent_hugepage/defrag;echo " --Swappiness";sysctl vm.swappiness ; cat /etc/sysctl.conf | grep
-i swap;echo " -- OOM for MySQL";cat /proc/$(pidof mysqld)/oom_score ; cat /proc/$(pidof mysqld)/
oom_score_adj;echo " -- jemalloc"; sudo pt-mysql-summary | grep -A5 -i "memory management" ; sudo
grep -i jem /proc/$(pidof mysqld)/maps

 -- THP check
always madvise [never]
always defer defer+madvise madvise [never]
 --Swappiness
vm.swappiness = 1
vm.swappiness = 1
 -- OOM for MySQL
0
-800
 -- jemalloc
# Memory management library ###################################
jemalloc enabled in mysql config for process with id 29584
Using jemalloc from /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
# The End #####################################################
7f3456ac1000-7f3456af4000 r-xp 00000000 08:01 63812       /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
7f3456af4000-7f3456cf3000 ---p 00033000 08:01 63812       /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
7f3456cf3000-7f3456cf5000 r--p 00032000 08:01 63812       /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
7f3456cf5000-7f3456cf6000 rw-p 00034000 08:01 63812       /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
```

Let's quickly fix it.

Disable THP

Let's create a service which will disable THP for us:

```
sudo su -
cat <<EOF > /usr/lib/systemd/system/disable-thp.service
[Unit]
Description=Disable Transparent Huge Pages (THP)

[Service]
Type=simple
ExecStart=/bin/sh -c "echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled && echo 'never'
> /sys/kernel/mm/transparent_hugepage/defrag"

[Install]
WantedBy=multi-user.target
EOF
```

And the below command to enable this service:

```
sudo systemctl daemon-reload
sudo systemctl start disable-thp
sudo systemctl enable disable-thp
```

vm.swappiness = 1
Change swappiness at runtime.

```
echo 1 > /proc/sys/vm/swappiness
```

And let's persist it in the config file:

```
echo "# Swappiness" >> /etc/sysctl.conf
echo "vm.swappiness = 1" >> /etc/sysctl.conf
```

And enable this change.

```
sudo sysctl -p
```

OOM and Jemalloc

We are halfway through improving things, but let's keep pushing for better memory usage. Let's install jemalloc.

```
sudo apt-get install libjemalloc1
```

Please confirm that we have it on the correct path:

```
ls -l /usr/lib/x86_64-linux-gnu/libjemalloc.so.1
```

And the last thing we need to push our MySQL service to use our magic jemalloc library, let's create an override for systemd:

Note: Depending on the system, it can be shown as mysql or mysqld. You can use systemctl | grep mysql to get the proper mysql service name.

```
sudo systemctl edit mysql
```

Add the specified content to the file below immediately.

```
[Service]
Environment= "LD_PRELOAD=/usr/lib64/libjemalloc.so.1"
OOMScoreAdjust=-800
```

To apply this change, we need to reload daemon and mysql service.

```
sudo systemctl daemon-reload
sudo systemctl restart mysql
```

The optimization of our memory settings has been completed successfully. You can verify it by executing the same check above.

```
echo " -- THP check";cat /sys/kernel/mm/transparent_hugepage/enabled;  cat /sys/kernel/mm/
transparent_hugepage/defrag;echo " --Swappiness";sysctl vm.swappiness ; cat /etc/sysctl.conf |
grep -i swap;echo " -- OOM for MySQL";cat /proc/$(pidof mysqld)/oom_score ; cat /proc/$(pidof
mysqld)/oom_score_adj;echo " -- jemalloc"; sudo pt-mysql-summary | grep -A5 -i "memory management"
; sudo grep -i jem /proc/$(pidof mysqld)/maps
```

Mount point option for disk

Another thing I want to address in this article is how to reduce IO stress on our disks. It's one of the most straightforward tasks we have, but it will give us a lot of performance for our powerful disks, which keeps our databases healthy and durable.

By default, when most disks are mounted using the relatime option, the system updates the metadata statistics for files each time they are accessed or changed on the mount point. This process can result in a significant amount of IO usage, which can be particularly problematic when running a database on that mount point. Given that MySQL typically accesses and writes numerous files concurrently, we must prioritize IO for more critical processes within the database rather than for updating metadata. Therefore, it is advisable to refrain from using the **relatime** option by default in such scenarios. To make this happen, we need to update it to **noatime,nodiratime.**

How to check the current options we have: I assume that you are using a separate mount point for the MySQL database attached to **/var/lib/mysql** path.

```
sudo mount | grep "/var/lib/mysql"
```

The result you will more likely get is:

```
/dev/sdb on /var/lib/mysql type ext4 (rw,relatime)
```

Action steps to apply best practices for disk settings
Let's find out where we have these disk settings for that **fstab** coming to help.

```
> cat /etc/fstab |  grep "/var/lib/mysql"
/dev/sdb      /var/lib/mysql     ext4 defaults     0   0
```

So it's easy to update the fstab file and add the required options for mount point = noatime, nodiratime.

```
sudo vim  /etc/fstab
/dev/sdb         /var/lib/mysql     ext4 defaults,noatime,nodiratime    0   0
```

From that moment, we are almost done, but we can't apply these changes until our MySQL server is running, so we need to stop our mysql service, umount datadir directory, and mount it with new options.

```
sudo systemctl stop mysql
```

Once MySQL service is stopped, we can unmount our /mysql directory,

```
sudo umount /var/lib/mysql
```

and mount it again using updated /etc/fstab settings:

```
sudo mount -av
```

At that point, disk settings should be good, but it's worth verifying that we have the desired mount point options. Afterward, we can start the MySQL service:

```
> sudo mount | grep  grep "/var/lib/mysql"
/dev/sdb on /var/lib/mysql type ext4 (rw,noatime,nodiratime)
```

We see the options are correct, so we can start the mysql service.

```
sudo systemctl start mysql
```

Conclusion

Optimizing memory and disk settings for MySQL can greatly improve the performance and stability of your database. Following the steps outlined in this article, you can reduce IO stress on your disks, prioritize IO for critical processes within the database, and improve memory usage. Remember always to verify your changes and consult with a professional if you have any questions or concerns. With these optimizations in place, your MySQL database will be better equipped to handle the demands of your applications and users.

Indexing

Indexes, when optimized, can dramatically enhance the speed at which queries are executed, elevating the overall efficiency of a MySQL database. Through the collective wisdom of our experts, this chapter covers their strategies aimed at refining the functionality of indexes to facilitate quicker data retrieval while significantly reducing the load on resources. And while there are so many strategies out there for doing so, these can serve as a starting point for developers and DBAs looking to optimize their MySQL indexes for better performance.

MySQL Query Performance: Not Just Indexes

By Peter Zaitsev

Today, I'll look at whether optimizing indexing is always the key to improving MySQL query performance (spoiler, it isn't).

As we look at MySQL query performance, our first concern is often whether a query is using the right indexes to retrieve the data. This is based on the assumption that finding the data is the most expensive operation – and the one you should focus on for MySQL query optimization. However, this is not always the case.

Let's look at this query for illustration:

```
mysql> show create table tbl \G
*************************** 1. row ***************************
       Table: tbl
Create Table: CREATE TABLE `tbl` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `k` int(11) NOT NULL DEFAULT '0',
 `g` int(10) unsigned NOT NULL,
 PRIMARY KEY (`id`),
 KEY `k_1` (`k`)
) ENGINE=InnoDB AUTO_INCREMENT=2340933 DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql> explain select g,count(*) c from tbl where k<1000000 group by g having c>7 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tbl
   partitions: NULL
         type: ALL
possible_keys: k_1
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 998490
     filtered: 50.00
        Extra: Using where; Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)

mysql> select g,count(*) c from tbl where k<1000000 group by g having c>7;
+--------+----+
| g      | c  |
+--------+----+
|  28846 |  8 |
| 139660 |  8 |
| 153286 |  8 |
...
| 934984 |  8 |
+--------+----+

22 rows in set (6.80 sec)
```

Looking at this query, many might assume the main problem is that this query is doing a full table scan. One could wonder then, "Why does the MySQL optimizer not use index (k)?" (It is because the clause is not selective enough, by the way.) This thought might cause someone to force using the index, and get even worse performance:

```
mysql> select g,count(*) c from tbl force index(k) where k<1000000 group by g having c>7;
+--------+----+
| g      | c  |
+--------+----+
|  28846 |  8 |
| 139660 |  8 |
...
| 934984 |  8 |
+--------+----+
22 rows in set (9.37 sec)
```

Or someone might extend the index on (k) to (k,g) to be a covering index for this query. This won't improve performance either:

```
mysql> alter table tbl drop key k_1, add key(k,g);
Query OK, 0 rows affected (5.35 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain select g,count(*) c from tbl where k<1000000 group by g having c>7 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tbl
   partitions: NULL
         type: range
possible_keys: k
          key: k
      key_len: 4
          ref: NULL
         rows: 499245
     filtered: 100.00
        Extra: Using where; Using index; Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)

mysql> select g,count(*) c from tbl where k<1000000 group by g having c>7;
+--------+----+
| g      | c  |
+--------+----+
|  28846 |  8 |
| 139660 |  8 |
...
| 915436 |  8 |
| 934984 |  8 |
+--------+----+
22 rows in set (6.80 sec)
```

This wasted effort is all due to focusing on the wrong thing: figuring out how can we find all the rows that match k<1000000 as soon as possible. This is not the problem in this case. In fact, the query that touches all the same columns but doesn't use GROUP BY runs 10 times as fast:

```
mysql> select sum(g) from tbl where k<1000000;
+--------------+
| sum(g)       |
+--------------+
| 500383719481 |
+--------------+
1 row in set (0.68 sec)
```

For this particular query, whether or not it is using the index for lookup should not be the main question. Instead, we should look at how to optimize GROUP BY – which is responsible for some 90% of the query response time.

Putting the Fun in MySQL Functional Indexes

By David Stokes

David is a Technology Evangelist for Percona, and is the author of MySQL & JSON - A Practical Programming Guide.

Functional indexes are found in both of Percona's relational databases, MySQL and PostgreSQL, but they are probably the least used and most understood index type, aside from geospatial. You may also hear this type of index being called Index on expression.

So, what is a functional index?

Definitions

The examples below use MySQL, but it is helpful to check their competition's documentation to help explain what a functional index is. PostgreSQL defines them clearly. For a functional index, an index is defined as the result of a function applied to one or more columns of a single table. Functional indexes can be used to obtain fast access to data based on the result of function calls.

MySQL's definition is a little more convoluted, stating that in MySQL versions 8.0.13 and higher support functional key parts that index expression values rather than column or column prefix values. The use of functional key parts enables the indexing of values not stored directly in the table.

Okay, enough technical manual-ese. Functional indexes allow you to create an index based on a calculation or function to a value of a column in a table. You could index the total price of a good by adding the sales or VAT tax to get the cost of a good. Or add twenty-four hours to the checkout time of a rental item to determine the return time. The possibilities are only limited by your data needs and your imagination.

Why is this a good thing?

PostgreSQL's manual expresses this succinctly. The index expressions are not recomputed during an indexed search since they are already stored in the index. In both examples below, the system sees the query as just WHERE indexed column = 'constant' and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

What was that about insert and update speed? Well, there is always a drawback to having the server do something complicated for you. In this case, the calculation on the expression must be completed before the data can be inserted into the row. That is a light overhead that could grow into something that overwhelms the server with an extremely heavy write load.

MySQL's functional indexes are implemented as hidden virtual generated columns The virtual generated column itself requires no storage. The index itself takes up storage space as any other index.

Example one

Anyone who has used Structured Query Language (SQL) for over thirty seconds has run into a situation where the query 'looks' good, but its results do not, or it runs much more slowly than expected. Recently, someone contacted me about a functional index that just was not working. To demonstrate, we will start with a simple table with a datetime field. The desired result includes picking all the rows in the table where the month part of the DateTime field matches a particular month.

```
SQL > create table product (id int, date_available datetime, price double, weight int);
Query OK, 0 rows affected (0.0124 sec)
SQL > insert into product values (1,now(),10.00,1),(32,now(),7.74,2);
Query OK, 2 rows affected (0.0043 sec)

Records: 2 Duplicates: 0 Warnings: 0
SQL > alter table product add index((month(date_available)));
Query OK, 0 rows affected (0.0178 sec)

Records: 0 Duplicates: 0 Warnings: 0
```

If we double-check that index, it looks pretty good.

```
SQL > SHOW INDEX FROM product \G
*************************** 1. row ***************************
Table: product
Non_unique: 1
Key_name: functional_index
Seq_in_index: 1
Column_name: NULL
Collation: A
Cardinality: 1
Sub_part: NULL
Packed: NULL
Null: YES
Index_type: BTREE
Comment:
Index_comment:
Visible: YES
Expression: month(`date_available`)
1 row in set (0.0017 sec)
SQL >
```

Heck, the system even calls it functional_index for our benefit. Please note that future functional indexes get _n postpended to the name.

We should be good to go, but the wrong thing happens. When the query is run, the optimizer does not use the new index.

```
SQL > explain select id from product where month(date_available) = '8';
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+--
-----------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
|
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+--
-----------+
| 1 | SIMPLE | product | NULL | ALL | NULL | NULL | NULL | NULL | 2 | 100 | Using where |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+--
-----------+
1 row in set, 1 warning (0.0014 sec)
Note (code 1003): /* select#1 */ select `test`.`product`.`id` AS `id` from `test`.`product` where
(month(`test`.`product`.`date_available`) = '8')
```

Wrong month?

My first assumption was that I got the value for the month wrong. But when I checked, it was correct.

```
SQL > select month(now());
+--------------+
| month(now()) |
+--------------+
| 8 |
+--------------+
1 row in set (0.0007 sec)
```

Some of you may have already guessed that the mistake was mine, but not what I thought it was. Eventually, I retyped the query, but this time, I left off the single quotes around the value of the month.

```
SQL > explain select id from product where month(date_available) = 8 \G
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: product
partitions: NULL
type: ref
possible_keys: functional_index
key: functional_index
key_len: 5
ref: const
rows: 4
filtered: 100
Extra: NULL
1 row in set, 1 warning (0.0012 sec)
Note (code 1003): /* select#1 */ select `test`.`product`.`id` AS `id` from `test`.`product` where
(month(`date_available`) = 8)
SQL >
```

Casting

Sometimes, you are going to get that data with the quotes, and you need to scrap them off. In these circumstances, you can use the cast operator to do just that. And cast takes of double and single quotes.

```
SQL > explain select id from product where month(date_available) = cast(“8” as unsigned) \G
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: product
partitions: NULL
type: ref
possible_keys: functional_index
key: functional_index
key_len: 5
ref: const
rows: 4
filtered: 100
Extra: NULL
1 row in set, 1 warning (0.0013 sec)
Note (code 1003): /* select#1 */ select `test`.`product`.`id` AS `id` from `test`.`product` where
(month(`date_available`) = cast(‘8’ as unsigned))
SQL >
```

Naming functional indexes

I recommend naming your functions, and, like other MySQL indexing options, you can name your functional indexes. Use a name that describes what the index is doing. This makes it easier to quickly identify than looking up the definition of functional_index_1, functional_index_2, etc., from a SHOW CREATE TABLE statement.

```
SQL > alter table product add index product_shipping ( (weight * 10) );
Query OK, 0 rows affected (0.0329 sec)

Records: 0 Duplicates: 0 Warnings: 0

SQL > explain select * from product where (weight * 10) > 5 \G
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: product
partitions: NULL
type: range
possible_keys: product_shipping
key: product_shipping
key_len: 9
ref: NULL
rows: 4
filtered: 100
Extra: Using where
1 row in set, 1 warning (0.0014 sec)
Note (code 1003): /* select#1 */ select `test`.`product`.`id` AS `id`,`test`.`product`.`date_
available` AS `date_available`,`test`.`product`.`price` AS `price`,`test`.`product`.`weight` AS
`weight` from `test`.`product` where ((`weight` * 10) > 5)
```

Conclusion

Functional Indexes, like other indexes, are a handy way to speed up queries, but be sure to test them with EXPLAIN to ensure the optimizer knows to use them.

Duplicate, Redundant, and Invisible Indexes

By Arunjith Aravindan

Arunjith is a Senior MySQL DBA in Managed Services and consults with Percona's customers on building and maintaining reliable and high-performance MySQL infrastructures. Arunjith joined Percona in July 2014.

MySQL index is a data structure used to optimize the performance of database queries at the expense of additional writes and storage space to keep the index data structure up to date. It is used to quickly locate data without having to search every row in a table. Indexes can be created using one or more columns of a table, and each index is given a name. Indexes are especially useful for queries that filter results based on columns with a high number of distinct values.

Indexes are useful for our queries, but duplicate, redundant, and unused indexes reduce performance by confusing the optimizer with query plans, requiring the storage engine to maintain, calculate, and update more index statistics, as well as requiring more disk space.

Since MySQL 8.0, indexes can be marked as invisible. In this article, I will detail how to detect duplicate and underused indexes as well as the new feature of invisible indexes and how it can help with index management.

How to find the duplicate MySQL indexes?

pt-duplicate-key-checker is a command-line tool from Percona Toolkit that scans a MySQL database and identifies tables that have duplicate indexes or primary keys. It can help identify potential problems with the schema and provide guidance on how to fix them.

For each duplicate key, the tool prints a DROP INDEX statement by default, allowing you to copy-paste the statement into MySQL to get rid of the duplicate key.

For example:

```
$ pt-duplicate-key-checker --host=localhost --user=percona --ask-pass
Enter password:
# ########################################################################
# mytestdb.authors_test
# ########################################################################


# idx_first_name is a left-prefix of idx_first_name_last_name
# Key definitions:
#   KEY `idx_first_name` (`first_name`),
#   KEY `idx_first_name_last_name` (`first_name`,`last_name`),
# Column types:
#       `first_name` varchar(50) default null
#       `last_name` varchar(50) default null
# To remove this duplicate index, execute:
ALTER TABLE `mytestdb`.`authors_test` DROP INDEX `idx_first_name`;


# Key idx_last_name_id ends with a prefix of the clustered index
# Key definitions:
#   KEY `idx_last_name_id` (`last_name`,`id`)
#   PRIMARY KEY (`id`),
# Column types:
#       `last_name` varchar(50) default null
#       `id` int not null auto_increment
# To shorten this duplicate clustered index, execute:
ALTER TABLE `mytestdb`.`authors_test` DROP INDEX `idx_last_name_id`, ADD INDEX `idx_last_name_id`
(`last_name`);


# ########################################################################
# Summary of indexes
# ########################################################################


# Size Duplicate Indexes   4125830
# Total Duplicate Indexes  2
# Total Indexes            129
```

How to find the unused MySQL indexes?

Unused indexes are indexes that are created in a database but are not being used in any queries. These indexes take up space in the database and can slow down query performance if they are not maintained. Unused indexes should be identified and removed to improve the performance of the database.

We can use the sys.schema unused indexes view to see which indexes have not been used since the last time the MySQL server was restarted. Example:

```
mysql> select * from sys.schema_unused_indexes
       where index_name not like 'fk_%' and object_schema='mytestdb' and object_name='testtable';
+---------------+-------------+----------------+
| object_schema | object_name | index_name     |
+---------------+-------------+----------------+
| mytestdb      | testtable   | idx_first_name |
| mytestdb      | testtable   | idx_last_name  |
+---------------+-------------+----------------+
2 rows in set (0.00 sec)
```

Since the index idx_first_name was used to retrieve the records, it is not listed.

```
mysql> select last_name from testtable WHERE first_name='Arun';
+-----------+
| last_name |
+-----------+
| Jith      |
+-----------+
1 rows in set (0.00 sec)

mysql> select * from sys.schema_unused_indexes
       where index_name not like 'fk_%' and object_schema='mytestdb' and object_name='testtable';
+---------------+-------------+---------------+
| object_schema | object_name | index_name    |
+---------------+-------------+---------------+
| mytestdb      | testtable   | idx_last_name |
+---------------+-------------+---------------+
1 row in set (0.01 sec)
```

Use invisible indexes before deleting MySQL indexes

In MySQL 8.0, there is a feature that allows you to have an invisible index. This means that an index is created on a table, but the optimizer does not use it by default. By using this feature, you can test the impacts of removing an index without actually dropping it. If desired, the index can be made visible again, therefore avoiding the time-consuming process of re-adding the index to a larger table.

The SET_VAR(optimizer_switch = 'use_invisible_indexes=on') allows the invisible index to be used for specific application activities or modules during a single query while preventing it from being used across the entire application. Let's look at a simple example to see how it works.

```
CREATE TABLE `authors` (
  `id` int NOT NULL AUTO_INCREMENT,
  `first_name` varchar(50),
  `last_name` varchar(50),
  `email` varchar(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
   PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`)
) ENGINE=InnoDB;
```

The indexes are set to visible by default, and we may use the ALTER TABLE table name ALTER INDEX index name to make it INVISIBLE/VISIBLE.

To find the index details, we can use “SHOW INDEXES FROM table;” or query the INFORMATION SCHEMA. STATISTICS.

```
mysql> SELECT INDEX_NAME, IS_VISIBLE
       FROM INFORMATION_SCHEMA.STATISTICS
       WHERE TABLE_SCHEMA = 'mytestdb' AND TABLE_NAME = 'authors_test';
+------------+------------+
| INDEX_NAME | IS_VISIBLE |
+------------+------------+
| email      | YES        |
| PRIMARY    | YES        |
+------------+------------+
2 rows in set (0.00 sec)

mysql> ALTER TABLE authors_test ALTER INDEX email INVISIBLE;
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT INDEX_NAME, IS_VISIBLE
       FROM INFORMATION_SCHEMA.STATISTICS
       WHERE TABLE_SCHEMA = 'mytestdb' AND TABLE_NAME = 'authors_test';
+------------+------------+
| INDEX_NAME | IS_VISIBLE |
+------------+------------+
| email      | NO         |
| PRIMARY    | YES        |
+------------+------------+
2 rows in set (0.00 sec)
```

The query cannot utilize the index on the email column since it is invisible.

```
mysql> explain select email from authors_test WHERE email='amanda31@example.org'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: authors_test
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10063
     filtered: 10.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

The SET VAR optimizer hint can be used to temporarily update the value of an optimizer switch and enable invisible indexes for a single query only, as shown below:

```
mysql> explain select /*+ SET_VAR(optimizer_switch = 'use_invisible_indexes=on') */ first_name
               from authors_test WHERE email='amanda31@example.org'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: authors_test
   partitions: NULL
         type: const
possible_keys: email
          key: email
      key_len: 302
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.00 sec)
```

Summary

Removal of duplicate and unnecessary indexes is recommended to prevent performance degradation in high concurrent workloads with less-than-ideal data distribution on a table. Unwanted keys take up unnecessary disc space, which can cause overhead to DML and read queries. To test the effects of dropping an index without fully removing it, it can be made invisible first.

An Overview of Indexes in MySQL 8.0: MySQL CREATE INDEX, Functional Indexes, and More

By Corrado Pandiani

Prior to joining Percona as a Senior Consultant, Corrado spent more than 20 years developing websites and designing and administering MySQL. He is now an Architect in Professional Services and his skills are focused on performance and architectural design.

Working with hundreds of different customers, I often face similar problems around running queries. One very common problem when trying to optimize a database environment is index usage. A query that cannot use an index is usually a long-running one, consuming more memory or triggering more disk iops.

A very common case is when a query uses a filter condition against a column that is involved in some kind of functional expression. An index on that column can not be used.

Starting from MySQL 8.0.13, functional indexes are supported. In this article, I will first explain an overview of indexes in MySQL and cover the MySQL CREATE INDEX before diving into showing what functional indexes are and how they work.

Introduction to MySQL indexes

Indexes in MySQL are database structures used to optimize data retrieval speed and efficiency. They are a roadmap that speeds up the process of finding specific rows in a large table. By creating indexes on columns, MySQL creates a separate data structure that holds a sorted version of the indexed column's values, enabling the database engine to quickly locate rows that match specific queries by reducing the need for full-table scans. But, while indexes greatly enhance read operations, they can also impact the speed of write operations, as the indexes need to be updated whenever data changes. As such, index design must strike a balance between improved query speed and efficient data modification.

What is an index?

An index in MySQL is made up of several elements:

Index columns – These are the table columns on which the index is created. Each value in the index column is associated with the primary key of the related row in the table.

Index key – Representing a sorted data structure, the index key incorporates values from the index columns and pointers to the relevant table rows. This structure makes for faster data access.

The relationship between the index and the underlying data is based on a hierarchical structure that allows the database engine to quickly find the desired rows based on the index key's sorted values.

The process of crafting indexes involves a trade-off between improved data retrieval speed and impact on data modification operations. While indexes enhance query performance by reducing the need for full-table scans, some things need consideration:

- Choosing Appropriate Columns: Optimal column selection for indexing holds significance. Columns frequently used in queries for filtering or sorting are solid candidates. Over-indexing on too many columns could lead to unnecessary overhead.
- Balancing Index Size and Query Performance: The size of an index impacts its performance; larger indexes may slow down query processing. As such, index design should consider the size of indexed columns and the potential benefits they offer.
- Trade-offs: While indexes accelerate read operations, they can slow down write operations, as data modifications require index updates. A careful approach is required to ensure optimal performance for both read and write operations.

An index within MySQL is a powerful tool that considerably improves data retrieval speed via a structured way to access specific rows within a table. However, DBAs must strike a balance between query performance, index size, and the overall efficiency of operations.

MySQL CREATE INDEX

The MySQL `CREATE INDEX` statement syntax is for creating various types of indexes to enhance query performance. Here's how to create different types of indexes, along with specifying index columns and names:

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column1 [, column2, ...]);
```

Explanations:
– `CREATE INDEX`: Initiates the creation of an index.
– `UNIQUE` (optional): Specifies that the index values must be unique across the table.
– `index_name`: Specifies the name of the index being created.
– `table_name`: The name of the table on which the index is being created.
– `(column1 [, column2, ...])`: Lists the columns that the index will be based on. Multiple columns can be

specified to create composite indexes.

To create different types of indexes:

Non-unique index:

```
CREATE INDEX idx_column ON table_name (column_name);
```

Unique index:

```
CREATE UNIQUE INDEX idx_unique_column ON table_name (column_name);
```

Composite index (index on multiple columns):

```
CREATE INDEX idx_multi_columns ON table_name (column1, column2);
```

Prefix index (index on first N characters of a column):

```
CREATE INDEX idx_prefix ON table_name (column_name(N));
```

Full-text index (for textual data searches):

```
CREATE FULLTEXT INDEX idx_fulltext_column ON table_name (column_name);
```

Spatial index (for spatial data types):

```
CREATE SPATIAL INDEX idx_spatial_column ON table_name (column_name);
```

To specify index columns and names:

- List the desired column(s) within parentheses after the ON clause.
- For composite indexes, separate column names with commas.
- Provide a unique index_name to identify the index.

Example of creating a composite index:

```
CREATE INDEX idx_full_name ON employees (first_name, last_name);
```

Example of creating a unique index:

```
CREATE UNIQUE INDEX idx_unique_email ON users (email);
```

By identifying the particular columns utilized in filtering, joining, sorting, and text-based search operations and employing the CREATE INDEX statement to generate suitable indexes, you can improve the query performance of your MySQL database.

The well-known indexing problem

As already mentioned, a very common problem with index usage is when you have a filter condition against one or more columns involved in some kind of functional expression.

Let's see a simple example.

You have a table called products containing the details of your products, including a create_time TIMESTAMP column. If you would like to calculate the average price of your products in a specific month, you could do the following:

```
mysql>; SELECT AVG(price) FROM products WHERE MONTH(create_time)=10;
+------------+
| AVG(price) |
+------------+
| 202.982582 |
```

The query returns the right value, but take a look at the EXPLAIN:

```
mysql>; EXPLAIN SELECT AVG(price) FROM products WHERE MONTH(create_time)=10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 99015
     filtered: 100.00
        Extra: Using where
```

The query triggers a full scan of the table. Let's create an index on create_time and check again:

```
mysql> ALTER TABLE products ADD INDEX(create_time);
Query OK, 0 rows affected (0.71 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain SELECT AVG(price) FROM products WHERE MONTH(create_time)=10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 99015
     filtered: 100.00
        Extra: Using where
```

A full scan again. The index we have created is not effective. Indeed any time an indexed column is involved in a function, the index can not be used.

To optimize the query, the workaround is rewriting it differently to isolate the indexed column from the function.

Let's test the following equivalent query:

```
mysql> SELECT AVG(price) FROM products WHERE create_time BETWEEN '2019-10-01' AND '2019-11-01';
+------------+
| AVG(price) |
+------------+
| 202.982582 |
+------------+

mysql> EXPLAIN SELECT AVG(price) FROM products WHERE create_time BETWEEN '2019-10-01' AND '2019-
11-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: range
possible_keys: create_time
          key: create_time
      key_len: 5
          ref: NULL
         rows: 182
     filtered: 100.00
        Extra: Using index condition
```

Cool, now the index is used. Then rewriting the query was the typical suggestion.
Quite a simple solution, but not all the times it was possible to change the application code for many valid reasons. So, what to do then?

What is a function-based index?

A function-based index is an indexing technique for databases that allows indexing the results of functions applied to columns rather than the traditional index method that directly stores column values. Instead of indexing raw column data, the function-based index stores function-generated results, which helps to optimize queries and data retrieval efficiency involving complex expressions and functional operations.

Function-based indexes are very useful in scenarios where data retrieval entails complex computations like geographic calculations, date manipulations, or text transformations. By indexing the results of functions applied to columns, function-based indexes are valuable when working with large datasets, as they can enhance the efficiency of aggregations and complex analytical queries.

How do MySQL 8.0 functional indexes work?

Starting from version 8.0.13, MySQL supports functional indexes. Instead of indexing a simple column, you can create the index on the result of any function applied to a column or multiple columns.

Long story short, now you can do the following:

```
mysql> ALTER TABLE products ADD INDEX((MONTH(create_time)));
Query OK, 0 rows affected (0.74 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Be aware of the double parentheses. The syntax is correct since the expression must be enclosed within parentheses to distinguish it from columns or column prefixes.
Indeed the following returns an error:

```
mysql> ALTER TABLE products ADD INDEX(MONTH(create_time));
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near 'create_time))' at line 1
```

Let's check now our original query and see what happens to the EXPLAIN.

```
mysql> SELECT AVG(price) FROM products WHERE MONTH(create_time)=10;
+------------+
| AVG(price) |
+------------+
| 202.982582 |
+------------+


mysql> EXPLAIN SELECT AVG(price) FROM products WHERE MONTH(create_time)=10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: ref
possible_keys: functional_index
          key: functional_index
      key_len: 5
          ref: const
         rows: 182
     filtered: 100.00
        Extra: NULL
```

The query is no longer a full scan and runs faster. The functional_index has been used, with only 182 rows examined. Awesome.

Thanks to the functional index we are no longer forced to rewrite the query.

Which functional indexes are permitted

We have seen an example involving a simple function applied to a column, but you are granted to create more complex indexes.

A functional index may contain any kind of expressions, not only a single function. The following patterns are valid functional indexes:

```
INDEX( ( col1 + col2 ) )
INDEX( ( FUNC(col1) + col2 – col3 ) )
You can use ASC or DESC as well:
INDEX( ( MONTH(col1) ) DESC )
You can have multiple functional parts, each one included in parentheses:
INDEX( ( col1 + col2 ), ( FUNC(col2) ) )
You can mix functional with nonfunctional parts:
INDEX( (FUNC(col1)), col2, (col2 + col3), col4 )
```

There are also limitations you should be aware of:

- A functional key can not contain a single column. The following is not permitted: INDEX((col1), (col2))
- The primary key can not include a functional key part
- The foreign key can not include a functional key part
- SPATIAL and FULLTEXT indexes can not include functional key parts
- A functional key part can not refer to a column prefix

At last, remember that the functional index is useful only to optimize the query that uses the exact same expression. An index created with nonfunctional parts can be used instead to solve multiple different queries.

For example, the following conditions can not rely on the functional index we have created:

```
WHERE YEAR(create_time) = 2019
WHERE create_time > '2019-10-01'
WHERE create_time BETWEEN '2019-10-01' AND '2019-11-01'
WHERE MONTH(create_time+INTERVAL 1 YEAR)
```

All these will trigger a full scan.

Functional index internal

The functional indexes are implemented as hidden virtual generated columns. For this reason, you can emulate the same behavior even on MySQL 5.7 by explicitly creating the virtual column. We can test this, starting by dropping the indexes we have created so far.

```
mysql> SHOW CREATE TABLE products\G
*************************** 1. row ***************************
       Table: products
Create Table: CREATE TABLE `products` (
  `id` int unsigned NOT NULL AUTO_INCREMENT,
  `description` longtext,
  `price` decimal(8,2) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `create_time` (`create_time`),
  KEY `functional_index` ((month(`create_time`)))
) ENGINE=InnoDB AUTO_INCREMENT=149960 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

mysql> ALTER TABLE products DROP INDEX `create_time`, DROP INDEX `functional_index`;
Query OK, 0 rows affected (0.03 sec)
```

We can try now to create the virtual generated column:

```
mysql> ALTER TABLE products ADD COLUMN create_month TINYINT GENERATED ALWAYS AS (MONTH(create_
time)) VIRTUAL;
Query OK, 0 rows affected (0.04 sec)
```

Create the index on the virtual column:

```
mysql>; ALTER TABLE products ADD INDEX(create_month);
Query OK, 0 rows affected (0.55 sec)

mysql>; SHOW CREATE TABLE products\G
*************************** 1. row ***************************
       Table: products
Create Table: CREATE TABLE `products` (
  `id` int unsigned NOT NULL AUTO_INCREMENT,
  `description` longtext,
  `price` decimal(8,2) DEFAULT NULL,
  `create_time` timestamp NULL DEFAULT NULL,
  `create_month` tinyint GENERATED ALWAYS AS (month(`create_time`)) VIRTUAL,
  PRIMARY KEY (`id`),
  KEY `create_month` (`create_month`)
) ENGINE=InnoDB AUTO_INCREMENT=149960 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

We can now try our original query. We expect to see the same behavior as the functional index.

```
mysql>; SELECT AVG(price) FROM products WHERE MONTH(create_time)=10;
+------------+
| AVG(price) |
+------------+
| 202.982582 |
+------------+

mysql> EXPLAIN SELECT AVG(price) FROM products WHERE MONTH(create_time)=10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
   partitions: NULL
         type: ref
possible_keys: create_month
          key: create_month
      key_len: 2
          ref: const
         rows: 182
     filtered: 100.00
        Extra: NULL
```

Indeed, the behavior is the same. The index on the virtual column can be used, and the query is optimized.

The good news is that you can use this workaround to emulate a functional index even on 5.7, getting the same benefits. The advantage of MySQL 8.0 is that it is completely transparent; no need to create the virtual column.

Since the functional index is implemented as a hidden virtual column, there is no additional space needed for the data and only the index space will be added to the table.

By the way, this is the same technique used for creating indexes on JSON documents' fields.

Limitations of functional indexes

In situations where indexed functions may rarely be used in queries, using functional indexes may not be the best choice because the maintenance overhead could outweigh any potential performance benefits. In these scenarios, different approaches may be warranted, including:

- You can optimize queries using query rewriting or using materialized views. By rewriting queries to minimize the requirement of using complex functions and using materialized views to store pre-computed results, query performance can be improved without relying on functional indexes.
- Precomputing and storing function outputs as columns within the table enables traditional indexing on these columns. This minimizes the need for function-based indexes while retaining the advantages of indexed computations.

So, while functional indexes can be great tools for optimizing complex queries, it's important to know the limitations and potential downsides of using them.

Basic Housekeeping for MySQL Indexes

By Daniel Guzmán Burgos

Daniel has worked as a DBA since 2007 for several companies and joined Percona in 2014, where he is the Product Management Engineer in Commercial Product Engineering.

We all know that indexes can be the difference between a high-performance database and a bad/slow/painful query ride. It's a critical part that needs deserves some housekeeping once in a while. So, what should you check? In no particular order, here are some things to look at

1. Unused indexes

With sys schema, is pretty easy to find unused indexes: use the schema_unused_indexes view.

```
mysql> select * from sys.schema_unused_indexes;
+---------------+-----------------+-------------+
| object_schema | object_name     | index_name  |
+---------------+-----------------+-------------+
| world         | City            | CountryCode |
| world         | CountryLanguage | CountryCode |
+---------------+-----------------+-------------+
2 rows in set (0.01 sec)
```

This view is based on the performance_schema.table_io_waits_summary_by_index_usage table, which will require enabling the Performance Schema, the events_waits_current consumer and the wait/io/table/sql/handler instrument. PRIMARY (key) indexes are ignored.

If you don't have them enabled, just execute these queries:

```
update performance_schema.setup_consumers set enabled = 'yes' where name = 'events_waits_
current';
update performance_schema.setup_instruments set enabled = 'yes' where name = 'wait/io/table/sql/
handler';
```

Quoting the documentation:

"To trust whether the data from this view is representative of your workload, you should ensure that the server has been up for a representative amount of time before using it."

And by representative amount, I mean **representative**:

- Do you have a weekly job? Wait at least one week
- Do you have monthly reports? Wait at least one month
- Don't rush!

Once you've found unused indexes, remove them.

2. Duplicated indexes

You have two options here:

- pt-duplicate-key-checker
- the schema_redundant_indexes view from sys_schema

The pt-duplicate-key-checker is part of Percona Toolkit. The basic usage is pretty straightforward:

```
[root@e51d333b1fbe mysql-sys]# pt-duplicate-key-checker
# ############################################################################
# world.CountryLanguage
# ############################################################################

# CountryCode is a left-prefix of PRIMARY
# Key definitions:
#   KEY `CountryCode` (`CountryCode`),
#   PRIMARY KEY (`CountryCode`,`Language`),
# Column types:
#              `countrycode` char(3) not null default ''
#              `language` char(30) not null default ''
# To remove this duplicate index, execute:
ALTER TABLE `world`.`CountryLanguage` DROP INDEX `CountryCode`;

# ############################################################################
# Summary of indexes
# ############################################################################

# Size Duplicate Indexes   2952
# Total Duplicate Indexes  1
# Total Indexes            37
```

Now, the schema_redundant_indexes view is also easy to use once you have sys schema installed. The difference is that it is based on the information_schema.statistics table:

```
mysql> select * from schema_redundant_indexes\G
*************************** 1. row ***************************
              table_schema: world
                table_name: CountryLanguage
      redundant_index_name: CountryCode
   redundant_index_columns: CountryCode
redundant_index_non_unique: 1
       dominant_index_name: PRIMARY
    dominant_index_columns: CountryCode,Language
 dominant_index_non_unique: 0
            subpart_exists: 0
            sql_drop_index: ALTER TABLE `world`.`CountryLanguage` DROP INDEX `CountryCode`
1 row in set (0.00 sec)
```

Again, once you find the redundant index, remove it.

3. Potentially missing indexes

The statements summary tables from the performance schema have several interesting fields. For our case, two of them are pretty important: NO_INDEX_USED (means that the statement performed a table scan without using an index) and NO_GOOD_INDEX_USED ("1" if the server found no good index to use for the statement, "0" otherwise).

Sys schema has one view that is based on the **performance_schema.events_statements_summary_by_digest table**, and is useful for this purpose: statements_with_full_table_scans, which lists all normalized statements that have done a table scan.

For example:

```
mysql> select * from world.CountryLanguage where isOfficial = 'F';
55a208785be7a5beca68b147c58fe634  -
746 rows in set (0.00 sec)

mysql> select * from statements_with_full_table_scans\G
*************************** 1. row ***************************
                  query: SELECT * FROM `world` . `Count ... guage` WHERE `isOfficial` = ?
                     db: world
             exec_count: 1
          total_latency: 739.87 us
     no_index_used_count: 1
no_good_index_used_count: 0
       no_index_used_pct: 100
               rows_sent: 746
           rows_examined: 984
           rows_sent_avg: 746
       rows_examined_avg: 984
              first_seen: 2016-09-05 19:51:31
               last_seen: 2016-09-05 19:51:31
                  digest: aa637cf0867616c591251fac39e23261
1 row in set (0.01 sec)
```

The above query doesn't use an index because there was no good index to use, and thus was reported. See the explain output:

```
mysql> explain select * from world.CountryLanguage where isOfficial = 'F'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: CountryLanguage
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 984
        Extra: Using where
```

Note that the "query" field reports the query digest (more like a fingerprint) instead of the actual query.

In this case, the CountryLanguage table is missing an index over the "isOfficial" field. It is your job to decide whether it is worth it to add the index or not.

4. Multiple column indexes order

It was explained before that Multiple Column index beats Index Merge in all cases when such index can be used, even when sometimes you might have to use index hints to make it work.

But when using them, don't forget that the order matters. MySQL will only use a multi-column index if at least one value is specified for the first column in the index.

For example, consider this table:

```
mysql> show create table CountryLanguage\G
*************************** 1. row ***************************
       Table: CountryLanguage
Create Table: CREATE TABLE `CountryLanguage` (
  `CountryCode` char(3) NOT NULL DEFAULT '',
  `Language` char(30) NOT NULL DEFAULT '',
  `IsOfficial` enum('T','F') NOT NULL DEFAULT 'F',
  `Percentage` float(4,1) NOT NULL DEFAULT '0.0',
  PRIMARY KEY (`CountryCode`,`Language`),
  KEY `CountryCode` (`CountryCode`),
  CONSTRAINT `countryLanguage_ibfk_1` FOREIGN KEY (`CountryCode`) REFERENCES `Country` (`Code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

A query against the field "Language" won't use an index:

```
mysql> explain select * from CountryLanguage where Language = 'English'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: CountryLanguage
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 984
        Extra: Using where
```

Simply because it is not the leftmost prefix for the Primary Key. If we add the "CountryCode" field, now the index will be used:

```
mysql>  explain select * from CountryLanguage where Language = 'English' and CountryCode =
'CAN'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: CountryLanguage
         type: const
possible_keys: PRIMARY,CountryCode
          key: PRIMARY
      key_len: 33
          ref: const,const
         rows: 1
        Extra: NULL
```

Now, you'll have to also consider the selectivity of the fields involved. Which is the preferred order?

In this case, the "Language" field has a higher selectivity than "CountryCode":

```
mysql> select count(distinct CountryCode)/count(*), count(distinct Language)/count(*) from
CountryLanguage;
+---------------------------------------+-----------------------------------+
| count(distinct CountryCode)/count(*) | count(distinct Language)/count(*) |
+---------------------------------------+-----------------------------------+
|                                0.2368 |                            0.4644 |
+---------------------------------------+-----------------------------------+
```

So in this case, if we create a multi-column index, the preferred order will be (Language, CountryCode).

Placing the most selective columns first is a good idea when there is no sorting or grouping to consider, and thus the purpose of the index is only to optimize where lookups. You might need to choose the column order, so that it's as selective as possible for the queries that you'll run most.

Now, is this good enough? Not really. What about special cases where the table doesn't have an even distribution? When a single value is present way more times than all the others? In that case, no index will be good enough. Be careful not to assume that average-case performance is representative of special-case performance. Special cases can wreck performance for the whole application.

In conclusion, we depend heavily on proper indexes. Give them some love and care once in a while, and the database will be very grateful.

All the examples were done with the following MySQL and Sys Schema version:

```
mysql> select * from sys.version;
+-------------+-----------------+
| sys_version | mysql_version   |
+-------------+-----------------+
| 1.5.1       | 5.6.31-77.0-log |
+-------------+-----------------+
```

Memory/resource allocation

Smart memory and resource allocation are essential aspects of MySQL optimization that directly impact your database's efficiency and stability. Proper resource allocation does more than just improve performance; it is also about guaranteeing your databases' scalability and durability in the face of changing workloads and data needs. This chapter looks into the complexity of memory and resource management in MySQL systems, providing insight into how strategic allocation can significantly improve operational efficacy.

You'll find several helpful ways to improve database performance through proper resource management, from fine-tuning your system following a memory upgrade to implementing user-specific resource limits.

Tuning MySQL After Upgrading Memory

By Francisco Bordenave

Daniel has worked as a DBA since 2007 for several companies and joined Percona in 2014, where he is the Product Management Engineer in Commercial Product Engineering.

In this article, we will discuss what to do when you add more memory to your instance. Adding memory to a server where MySQL is running is common practice when scaling resources.

First, some context

Scaling resources is just adding more resources to your environment, and this can be split in two main ways: vertical scaling and horizontal scaling.

Vertical scaling is increasing hardware capacity for a given instance, thus having a more powerful server, while horizontal scaling is adding more servers, a pretty standard approach for load balancing and sharding.

As traffic grows, working datasets are getting bigger, and thus we start to suffer because the data that doesn't fit into memory has to be retrieved from disk. This is a costly operation, even with modern NVME drives, so at some point, we will need to deal with either of the scaling solutions we mentioned.

In this case, we will discuss adding more RAM, which is usually the fastest and easiest way to scale hardware vertically, and also having more memory is probably the main benefit for MySQL.

How to calculate memory utilization

First of all, we need to be clear about what variables allocate memory during MySQL operations, and we will cover only commons ones as there are a bunch of them. Also, we need to know that some variables will allocate memory globally, and others will do a per-thread allocation.

For the sake of simplicity, we will cover this topic considering the usage of the standard storage engine: InnoDB.

We have globally allocated variables:

key_buffer_size: MyISAM setting should be set to 8-16M, and anything above that is just wrong because we shouldn't use MyISAM tables unless for a particular reason. A typical scenario is MyISAM being used by system tables only, which are small (this is valid for versions up to 5.7), and in MySQL 8 system tables were migrated to the InnoDB engine. So the impact of this variable is negligible.

query_cache_size: 0 is default and removed in 8.0, so we won't consider it.

innodb_buffer_pool_size: which is the cache where InnoDB places pages to perform operations. The bigger, the better. :)

Of course, there are others, but their impact is minimal when running with defaults.

Also, there are other variables that are allocated on each thread (or open connection): read_buffer_size, read_rnd_buffer_size, sort_buffer_size, join_buffer_size and tmp_table_size, and few others. All of them, by default, work very well as allocation is small and efficient. Hence, the main potential issue becomes where we allocate many connections that can hold these buffers for some time and add extra memory pressure. The ideal situation is to control how many connections are being opened (and used) and try to reduce that number to a sufficient number that doesn't hurt the application.

But let's not lose the focus, we have more memory, and we need to know how to tune it properly to make the best usage.

The most memory-impacting setting we need to focus on is innodb_buffer_pool_size, as this is where almost all magic happens and is usually the more significant memory consumer. There is an old rule of thumb that says, "size of this setting should be set around 75% of available memory", and some cloud vendors setup this value to total_memory*0.75.

I said "old" because that rule was good when running instances with 8G or 16G of RAM was common, so allocating roughly 6G out of 8G or 13G out of 16G used to be logical.

But what if we run into an instance with 100G or even 200G? It's not uncommon to see this type of hardware nowadays, so we will use 80G out of 100G or 160G out of 200G? Meaning, will we avoid allocating something

between 20G to 40G of memory and leave that for filesystem cache operations? While these filesystem operations are not useless, I don't see OS needing more than 4G-8G for this purpose on a dedicated DB. Also, it is recommended to use the O_DIRECT flushing method for InnoDB to bypass the filesystem cache.

Example

Now that we understand the primary variables allocating memory let's check a good use case I'm currently working on. Assuming this system:

```
$ free -m
      total      used      free    shared    buff/cache    available
Mem: 385625    307295    40921         4         37408        74865
```

So roughly 380G of RAM, a nice amount of memory. Now let's check what is the maximum potential allocation considering max used connections.

**A little disclaimer here, while this query is not entirely accurate and thus it can diverge from real results, we can have a sense of what is potentially going to be allocated, and we can take advantage of performance_schema database, but this may require enabling some instruments disabled by default:*

```
mysql > show global status like 'max_used_connections';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| Max_used_connections |    67 |
+----------------------+-------+
1 row in set (0.00 sec)
```

So with a maximum of 67 connections used, we can get:

```
mysql > SELECT ( @@key_buffer_size
-> + @@innodb_buffer_pool_size
-> + 67 * (@@read_buffer_size
-> + @@read_rnd_buffer_size
-> + @@sort_buffer_size
-> + @@join_buffer_size
-> + @@tmp_table_size )) / (1024*1024*1024) AS MAX_MEMORY_GB;
+---------------+
| MAX_MEMORY_GB |
+---------------+
| 316.4434      |
+---------------+
1 row in set (0.00 sec)
```

So far, so good, we are within memory ranges, now let's see how big the innodb_buffer_pool_size is and if it is well sized:

```
mysql > SELECT (@@innodb_buffer_pool_size) / (1024*1024*1024) AS BUFFER_POOL_SIZE;
+------------------+
| BUFFER_POOL_SIZE |
+------------------+
| 310.0000         |
+------------------+
1 row in set (0.01 sec)
```

So the buffer pool is 310G, roughly 82% of total memory, and total usage so far was around 84% which leaves us around 60G of memory not being used. Well, being used by filesystem cache, which, in the end, is not used by InnoDB.

Ok now, let's get to the point, how to properly configure memory to be used effectively by MySQL. From pt-mysql-summary we know that the buffer pool is fully filled:

Buffer Pool Size | 310.0G
Buffer Pool Fill | 100%

Does this mean we need more memory? Maybe, so let's check how many disk operations we have in an instance we know with a working dataset that doesn't fit in memory (the very reason why we increased memory size) using with this command:

```
mysqladmin -r -i 1 -c 60 extended-status | egrep "Innodb_buffer_pool_read_requests|Innodb_buffer_
pool_reads"
| Innodb_buffer_pool_read_requests | 99857480858|
| Innodb_buffer_pool_reads         | 598600690  |
| Innodb_buffer_pool_read_requests | 274985     |
| Innodb_buffer_pool_reads         | 1602       |
| Innodb_buffer_pool_read_requests | 267139     |
| Innodb_buffer_pool_reads         | 1562       |
| Innodb_buffer_pool_read_requests | 270779     |
| Innodb_buffer_pool_reads         | 1731       |
| Innodb_buffer_pool_read_requests | 287594     |
| Innodb_buffer_pool_reads         | 1567       |
| Innodb_buffer_pool_read_requests | 282786     |
| Innodb_buffer_pool_reads         | 1754       |
```

Innodb_buffer_pool_read_requests: page reads satisfied from memory (good)
Innodb_buffer_pool_reads: page reads from disk (bad)

As you may notice, we still get some reads from the disk, and we want to avoid them, so let's increase the buffer pool size to 340G (90% of total memory) and check again:

```
mysqladmin -r -i 1 -c 60 extended-status | egrep "Innodb_buffer_pool_read_requests|Innodb_buffer_
pool_reads"
| Innodb_buffer_pool_read_requests | 99937722883 |
| Innodb_buffer_pool_reads         | 599056712   |
| Innodb_buffer_pool_read_requests | 293642      |
| Innodb_buffer_pool_reads         | 1           |
| Innodb_buffer_pool_read_requests | 296248      |
| Innodb_buffer_pool_reads         | 0           |
| Innodb_buffer_pool_read_requests | 294409      |
| Innodb_buffer_pool_reads         | 0           |
| Innodb_buffer_pool_read_requests | 296394      |
| Innodb_buffer_pool_reads         | 6           |
| Innodb_buffer_pool_read_requests | 303379      |
| Innodb_buffer_pool_reads         | 0           |
```

Now we are barely going to disk, and IO pressure was released; this makes us happy - right?

Summary

If you increase the memory size of a server, you mostly need to focus on innodb_buffer_pool_size, as this is the most critical variable to tune. Allocating 90% to 95% of total available memory on big systems is not bad at all, as OS requires only a few GB to run correctly, and a few more for memory swap should be enough to run without problems.

Also, check your maximum connections required (and used,) as this is a common mistake causing memory issues, and if you need to run with 1000 connections opened, then allocating 90% of the memory of the buffer pool may not be possible, and some additional actions may be required (i.e., adding a proxy layer or a connection pool).

From MySQL 8, we have a new variable called innodb_dedicated_server, which will auto-calculate the memory allocation. While this variable is really useful for an initial approach, it may under-allocate some memory in systems with more than 4G of RAM as it sets the buffer pool size = (detected server memory * 0.75), so in a 200G server, we have only 150 for the buffer pool.

Conclusion

Vertical scaling is the easiest and fastest way to improve performance, and it is also cheaper - but not magical. Tuning variables properly requires analysis and understanding of how memory is being used. This article focused on the essential variables to consider when tuning memory allocation, specifically innodb_buffer_pool_size and max_connections. Don't over-tune when it's not necessary and be cautious of how these two affect your systems.

Setting up Resource Limits on Users in MySQL

By Smit Arora

Smit is a MySQL DBA I in Managed Services and has been working at Percona since August 2022. He resides in Delhi and has been working in MySQL-related technologies since 2018.

Often while managing and creating new users, we use all the default options and tend not to use extra features provided by MySQL. These extra options could prevent a user from using all the resources and degrading the performance of MySQL. In this article, we will discuss a few such features that will put resource restrictions on users.

max_user_connections

Sometimes, due to unprecedented growth or huge transactions, a single user makes too many connections, and the MySQL server gets starved of free connections. This blocks the DBA from logging into MySQL to fix it. To fix it in MySQL 5.7 and below, we have to do a restart. For MySQL 8, it can be done without a restart, but we have to configure the admin interface beforehand. You can read more about it here in Dealing With "Too Many Connections" Error in MySQL 8.

But it can be prevented earlier if we restrict the expected user to a certain number of concurrent connections. It can be done at the creation of the user, or we can execute an alter command to modify this configuration.

```
CREATE USER 'monitor_user'@'%' IDENTIFIED BY 'YourPasswordHere' WITH  MAX_USER_
CONNECTIONS 100;
ALTER USER 'monitor_user'@'%'  WITH  MAX_USER_CONNECTIONS 100;
```

There is also an option to restrict all users, and not just individual user accounts, by setting up the variable max_user_connections. It is a dynamic variable, it can be set up globally, and it can also be changed by making the change in the configuration file.

```
mysql> SET GLOBAL max_user_connections=100;
Query OK, 0 rows affected (0.00 sec)
[mysqld]
max_user_connections =100;
```

In this case, this will restrict all users to only 100 concurrent logins. If the user tries to log in after it has made more than 100 concurrent logins, an error message will appear.

```
[root@centos]# mysql -u 'monitor_user' -p
ERROR 1226 (42000): User 'monitor_user' has exceeded the 'max_user_connections' resource (current
value: 100)
```

max_connections_per_hour

Instead of setting up max_user_connections for users, we can also restrict the limit to per hour. This will add a time-based restriction to the user, only allowing a specific number of connections per hour.

```
mysql> ALTER USER 'john_smith'@'%' WITH MAX_CONNECTIONS_PER_HOUR 10;
Query OK, 0 rows affected (0.00 sec)
```

The user will get an error message telling them the cause and the current value of max_connections_per_hour:

```
[root@centos]# mysql -u 'john_smith' -p
ERROR 1226 (42000): User 'john_smith' has exceeded the 'max_connections_per_hour' resource
(current value: 10)
```

max_queries_per_hour

This option will set a limit for any kind of queries that a user can run per hour and will only allow the user to run a pre-defined number of queries per hour.

```
mysql> ALTER USER 'john_smith'@'%' WITH MAX_QUERIES_PER_HOUR 10;
```

Query OK, 0 rows affected (0.00 sec)

The queries that are restricted are not just DML operations but for any kind of queries.

```
mysql> use app_schema
Database changed

mysql> show tables;
ERROR 1226 (42000): User 'john_smith' has exceeded the 'max_questions' resource (current value:
10)
```

max_updates_per_hour

If the user tries to run more than the defined updates per hour, the following error message will appear.

```
mysql> UPDATE some_table SET col1='Value d' WHERE id=2;
ERROR 1226 (42000): User 'john_smith' has exceeded the 'max_updates' resource (current value: 5)
```

If the user tries to run more than the defined updates per hour, the following error message will appear.

```
mysql> UPDATE some_table SET col1='Value d' WHERE id=2;
ERROR 1226 (42000): User 'john_smith' has exceeded the 'max_updates' resource (current value: 5)
```

Conclusion

These options must be used with a lot of caution and thought, as you can block important users from the database that could hamper day-to-day operations. MAX_USER_CONNECTIONS can be used for any monitoring user to avoid a pile-up of connections. It will act as a failsafe to not starve the database off any connections. MAX_UPDATES_PER_HOUR, MAX_CONNECTIONS_PER_HOUR, and MAX_QUERIES_PER_HOUR are useful if you want to set a resource-based limit for individual users. The common use case for these could be where the number of queries per hour is dependent on the price paid by the user. It is not recommended to use these settings for app-based connections. If you are facing such issues from app users, it is recommended to use ProxySQL as a Multiplexing intermediary.

If you don't want to restrict users from running queries or logging in but still want to manage resource limit, you can configure resource groups that set CPU affinity and thread priority for a thread. You can read more about it in MySQL 8.0 RESOURCE_GROUP Overview and use them to fine-tune the load at MySQL 8: Load Fine Tuning With Resource Groups.

Setting up a resource limit is an overlooked configuration in resource management; it is fast, easy, and simple to use. However, these settings must be used very carefully and with proper planning, as any mistake in setting up the limits could result in users not being able to use the database.

Understanding MySQL Memory Usage with Performance Schema

By Peter Zaitsev

Understanding how MySQL uses memory is key to tuning it for optimal performance as well as troubleshooting cases of unexpected memory usage, i.e. when you have MySQL Server using a lot more than you would expect based on your configuration settings.

Early in MySQL history, understanding memory usage details was hard and included a lot of guesswork. Is it possible that some queries running require a large temporary table or allocated a lot of memory for stored user variables? Are any stored procedures taking an unexpectedly high amount of memory? All could be reasons for excessive MySQL memory usage, but you would not easily see if that is just the case.

All that changed with MySQL 5.7, which added memory instrumentation in Performance Schema, and with MySQL 8.0, this instrumentation is enabled by default, so you can get this data from pretty much any running instance.

If you're looking for current memory usage details, Sys schema provides fantastic views:

```
mysql> select event_name,current_alloc from sys.memory_global_by_current_bytes limit 10;
+-----------------------------------------------------------------+---------------+
| event_name                                                      | current_alloc |
+-----------------------------------------------------------------+---------------+
| memory/innodb/buf_buf_pool                                      | 262.12 MiB    |
| memory/temptable/physical_ram                                   | 64.00 MiB     |
| memory/performance_schema/events_statements_summary_by_digest   | 39.67 MiB     |
| memory/sql/TABLE                                                | 33.32 MiB     |
| memory/innodb/ut0link_buf                                       | 24.00 MiB     |
| memory/innodb/lock0lock                                         | 20.51 MiB     |
| memory/innodb/memory                                            | 17.79 MiB     |
| memory/innodb/buf0dblwr                                         | 17.08 MiB     |
| memory/innodb/ut0new                                            | 16.08 MiB     |
| memory/performance_schema/events_statements_history_long        | 13.89 MiB     |
+-----------------------------------------------------------------+---------------+
10 rows in set (0.01 sec)
```

This view shows the current memory allocated overall. You can also drill down further, by looking at memory allocated by connections coming from different hosts:

```
mysql> select host,current_allocated from memory_by_host_by_current_bytes;
+-------------------------------+-------------------+
| host                          | current_allocated |
+-------------------------------+-------------------+
| localhost                     | 1.19 GiB          |
| background                    | 101.28 MiB        |
| li1317-164.members.linode.com | 49.61 MiB         |
| li1319-234.members.linode.com | 27.90 MiB         |
| li1316-24.members.linode.com  | 27.00 MiB         |
+-------------------------------+-------------------+
5 rows in set (0.02 sec)
```

Or even check allocation by given thread_id which can be super helpful to diagnose memory-heavy transactions or queries:

```
mysql> select thread_id,user,current_allocated from memory_by_thread_by_current_bytes limit 5;
+-----------+-------------------------------------+-------------------+
| thread_id | user                                | current_allocated |
+-----------+-------------------------------------+-------------------+
|        44 | innodb/srv_worker_thread            | 1.99 MiB          |
|        48 | innodb/srv_worker_thread            | 1.16 MiB          |
|     54322 | root@localhost                      | 1.10 MiB          |
|        46 | innodb/srv_worker_thread            | 777.29 KiB        |
|     43881 | app1@li1317-164.members.linode.com  | 274.84 KiB        |
+-----------+-------------------------------------+-------------------+
5 rows in set (0.43 sec)
```

Looking at current stats, though, is not very helpful to diagnose past incidents or even to answer the question of why a particular object taking so much memory is normal or excessive. For this, you better have history captured and available for trending… exactly what Percona Monitoring and Management is designed for.

Unfortunately, as of PMM 2.11, we do not have Performance Schema Memory Instrumentation included in the release. It is, however, quite easy to get it added using Custom Queries.

MySQL custom queries

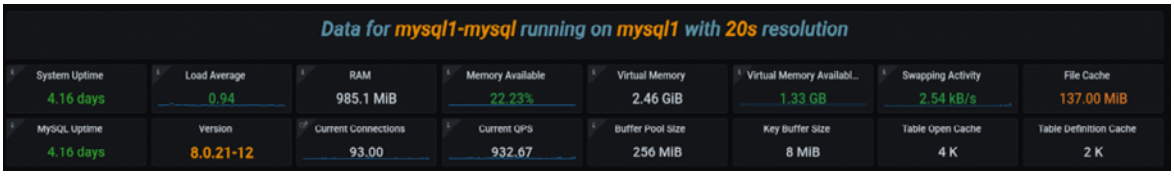
Custom queries is a great feature that allows you to get stats from a local MySQL instance using standard SQL queries and make them available together with other metrics collected by MySQL exporter. This can be used for Performance Schema tables, Information Schema tables, or even queries on your own schema to expose data you consider relevant for your application. Check out Custom Queries In Percona Monitoring and Management for more details.

You can install the custom queries to read memory use statistics from MySQL Performance Schema this way...

```
mysql> select thread_id,user,current_allocated from memory_by_thread_by_current_bytes limit 5;
cd /usr/local/percona/pmm2/collectors/custom-queries/mysql/medium-resolution
wget https://raw.githubusercontent.com/Percona-Lab/pmm-custom-queries/master/mysql/ps-memory-summary.yml
```

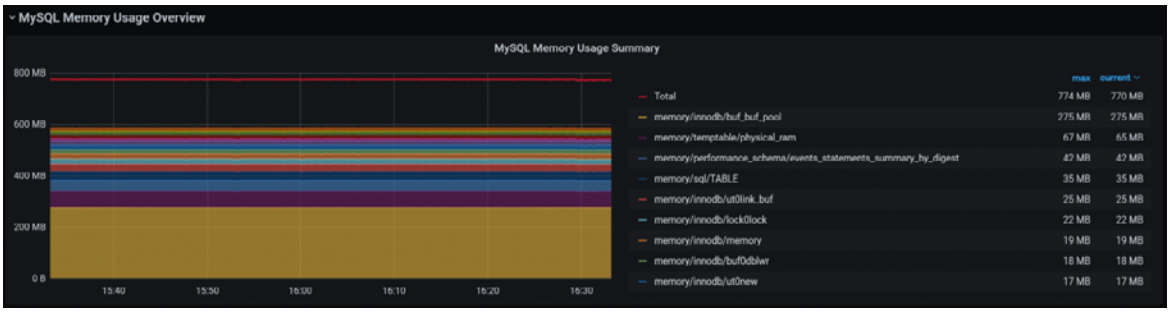
when you install the MySQL Memory Usage Details dashboard from Grafana.com. Let's see what you get by having this dashboard installed.

First, we have the block which does not have much to do with the information we capture from Performance Schema but provides some helpful background in interpreting such information:



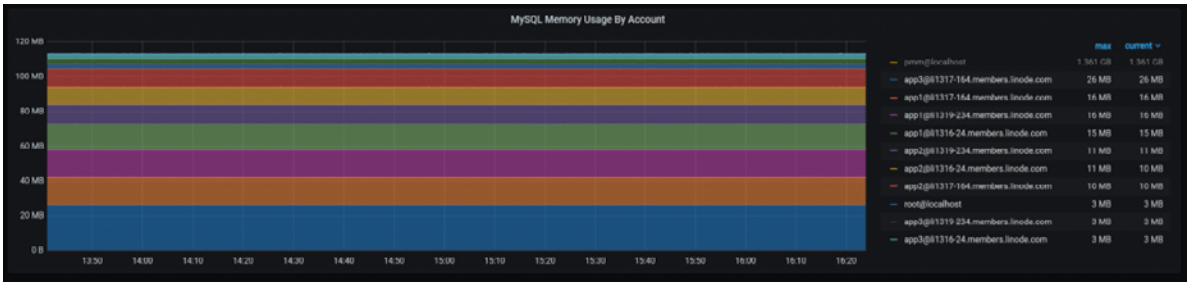
We can see how long the system and MySQL process have been up, how much memory the system has and how much is available, whenever there is any aggressive swapping activity going on, etc., and all of those help us to know whenever MySQL is currently using too much memory... or too little. We can show the number of current connections, which is a common driver of high memory utilization as well as a couple of variables that can impact memory consumption.

The MySQL Memory Usage Summary shows how much memory different objects contain, as well as what the total instrumented memory allocated by the MySQL process is.



With hundreds of different places where memory can be allocated, even memory consumers which are not at the top can add up to a lot:

When we have memory allocation by Host and by User, these only cover memory which can be attributed to specific connections but is a great tool to see if there are particular hosts, users, or applications that are significant memory consumers.



You can also see memory usage details by specific accounts (a given user coming from a given host). This is as far as we go with dashboards, as specific Connection IDs (and Thread IDs) tend to change so quickly that they are not particularly helpful for long term capture.

You may also have noticed I unselected the pmm@localhost user, as it looks like this user workload triggers some bug in memory accounting and unrealistic numbers get reported (this is also something you need to keep in mind, to not trust the data you see blindly).

Finally, you can see memory allocation by different subsystems, which can be rather helpful for advanced analyses, where just the global top consumers do not provide enough information.

What do you think? Is it helpful, or would you like to see some additional visualizations added before we consider including this feature in Percona Monitoring and Management?

MySQL Capacity Planning

By Peter Sylvester

Peter is one of the Senior MySQL Database Administrators within Percona's Managed Services team and has been with Percona since October of 2021.

As businesses grow and develop, the requirements that they have for their data platform grow along with it. As such, one of the more common questions I get from my clients is whether or not their system will be able to endure an anticipated load increase. Or worse yet, sometimes I get questions about regaining normal operations after a traffic increase caused performance destabilization.

As the subject of this article suggests, this all comes down to proper capacity planning. Unfortunately, this topic is more of an art than a science, given that there is really no foolproof algorithm or approach that can tell you exactly where you might hit a bottleneck with server performance. But we can discuss common bottlenecks, how to assess them, and have a better understanding as to why proactive monitoring is so important when it comes to responding to traffic growth.

Hardware considerations

The first thing we have to consider here is the resources that the underlying host provides to the database. Let's take a look at each common resource. In each case, I'll explain why a 2x increase in traffic doesn't necessarily mean you'll have a 2x increase in resource consumption.

Memory

Memory is one of the easier resources to predict and forecast and one of the few places where an algorithm might help you, but for this, we need to know a bit about how MySQL uses memory.

MySQL has two main memory consumers. Global caches like the InnoDB buffer pool and MyISAM key cache and session-level caches like the sort buffer, join buffer, random read buffer, etc.

Global memory caches are static in size as they are defined solely by the configuration of the database itself. What this means is that if you have a buffer pool set to 64Gb, having an increase in traffic isn't going to make this any bigger or smaller. What changes is how session-level caches are allocated, which may result in larger memory consumption.

A tool that was popular at one time for calculating memory consumption was mysqlcalculator.com. Using this tool, you could enter in your values for your global and session variables and the number of max connections, and it would return the amount of memory that MySQL would consume. In practice, this calculation doesn't really work, and that's due to the fact that caches like the sort buffer and join buffer aren't allocated when a new connection is made; they are only allocated when a query is run and only if MySQL determines that one or more of the session caches will be needed for that query. So idle connections don't use much memory at all, and active connections may not use much more if they don't require any of the session-level caches to complete their query.

The way I get around this is to estimate the amount of memory consumed on average by sessions as such… ({Total memory consumed by MySQL} - {sum of all global caches}) / {average number of active sessions}

Keep in mind that even this isn't going to be super accurate, but at least it gives you an idea of what common session-level memory usage looks like. If you can figure out what the average memory consumption is per active session, then you can forecast what 2x the number of active sessions will consume.

This sounds simple enough, but in reality, there could be more to consider. Does your traffic increase come with updated code changes that change the queries? Do these queries use more caches? Will your increase in traffic mean more data, and if so, will you need to grow your global cache to ensure more data fits into it?

With the points above under consideration, we know that we can generally predict what MySQL will do with memory under a traffic increase, but there may be changes that could be unforeseen that could change the amount of memory that sessions use.

The solution is proactive monitoring using time-lapse metrics monitoring like what you would get with Percona Monitoring and Management. Keep an eye on your active session graph and your memory consumption graph and see how they relate to one another. Checking this frequently can help you get a better understanding of how session memory allocation changes over time and will give you a better understanding of what you might need as traffic increases.

CPU

When it comes to CPU, there's obviously a large number of factors that contribute to usage. The most common is the queries that you run against MySQL itself. However, having a 2x increase in traffic may not lead to a 2x increase in CPU as, like memory, it really depends on the queries that are run against the database. In fact, the most common cause of massive CPU increase that I've seen isn't traffic increase; it's code changes that introduced inefficient revisions to existing queries or new queries. As such, a 0% increase

in traffic can result in full CPU saturation.

This is where proactive monitoring comes into play again. Keep an eye on CPU graphs as traffic increases. In addition, you can collect full query profiles on a regular basis and run them through tools like pt-query-digest or look at the Query Analyzer (QAN) in PMM to keep track of query performance, noting where queries may be less performant than they once were, or when new queries have unexpected high load.

Disk space

A 2x increase in traffic doesn't mean a 2x increase in disk space consumption. It may increase the rate at which disk space is accumulated, but that also depends on how much of the traffic increase is write-focused. If you have a 4x increase in reads and a 1.05X increase in writes, then you don't need to be overly concerned about disk space consumption rates.

Once again, we look at proactive monitoring to help us. Using time-lapse metrics monitoring, we can monitor overall disk consumption and the rate at which consumption occurs and then predict how much time we have left before we run out of space.

Disk IOPS

The amount of disk IOPS your system uses will be somewhat related to how much of your data can fit into memory. Keep in mind that the disk will still need to be used for background operations as well, including writing to the InnoDB redo log, persisting/checkpointing data changes to table spaces from the redo log, etc. But, for example, if you have a large traffic increase that's read-dependent and all of the data being read in the buffer pool, you may not see much of an IOPS increase at all.

Guess what we should do in this case? If you said "proactive monitoring," you get a gold star. Keep an eye out for metrics related to IOPS and disk utilization as traffic increases.

Before we move on to the next section, consider the differences in abnormal between disk space and disk IOPS. When you saturate disk IOPS, your system is going to run slow. If you fill up your disk, your database will start throwing errors and may stop working completely. It's important to understand the difference so you know how to act based on the situation at hand.

Database engine considerations

While resource utilization/saturation are very common bottlenecks for database performance, there are limitations within the engine itself. Row-locking contention is a good example, and you can keep an eye on row-lock wait time metrics in tools like PMM. But, much like any other software that allows for concurrent session usage, there are mutexes/semaphores in the code that are used to limit the number of sessions that can access shared resources. Information about this can be found in the semaphores section in the output of the "SHOW ENGINE INNODB STATUS" command.

Unfortunately, this is the single hardest bottleneck to predict and is based solely on the use case. I've seen systems running 25,000+ queries per second with no issue, and I've also seen systems running ~5,000 queries per second that ran into issues with mutex contention.

Keeping an eye on metrics for OS context switching will help with this a little bit, but unfortunately this is a situation where you normally don't know where the wall is until you run right into it. Adjusting variables like innodb_thread_concurrency can help with this in a pinch, but when you get to this point, you really need to look at query efficiency and horizontal scaling strategies.

Another thing to consider is configurable hard limits like max_connections, where you can limit the upper bound of the number of connections that can connect to MySQL at any given time. Keep in mind that increasing this value can impact memory consumption as more connections will use more memory, so use caution when adjusting upward.

Conclusion

Capacity planning is not something you do once a year or more as part of a general exercise. It's not something you do when management calls you to let you know a big sale is coming up that will increase the load on the hosts. It's part of a regular day-to-day activity for anyone that's operating in a database administrator role.

Proactive monitoring plays a big part in capacity planning. I'm not talking about alert-based monitoring that hits your pager when it's already too late, but evaluating metrics usage on a regular basis to see what the data platform is doing, how it's handling its current traffic, etc. In most cases, you don't see massive increases in traffic all at once; typically, it's gradual enough that you can monitor as it increases and adjust your system or processes to avoid saturation.

Tools like PMM and the Percona Toolkit play a big role in proactive monitoring and are open source for free usage. So if you don't have tools like this in place, this comes in at a price point that makes tool integration easier for your consideration.

InnoDB configuration

The InnoDB storage engine is at the heart of MySQL's high performance and durability, making its configuration a critical aspect of database optimization. Properly tuning InnoDB can improve your database's responsiveness and throughput, particularly in situations with high read and write activities. This chapter focuses on some of the foundational principles and strategies necessary to maximize the potential of InnoDB.

Our experts offer several tips on the basics of performance optimization all the way to InnoDB flushing in write-intensive workloads, and you'll gain an understanding of how to configure InnoDB for optimal performance, ensuring your MySQL databases are efficient and capable of meeting your needs.

InnoDB Performance Optimization Basics

By Rituja Borse

Rituja joined Percona in 2019 and is a MySQL DBA II on the Percona Managed Services team, focused on improving MySQL database performance.

Although there have been many articles about adjusting MySQL variables for better performance, I think this topic deserves anupdate since the last update was a decade ago, and MySQL 5.7 and 8.0 have been released since then with some major changes.

These guidelines work well for a wide range of applications, though the optimal settings, of course, depend on the workload.

Hardware

Memory

The amount of RAM to be provisioned for database servers can vary greatly depending on the size of the database and the specific requirements of the company. Some servers may need a few GBs of RAM, while others may need hundreds of GBs or even terabytes of RAM. Factors that can affect the amount of RAM needed by a database server include the total size of the database, the number of concurrent users, and the complexity of the database queries. As datasets continue to grow in size, the amount of RAM required to store and process these datasets also increases. By caching hot datasets, indexes, and ongoing changes, InnoDB can provide faster response times and utilize disk IO in a much more optimal way.
CPU

From a CPU standpoint, faster processors with many cores provide better throughput. CPUs with 32/64 cores are still common, and we see some large clients with 96 cores, and the latest MySQL versions can utilize them much better than before. However, it is worth noting that simply adding more CPU cores does not always result in improved performance. CPU core usage will also depend on the specific workload of the application, such as the number of concurrent users or the complexity of the queries being run.

Storage

The type of storage and disk used for database servers can have a significant impact on performance and reliability. Nowadays, solid-state drives (SSDs) or non-volatile memory express (NVMe) drives are preferred over traditional hard disk drives (HDDs) for database servers due to their faster read and write speeds, lower latency, and improved reliability. While NVMe or SSDs are generally more expensive than HDDs, the increased performance and reliability that they offer make them a cost-effective choice for database servers that require fast access to data and minimal downtime. RAID 10 is still the recommended level for most workloads, but make sure your RAID controller can utilize the SSD drive's performance and will not become the actual bottleneck.

Operating system

Linux is the most common operating system for high-performance MySQL servers. Make sure to use modern filesystems, like EXT4, XFS, or ZFS on Linux, combined with the most recent kernel. Each of them has its own limits and advantages: for example, XFS is fast in deleting large files, while EXT4 can provide better performance on fast SSD drives, and ZFS on Linux has progressed a lot. Benchmark before you decide.

For database servers, we usually recommend our clients have:

- Jemalloc installed and enabled for MySQL.
- Transparent huge pages (THP) disabled.
- Setting swappiness to one is generally recommended, lowering the tendency of swapping.
- Setting oom_score_adj to -800.

Cloud

Different cloud providers offer a range of instance types and sizes, each with varying amounts of CPU, memory, and storage. Some cloud providers also offer specialized instances for database workloads, which may provide additional features and optimizations for performance and scalability. One of the benefits of cloud-based database servers is the ability to scale resources up or down as needed. It's important to consider the potential need for scaling and select an instance type and size to accommodate future growth. Some cloud providers also offer auto-scaling features that can automatically adjust the number of instances based on workload demand.

MySQL InnoDB settings

(Dynamic) - Does not require MySQL restart for change.

(Static) - Requires MySQL restart for change.

innodb_buffer_pool_size (Dynamic) - InnoDB relies heavily on the buffer pool and should be set correctly. Typically a good value is 70%-80% of available memory. Also, refer to innodb_buffer_pool_chunk_size mentioned below.

innodb_buffer_pool_instances (Static) - Enabling this is useful in highly concurrent workloads as it may reduce contention of the global mutexes. The optimal value can be decided after testing multiple settings, starting from eight is a good choice.

innodb_buffer_pool_chunk_size (Static) - Defines the chunk size by which the buffer pool is enlarged or reduced. This variable is not dynamic, and if it is incorrectly configured, it could lead to undesired situations. Refer to InnoDB Buffer Pool Resizing: Chunk Change for more details on configuration.

innodb_log_file_size (Static) - Large enough InnoDB transaction logs are crucial for good, stable write performance. But also larger log files mean that the recovery process will be slower in case of a crash. However, this variable has been deprecated since 8.0.30. Refer to innodb_redo_log_capacity below.

innodb_redo_log_capacity (Dynamic) - Introduced in 8.0.30, this defines the amount of disk space occupied by redo log files. This variable supersedes the innodb_log_files_in_group and innodb_log_file_size variables. When this setting is defined, the innodb_log_files_in_group and innodb_log_file_size settings are ignored (those two variables are now deprecated since 8.0.30).

innodb_log_buffer_size (Dynamic) - InnoDB writes changed data records into its log buffer, which is kept in memory, and it saves disk I/O for large transactions as it does not need to write the log of changes to disk before transaction commit. If you have transactions that update, insert, or delete many rows, making the log buffer larger saves disk I/O.

innodb_flush_log_at_trx_commit (Dynamic) - The default value of '1' gives the most durability (ACID compliance) at a cost of increased filesystem writes/syncs. Setting the value to '0' or '2' will give more performance but less durability. At a minimum, transactions are flushed once per second.

innodb_thread_concurrency (Dynamic) - With improvements to the InnoDB engine, it is recommended to allow the engine to control the concurrency by keeping it to the default value (which is zero). If you see concurrency issues, you can tune this variable. A recommended value is two times the number of CPUs plus the number of disks.

innodb_flush_method (Static) - Setting this to O_DIRECT will avoid a performance penalty from double buffering; this means InnoDB bypasses the operating system's file cache and writes data directly to disk (reducing the number of I/O operations required).

innodb_online_alter_log_max_size (Dynamic) - The upper limit in bytes on the size of the temporary log files used during online DDL operations for InnoDB tables. If a temporary log file exceeds the upper size limit, the ALTER TABLE operation fails, and all uncommitted concurrent DML operations are rolled back. Thus, a large value for this option allows more DML to happen during an online DDL operation but also extends the period of time at the end of the DDL operation when the table is locked to apply the data from the log.

innodb_numa_interleave (Static) - For 'NUMA enabled systems' with large amounts of memory (i.e.,> 128GB), we recommend turning on NUMA interleaving. Enabling this parameter configures memory allocation to be 'interleaved' across the various CPU-Memory channels. This helps "even out" memory allocations so that one CPU does not become a memory bottleneck.

innodb_buffer_pool_dump_at_shutdown/innodb_buffer_pool_load_at_startup (Dynamic/Static respectively) - These variables allow you to dump the contents of the InnoDB buffer pool to disk at shutdown and load it back at startup, which will pre-warm the buffer pool so that you don't have to start

with a cold buffer pool after a restart.

innodb_buffer_pool_dump_pct (Dynamic) - The option defines the percentage of most recently used buffer pool pages to dump. By default, MySQL only saves 25% of the most actively accessed pages, which should be reasonable for most use cases, it can then be loaded faster than if you try to load every page in the buffer pool (100%), many of which might not be necessary for a general workload. You can increase this percentage if needed for your use case.

Innodb_io_capacity (Dynamic) - It defines the number of I/O operations per second (IOPS) available to InnoDB background tasks, such as flushing pages from the buffer pool and merging data from the change buffer. Ideally, keep the setting as low as practical but not so low that background activities fall behind. Refer to this for more information on configuration.

Innodb_io_capacity_max (Dynamic) - If the flushing activity falls behind, InnoDB can flush more aggressively, at a higher rate than innodb_io_capacity. innodb_io_capacity_max defines the maximum number of IOPS performed by InnoDB background tasks in such situations. Refer to Give Love to Your SSDs – Reduce innodb_io_capacity_max! for more information on configuration.

innodb_autoinc_lock_mode (Static) - Setting the value to '2' (interleaved mode) can remove the need for an auto-inc lock (at the table level) and can increase performance when using multi-row insert statements to insert values into a table with an auto-increment primary key. Note that this requires either ROW or MIXED binlog format. (The default setting is 2 as of MySQL 8.0)

innodb_temp_data_file_path (Static) - Defines the relative path, name, size, and attributes of InnoDB temporary tablespace data files. If you do not specify a value for innodb_temp_data_file_path, the default behavior is to create a single, auto-extending data file named ibtmp1 in the MySQL data directory. For 5.7, it is recommended to set a max value to avoid the risk of datadir partition filling up due to a heavy or bad query. 8.0 introduced session temporary tablespaces, temporary tables, or the internal optimizer tables no longer use 'ibtmp1'.

innodb_stats_on_metadata (Dynamic) - The default setting of "OFF" avoids unnecessary updating of InnoDB statistics and can greatly improve read speeds.

innodb_page_cleaners (Static) - InnoDB supports multiple page cleaner threads for flushing dirty pages from buffer pool instances. If your workload is write-IO bound when flushing dirty pages from buffer pool instances to data files, and if your system hardware has available capacity, increasing the number of page cleaner threads may help improve write-IO throughput.

innodb_deadlock_detect (Dynamic) - This option can be used to disable deadlock detection. On high-concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock.

Application tuning for InnoDB

Make sure your application is prepared to handle deadlocks that may happen. Review your table structure and see how you can take advantage of InnoDB properties – clustering by primary key, having a primary key in all indexes (so keep primary key short), and fast lookups by primary keys (try to use it in joins).

Conclusion

There are many other options you may want to tune, but here we've covered the important InnoDB parameters, OS-related tweaking, and hardware for optimal MySQL server performance. I hope this helps!

Tuning MySQL/InnoDB Flushing for a Write-Intensive Workload

By Yves Trudeau and Francisco Bordenave

Yves is a Principal Architect in Professional Services at Percona, specializing in distributed technologies such as MySQL Cluster, Pacemaker, and Percona XtraDB cluster. He was previously a senior consultant for MySQL and Sun Microsystems. He holds a Ph.D. in Experimental Physics.

Understanding the tuning process is very important since we don't want to make things worse or burn our SSDs. We proceed with one section per variable or closely-related variables. The variables are also grouped in sections based on the version of MySQL or Percona Server for MySQL where they are valid. A given variable may be present more than once if its meaning changes between versions. There may be other variables affecting the way InnoDB handles a write-intensive workload. Hopefully, the most important ones are covered.

MySQL Community Before 8.0.19

innodb_io_capacity

The default value for the innodb_io_capacity variable is 200 and is the number of iops used for idle flushing and tasks like applying the change buffer operations in the background. The adaptive flushing is independent of innodb_io_capacity. There are very few reasons to increase innodb_io_capacity above the default value:

- Reduce the change buffer lag
- Increase the Idle flushing rate (when LSN is constant)
- Increase the dirty pages percentage flushing rate

We have a hard time finding any reasons to increase the idle flushing rate. Also, the dirty pages percentage flushing shouldn't be used (see below). That leaves the changer buffer lag. If you constantly have a large number of entries in the change buffer when you do "show engine innodb status\G", like here:

```
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1229, free list len 263690, seg size 263692, 40050265 merges
merged operations:
 insert 98911787, delete mark 40026593, delete 4624943
discarded operations:
 insert 11, delete mark 0, delete 0
```

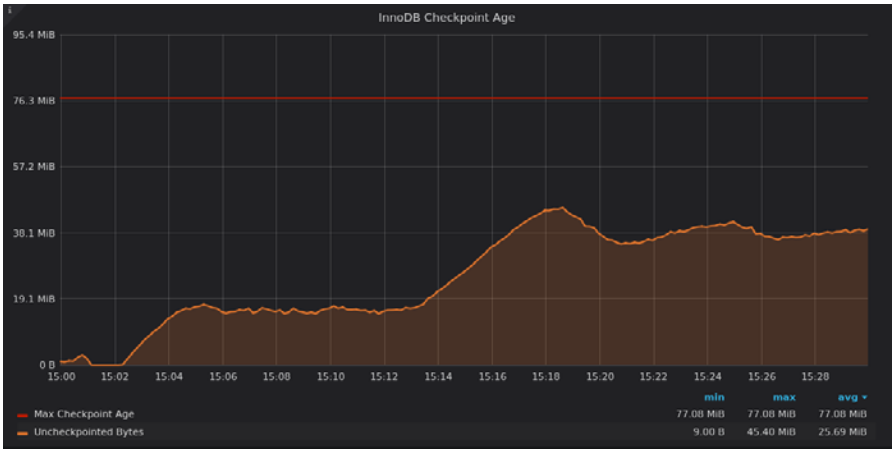
where there are 1229 unapplied changes, you could consider raising innodb_io_capacity.

innodb_io_capacity_max

The default value for innodb_io_capacity_max is 2000 when innodb_io_capacity is also at its default value. This variable controls how many pages per second InnoDB is allowed to flush. This roughly matches write IOPS generated by InnoDB. Your hardware should be able to perform that many page flushes per second. If you see messages like:

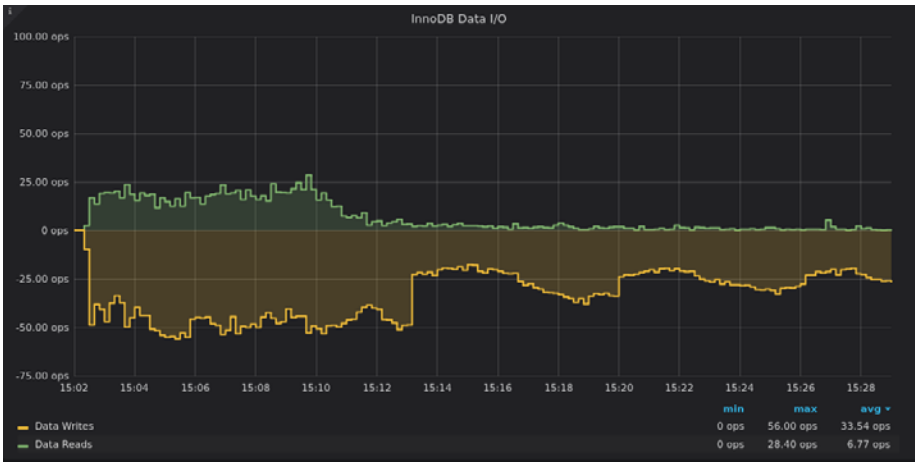
[Note] InnoDB: page_cleaner: 1000ms intended loop took 4460ms. The settings might not be optimal. (flushed=140, during the time.)

In the MySQL error log, then the hardware is not able to keep up, so you should lower the innodb_io_capacity_max value. The ideal value should allow the checkpoint age to be as high as possible while staying under 75% of the max checkpoint age. The higher the checkpoint age, the less writes the database will need to perform. At the same time, if the checkpoint age is above 75%, you'll risk stalls. A trade-off is required, write performance vs risk of short stalls. Choose where you stand according to the performance pain points you are facing and the risk of short stalls. If you have deployed Percona Monitoring and Management, the "InnoDB Checkpoint Age" in the "MySQL InnoDB Details" dashboard is very useful. Here is a simple example produced with the help of sysbench:



InnoDB checkpoint age[/caption]

The load starts at 15:02 and the checkpoint age quickly levels around 20%. Then, at 15:13, innodb_io_capacity_max is lowered, the checkpoint age raises to around 50%. This raise caused the flushing rate to fall by about half, as it can be seen in another PMM graph, "InnoDB Data I/O"



PMM InnoDB IO graph[/caption]

Percona Server for MySQL also exposes all the relevant variables:

```
mysql> show global status like 'innodb_ch%';
+---------------------------+------------+
| Variable_name             | Value      |
+---------------------------+------------+
| Innodb_checkpoint_age     |  400343763 |
| Innodb_checkpoint_max_age | 1738750649 |
+---------------------------+------------+
2 rows in set (0.00 sec)

mysql> show global variables like 'innodb_adaptive_fl%';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| innodb_adaptive_flushing     | ON    |
| innodb_adaptive_flushing_lwm | 10    |
+------------------------------+-------+
2 rows in set (0.00 sec)
```

Using the above example, the checkpoint age is roughly 23% of the max age, well above the 10% defined in innodb_adaptive_flushing_lwm. If the checkpoint age value is typical of when the database is under load and it never raises to much higher values, innodb_io_capacity_max could be lowered a bit. On the

contrary, if the checkpoint age is often close or even above 75%, the value innodb_io_capacity_max is too small.

innodb_max_dirty_pages_pct

innodb_max_dirty_pages_pct_lwm

These variables control the old MySQL 5.0 era flushing as a function of the percentage of dirty pages in the buffer pool. It should be disabled by setting innodb_max_dirty_pages_pct_lwm to 0. It is the default value in 5.7 but strangely, not in 8.0.

innodb_page_cleaners

innodb_page_cleaners sets a number of threads dedicated to scan the buffer pool instances for dirty pages and flush them. Furthermore, you cannot have more cleaner threads than buffer pool instances. The default value for this variable is 4.

With 4 cleaner threads, InnoDB is able to flush at a very high rate. Actually, unless you are using Percona Server for MySQL with the parallel doublewrite buffers feature, very likely the doublewrite buffer will bottleneck before the cleaner threads.

innodb_purge_threads

innodb_purge_threads defines the number of threads dedicated to purge operations. The purge operations remove deleted rows and clear undo entries from the history list. These threads also handle the truncation of the undo space on disk. The default number of purge threads is 4.

It is difficult to imagine a workload able to saturate 4 purge threads, the default is generous. Most of the work done by the purge threads is in memory. Unless you have a very large buffer pool and a high write load, the default value of 4 is sufficient. If you see high values for the history length in the “show engine innodb status” without any long-running transaction, you can consider increasing the number of purge threads.

innodb_read_io_threads

innodb_write_io_threads

Surprisingly enough, on Linux with asynchronous IO, these threads have little relevance. For example, one of our customers operates a server with 16 read io threads and 16 write io threads. With an uptime of about 35 days, the mysqld process consumed 104 hours of CPU, read 281 TB, and wrote 48 TB to disk. The dataset is composed essentially of compressed InnoDB tables. You could think that for such a server the IO threads would be busy right? Let's see:

```
root@prod:~# ps -To pid,tid,stime,time -p $(pidof mysqld)
   PID TID STIME     TIME
 72734  72734 Mar07 00:23:07
 72734  72752 Mar07 00:00:00 -> ibuf (unused)
 72734  77733 Mar07 00:01:16
 72734  77734 Mar07 00:01:51
 72734  77735 Mar07 00:05:46 -> read_io
 72734  77736 Mar07 00:06:04 -> read_io
 72734  77737 Mar07 00:06:06 -> read_io
 … 11 lines removed
 72734  77749 Mar07 00:06:04 -> read_io
 72734  77750 Mar07 00:06:04 -> read_io
 72734  77751 Mar07 00:35:36 -> write_io
 72734  77752 Mar07 00:34:02 -> write_io
 72734  77753 Mar07 00:35:14 -> write_io
 … 11 lines removed
 72734  77766 Mar07 00:35:18 -> write_io
 72734  77767 Mar07 00:34:15 -> write_io
```

We added annotations to help identify the IO thread and a number of lines were removed. The read IO threads have an average CPU time of 6 minutes while the write IO threads have a slightly higher one, at 35 minutes. Those are very small numbers, clearly, 16 is too many in both cases. With the default values of 4, the CPU time would be around 25 minutes for the read IO threads and around 2 hours for the write IO threads. Over the uptime of 34 days, these times represent way less than 1% of the uptime.

innodb_lru_scan_depth

innodb_lru_scan_depth is a very poorly named variable. A better name would be innodb_free_page_target_per_buffer_pool. It is a number of pages InnoDB tries to keep free in each buffer pool instance to speed up read and page creation operations. The default value of the variable is 1024.

The danger with this variable is that with high value and many buffer pools instances, you may end up wasting a lot of memory. With 64 buffer pool instances, the default value represents 1GB of memory. The best way to set this variable is to look at the output of "show engine innodb statusG" with a pager grepping for "Free buffers". Here's an example.

```
mysql> pager grep 'Free buffers'
PAGER set to 'grep 'Free buffers''
mysql> show engine innodb statusG
Free buffers            8187
Free buffers            1024
Free buffers            1023
Free buffers            1022
Free buffers            1020
Free buffers            1024
Free buffers            1025
Free buffers            1025
Free buffers            1024
1 row in set (0.00 sec)
```

The first line is the total for all the buffer pools. As we can see, InnoDB has no difficulty maintaining around 1024 free pages per buffer pool instance. The lowest observed value across all the instances is 1020. As a rule of thumb, you could use the current variable value (here 1024), minus the smallest observed value over a decent number of samples, and then multiply the result by 4. If 1020 was the lowest value, we would have (1024-1020)*4 ~ 16. The smallest allowed value for the variable is 128. We calculated 16, a value smaller than 128, so 128 should be used.

innodb_flush_sync

innodb_flush_sync allows InnoDB flushing to ignore the innodb_io_capacity_max value when the checkpoint age is above 94% of its maximal value. Normally this is correct, hence the default value of 'ON'. The only exception would be a case where the read SLAs are very strict. Allowing InnoDB to use beyond innodb_io_capacity_max IOPs for flushing would compete with the read load and read SLAs would be missed.

innodb_log_file_size and innodb_log_files_in_group

The variables innodb_log_file_size and innodb_log_files_in_group determine the size of the redo log ring buffer. Essentially, the redo log is a journal of recent changes. The largest allowed total size is 512GB minus one byte. A larger redo log ring buffer allows for pages to stay dirty for a longer period of time in the buffer pool. If during that time, pages receive more updates, the write load is essentially deflated. Lowering the write load is good for performance, especially if the database is close to IO-bound. A very large redo log ring buffer causes the recovery time after a crash to take more time but this is much less of an issue these days.

As a rule of thumb, the time to fully use the redo log should be around one hour. Watch the evolution of the innodb_os_log_written status variable over time and adjust accordingly. Keep in mind that one hour is a general rule, your workload may need to be different. This is a very easy tunable and it should be among the first knobs you turn.

innodb_adaptive_flushing_lwm

innodb_adaptive_flushing_lwm defines the low watermark to activate the adaptive flushing. The low watermark value is expressed as a percentage of the checkpoint age over the maximal checkpoint age. When the ratio is under the value defined by the variable, the adaptive flushing is disabled. The default value is 10 and the max allowed value is 70.

The main benefits of a higher watermark are similar to the ones of using a larger value for innodb_log_file_size but it puts the database closer to the edge of max checkpoint age. With a high low watermark value, the write performance will be better on average but there could be short stalls. For a production server, increasing innodb_log_file_size is preferred over increasing innodb_adaptive_flushing_lwm. However, temporarily, it can be useful to raise the value dynamically to speed up a logical dump restore or to allow a slave to catch up with its master.

innodb_flushing_avg_loops

This variable controls the reactivity of the adaptive flushing algorithm, it damps its reaction. It affects the calculation of the average rate of consumption of the redo log. By default, InnoDB calculates the average every 30 seconds and the resulting value is further averaged with the previous one. That means if there is a sudden transaction surge, the adaptive flushing algorithm will react slowly, only the checkpoint age-related part will react immediately. It is difficult to identify a valid use case where someone would need to modify this variable.

MySQL Community after 8.0.19

innodb_io_capacity

Prior to MySQL 8.0.19, when InnoDB needed to flush furiously at the sync point, it proceeded in large batches of pages. The issue with this approach is that the flushing order may not be optimal, too many pages could come from the same buffer pool instances.

Since MySQL 8.0.19, the furious flushing is done in chunks of innodb_io_capacity size. There is no wait time between the chunks so there is no relationship with the IO capacity. This way helps better balance the flushing order and lowers the potential stall time.

innodb_idle_flush_pct

With MySQL 8.0.19, the idle flushing rate is no longer directly set by innodb_io_capacity, it is multiplied by the percentage represented by innodb_idle_flush_pct. InnoDB uses the remaining IO capacity for the insert buffer thread. When a database becomes idle, meaning the LSN value does not move, InnoDB normally flushes the dirty pages first and then applies the pending secondary indexes changes stored in the change buffer. If this is not the behavior you want, lower the value to give some IO to the change buffer.
Percona Server for MySQL 5.7.x and 8.0.x

innodb_cleaner_lsn_age_factor

Percona Server for MySQL has a different default adaptive flushing algorithm, high_checkpoint, which essentially allows more dirty pages. This behavior is controlled by the variable innodb_cleaner_lsn_age_factor. You can restore the default MySQL algorithm by setting the variable to legacy. Since you should aim to have as many dirty pages as possible without running into flush storms and stalls, the high_checkpoint algorithm will help you. If you have extreme bursts of checkpoint age, maybe the legacy algorithm would fare better.

innodb_empty_free_list_algorithm

This variable controls the behavior of InnoDB when it struggles at finding free pages in a buffer pool instance. It has two possible values: legacy and backoff. It is only meaningful if you often have “Free buffers” close to 0 in “show engine innodb status\G”. In such a case, the LRU manager thread may hammer the free list mutex and cause contention with other threads. When set to backoff, the thread will sleep for some time after a failure to find free pages to lower the contention. The default is backoff and normally it should be fine in most cases.

Conclusion

There are other variables impacting InnoDB writes, but these are among the most important ones. As a general rule, don’t over-tune your database server and try to change one setting at a time.
This is our understanding of the InnoDB variables affecting the flushing. The findings are backed by the pretty unique exposure we have at Percona to a wide variety of use cases. We also spend many hours reading the code to understand the inner moving parts of InnoDB. As usual, we are open to comments but if possible, try to back any argument against our views with either a reference to the code or a reproducible use case.

InnoDB File Growth Weirdness

By Yves Trudeau

There is a common pattern in life, you often discover or understand things by accident. Many scientific discoveries fit such a description. In our database world, I was looking to see how BLOB/TEXT columns are allocated using overlay pages and I stumbled upon something interesting and unexpected. Let me present to you my findings, along with my attempt at explaining what is happening.

InnoDB tablespaces

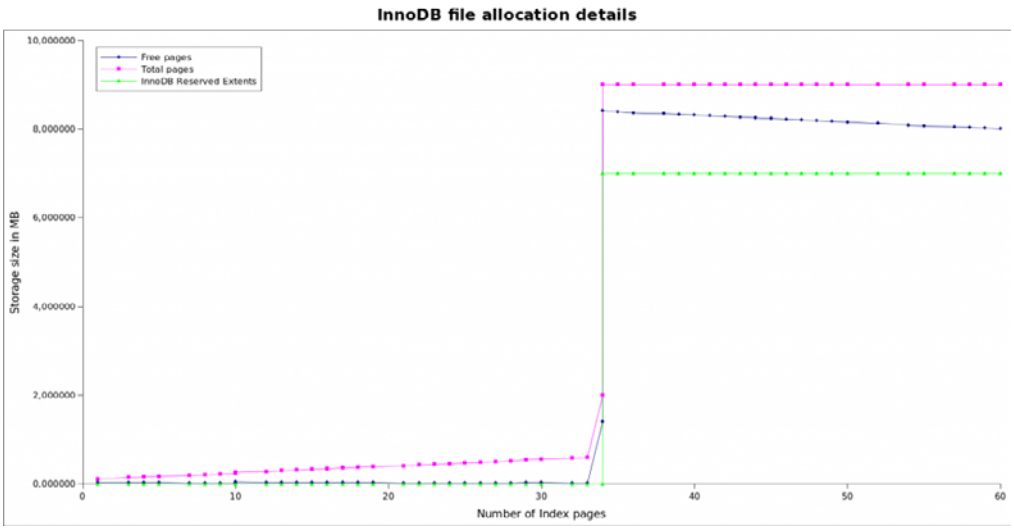
The first oddity I found is a bunch of free pages in each tablespace it is skipping. Here's an example from a simple table with only an integer primary key and a char(32) column:

```
root@LabPS8_1:~/1btr# innodb_space -f /var/lib/mysql/test/t1.ibd space-extents
start_page  page_used_bitmap
0            ################################..........................  <--- free pages
64           ################################################################
128          ################################################################
...
```

The innodb_space tool comes from the InnoDB ruby project of Jeremy Cole. If you want to explore InnoDB file formats, you need these tools. As you can see, the first extent has 27 free pages. These free pages are reserved for node pages (non-leaf) and will eventually be all used. At this point, I thought a table with 34 index pages, just starting to use the second extent for the leaf pages, would have 90 free pages (27 + 63) and use 2MB of disk space. While the previous statement proved to be true, I was up for quite a surprise.

InnoDB file growth

To better illustrate the amount of free space available in an InnoDB data file, I decided to follow the evolution of the tablespace file size as I added index pages. The following figure shows my result.



As I added rows, more leaf pages were allocated until the file segment of the leaf pages reached 32 pages. At this point, the table has 33 index pages, one root, and 32 leaves. The allocation of another page forces InnoDB to fully allocate the first extent and add the second one for the leaves. At this point, the size on the disk is 2MB. If we keep inserting rows, the following page allocation triggers InnoDB to allocate seven reserved extents of 1MB each. At this point, the tablespace size on the disk reaches 9MB.

InnoDB uses the reserved extents for btree maintenance operations. They are not accounted for in the free space of the tablespace. Now, reserving seven extents of 1MB each in a table containing only 560KB of data is pretty insane. At this point, the InnoDB tablespace has a size of 9MB on disk. This is extremely inefficient, about 8.4MB if just free space filled with "0". Of course, as the table grows, the size impact of these reserved extents is diluted. The amount of reserved space will grow by about 1MB (one extent) for every 100MB allocated.

This allocation of reserved extents is far from optimal, especially in a multi-tenants era where it is fairly common to see MySQL servers handling more than 100k tables. 100k tables, each with only 1MB of data in them will use 900GB of disk space. This phenomenon is not new, a bug report was created back in 2013 and is still open. The bug is considered a low priority and non-critical.

A lot of effort has been devoted to improving the capacity of MySQL 8.0 to handle a large number of tables. Until the allocation of reserved extents is fixed, be aware of this issue when planning your storage allocation. Of course, if you are using ZFS, the impacts are more limited...

```
root@LabPS8_1:~# du -hs /var/lib/mysql/test/t1.ibd
225K /var/lib/mysql/test/t1.ibd
root@LabPS8_1:~# du -hs --apparent-size /var/lib/mysql/test/t1.ibd
9,0M /var/lib/mysql/test/t1.ibd
```

My lab setup uses ZFS for LXC and KVM instances. LZ4 compression does magic on extents full of “0”, the actual consumed space is reduced to 225KB.

Data caching

This chapter will cover data caching as a technique to significantly reduce database load, improve query response times, and, ultimately, enhance the overall performance of applications relying on MySQL databases. By leveraging data caching, repetitive queries can fetch data from a fast-access memory store rather than the database itself, thus minimizing disk I/O operations and network latency.

Here, Percona experts offer up a few recommended strategies for approaching data caching optimization, from ensuring cache efficiency to the impact of querying table information from information_schema. It aims to equip you with the knowledge to implement and manage caching solutions for your MySQL-based applications effectively.

MySQL Data Caching Efficiency

By Fernando Laudares Camargos

Fernando is a Senior Architect in Professional Services at Percona, having joined Percona in early 2013 after working eight years for a Canadian company specialized in offering services based in open source technologies. Fernando's work experience includes the architecture, deployment and maintenance of IT infrastructures based on Linux, open source software and a layer of server virtualization.

A shared characteristic in most (if not all) databases, be them traditional relational databases like Oracle, MySQL, and PostgreSQL or some kind of NoSQL-style database like MongoDB, is the use of a caching mechanism to keep (a copy of) part of the data in memory.

The reasoning behind it is very simple: accessing data from memory remains many times faster than retrieving data from disk. And if a thousand users are trying to access a given bit of data simultaneously, having to fetch it from disk for each request would be extremely inefficient.

Note that the caching structure doesn't have to be as big as your dataset to work well. In most cases, if it is big enough to store the "hot" part of the dataset, or the data most often accessed, that's a win. Considering the example from above, even if that bit of data being requested by the users is not found in memory and needs to be retrieved from disk, that slow operation will have to be performed only once for that batch of requests.

So, how do you know if your hot data is in memory? How do you know if your MySQL database caching is operating efficiently?

By comparing the amount of data pages (the "bits of data") being served directly from memory to the amount of data pages having to be fetched from disk: the more the database needs to read from disk to complete queries, the less efficient the caching structure is operating. Note that unoptimized queries contribute to this by going over/processing more data than necessary, loading them into the caching structure, and often pushing hot data out of it.

Finding that ratio is not always easy; ideally, the database should expose these metrics. MySQL does. Its main data caching structure for the standard InnoDB storage engine is called Buffer Pool. The two status variables (or status counters in this case) that expose the Buffer Pool efficiency are (quoting the MySQL manual):

- Innodb_buffer_pool_read_requests: The number of logical read requests.
- Innodb_buffer_pool_reads: The number of logical reads that InnoDB could not satisfy from the buffer pool, and had to read directly from disk.

For years, MySQL DBAs have been tracking those metrics on monitoring tools or even capturing them manually during key moments. Two tools in the Percona Toolkit can help with that:

- pt-stalk can be used to capture status variables (among various other important metrics for analyzing server performance). For example:

```
$ pt-stalk --no-stalk --iterations=1 -- --user=root --password=secret
```

- pt-mext can be used to plot a collection of status variables captured by pt-stalk in a way that highlights how the status variables changed over time for a collection of samples. For example:

```
$ pt-mext -r -- cat /var/lib/pt-stalk/2023_03_31_18_50_33-mysqladmin > /tmp/ptmext
```

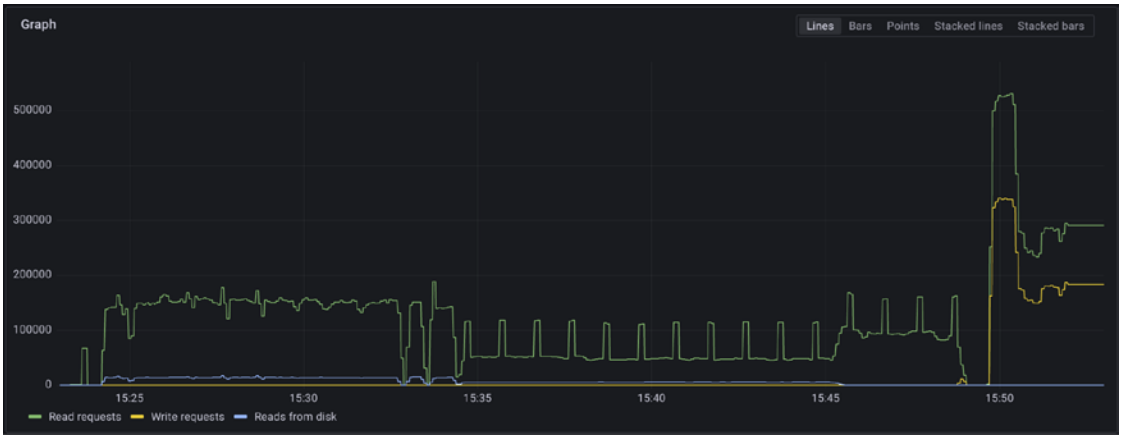
then:

```
$ grep "Innodb_buffer_pool_reads|Innodb_buffer_pool_read_requests" /tmp/ptmext
Innodb_buffer_pool_read_requests 105982819 46157 46299 47181 47637 46932 45713 48839 47792 48064
46446 48498 47994 47074 47367 46578 44069 46793 47815 48299 46636 47323 46392 43919 45229 42383
418932 326100 43851
Innodb_buffer_pool_reads 9370874 5325 5360 5480 5512 5464 5369 5621 5549 5541 5421 5614 5468 5453
5495 5401 5106 5404 5555 5604 5424 5494 5376 5111 5243 4887 4420 4868
```

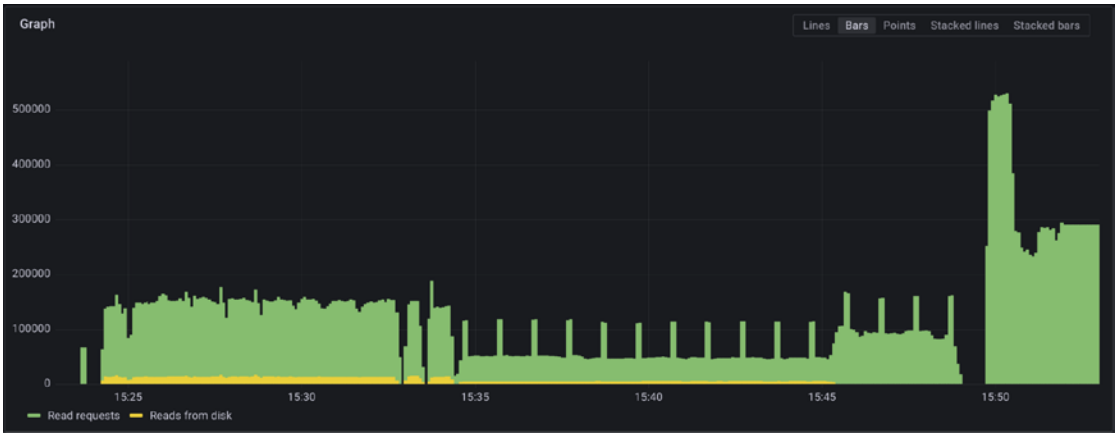
Roughly, it shows an average of 89% of pages read from the Buffer Pool - or 11% of pages being fetched from disk.

Is there an easy way to get that information? There is if you have **Percona Monitoring and Management**. In fact, there used to be a graph to display this on PMM v.1. It has since been removed from there on PMM2, so I created a feature request to bring it back: https://jira.percona.com/browse/PMM-11923

One thing that didn't work well at the time was mixing read requests with write requests and then having a third data point for reads from disk, like follows:



Innodb_buffer_pool_reads is a percentage of Innodb_buffer_pool_read_requests. Thus, it is best represented by a bars-style graph, decoupled from write requests:

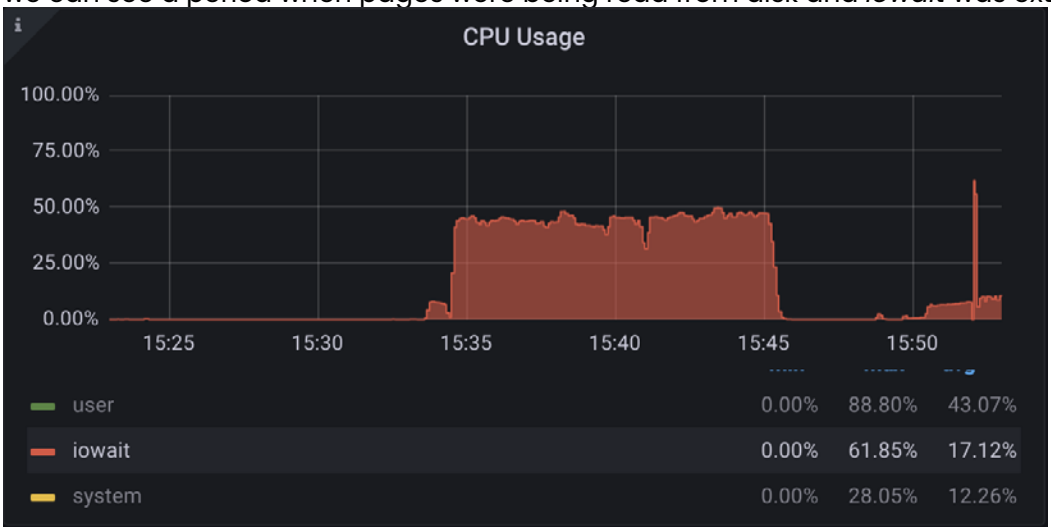


Should I aim for a 100% hit ratio?

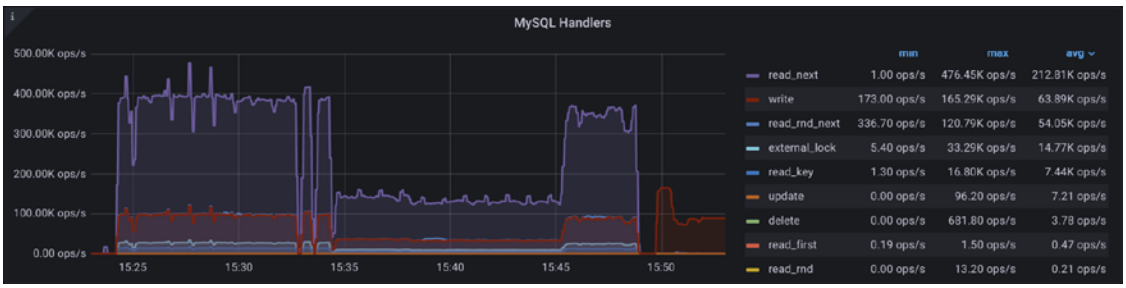
The second reason we had that graph removed from PMM was to avoid passing the wrong impression: that you should be looking at increasing your Buffer Pool size if you were seeing any reads from disk showing up there with the aim of reaching a 100% hit ratio.

Remember, you want the Buffer Pool to be big enough to hold all your hot data - not pages that are loaded due to unoptimized queries. Those need to be traced and fixed, and PMM can help you do that as well.

Comparing the *Reads from disk* from above with the CPU Usage graph from the Node Summary dashboard, we can see a period when pages were being read from disk and *iowait* was extremely high:



We can then check the *MySQL Handlers* graph on the MySQL Instance Summary dashboard in search of possible index scans (*read_next*) and full-table scans (*read_rnd_next*),



and even the MySQL Temporary Objects graph on the same dashboard to search for on-disk temporary tables. But the place where we will find the queries we are looking for, the queries we need to improve, is in the Query Analytics (QAN) dashboard.

Impact of Querying Table Information From information_schema

By Carlos Tutte

Carlos is a computer engineer from Montevideo, Uruguay, who joined Percona on February 2018, first as a support engineer and now a Senior Consultant in Professional Services. Working in complex IT solutions for more than 10 years, Carlos specializes in MySQL technologies.

On MySQL and Percona Server for MySQL, there is a schema called information_schema (I_S) which provides information about database tables, views, indexes, and more.

A lot of useful information can be retrieved from this schema, for example, table metadata and foreign key relations, but trying to query I_S can induce performance degradation if your server is under heavy load, as shown in the following example test.

Disclaimer: This article is meant to show a less-known problem but is not meant to be a serious benchmark. The percentage in degradation will vary depending on many factors {hardware, workload, number of tables, configuration, etc.}.

Test

The test compares a baseline of how the server behaves while "under heavy load but no queries against I_S" vs. " under heavy load + I_S queries" to check for performance degradation.

The same tests have been executed in Percona Server for MySQL 5.7 and 8.0. The queries executed against I_S to check performance degradation checks information about some table FKs (foreign keys) relationships.

Setup

The setup consists of creating 10K tables with sysbench and adding 20 FKs to 20 tables.

Hardware

Number of CPUs: 12
Memory size: 12288 MB

Main Percona Server for MySQL configuration variables tuned in my.cnf:

```
Open-files-limit = 65535
Table-definition-cache = 4096
Table-open-cache = 2000
Table_open_cache_instances = 1
Innodb-buffer-pool-size = 10G
Innodb_buffer_pool_instances = 2
```

You can review the full my.cnf file here: https://github.com/ctutte/blog_IS/blob/master/my.cnf

Executed queries

To generate the database setup, I used sysbench with the following flags:

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=123456Ab!
    --mysql-db=test --range_size=100
    --table_size=2250 --tables=10000 --threads=12 --events=0 --time=120
    --rand-type=uniform /usr/share/sysbench/oltp_read_only.lua prepare
Then manually created 20 FKs:
ALTER TABLE sbtest1 ADD FOREIGN KEY (id) REFERENCES sbtest8001 (id);
ALTER TABLE sbtest2 ADD FOREIGN KEY (id) REFERENCES sbtest8002 (id);
…
```

{repeat 20 times}

After the scenario was set up, I executed three runs of sysbench with the following query:

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=123456Ab!
    --mysql-db=test --range_size=100
    --table_size=2250 --tables=1000 --threads=10 --events=0 --time=60
    --rand-type=uniform /usr/share/sysbench/oltp_read_write.lua --report-interval=1 run
```

To query against I_S, I used the following bash command line, which checks FK in a while loop from bash:

```
while true; do { mysql -u root -p123456Ab! -e "SELECT TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME,
REFERENCED_TABLE_NAME,
REFERENCED_COLUMN_NAME FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_SCHEMA IS
NOT NULL"; }; done
```

Considering that the server has 12 vCPUs and sysbench is using 10, then there is some spare CPU capacity for running the queries against I_S, right? Let's see the results.

Results for Percona Server for MySQL 5.7

Three runs of 60 seconds and 1000 tables:

	Total queries	per second
Without FK check	8337600	46320
With FK checks	6924900	38472

It can be seen that only running sysbench, 46320 queries per second, could be executed on the server.

In the scenario where I checked sysbench + I_S queries to check FKs continuously, 38472 queries per second were executed. That is (46320 - 38472) * 100 / 46320 = 16.9% performance degradation.

If the I_S queries were CPU bound, then I would expect the server to be able to do more than 46320 QPS, but something is going on, and actually, the amount of QPS did go down.

The reason for this performance degradation is that querying against I_S will need to open tables frm files (limited by table_open_cache variable).

In my example test, when running only sysbench, the workload shows few Table_open_cache_misses (the first column is initial value, each successive value is delta increase per second).

Table_open_cache_misses while NOT checking for FKs:

```
58437 28  15 13 7 5 7 6 1 0 0 1 0 3 8 10 5 0 0 0 0 0 0 1 0 5 0 6 5 7 5 4 0 2 2 4 6 1 0 0 0 0 0 0
0 0 0 0 0
```

Whereas when running sysbench + I_S queries, there is a greater number of Table_open_cache_misses due to MySQL/Percona Server for MySQL 5.7 having to open each table.frm (and in which my test runs, I have purposely read a very high number of tables compared to "Table-open-cache" variable).

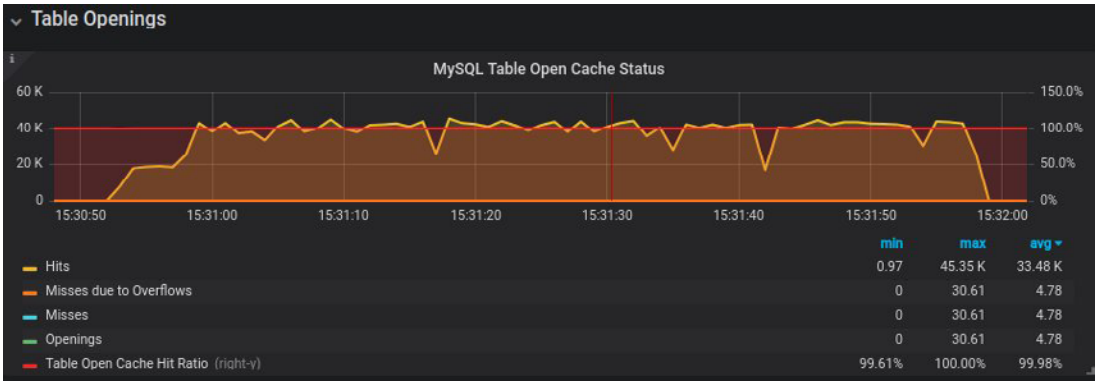
Table_open_cache_misses while checking for FKs:

```
68133 30   31 33 30 32 31 31 29 32 28 32 30 3230 33 31 30 31 28 31 31 33 32 33 29 33 34 32 31 33
30 28 34 33 30 29 29 29 25 30 31 29 33 27 29 29 28 28
```

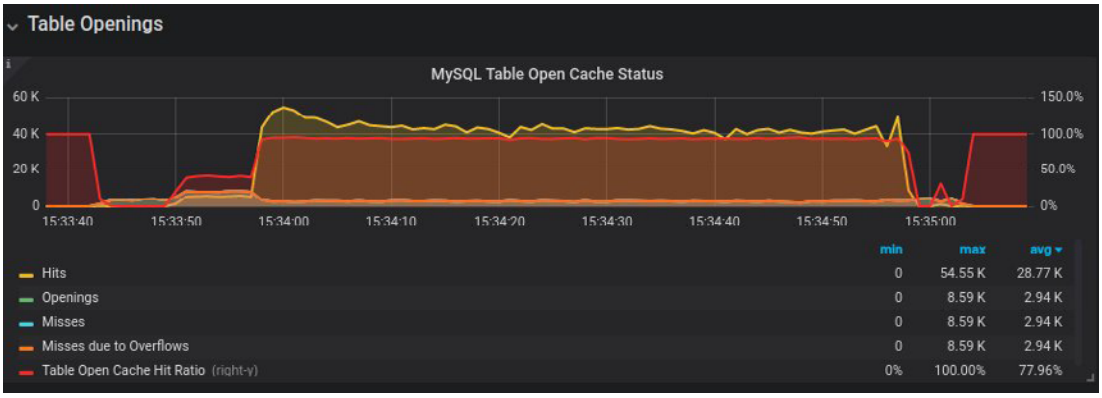
The above outputs were generated using pt-stalk + pt-mext from Percona Toolkit.

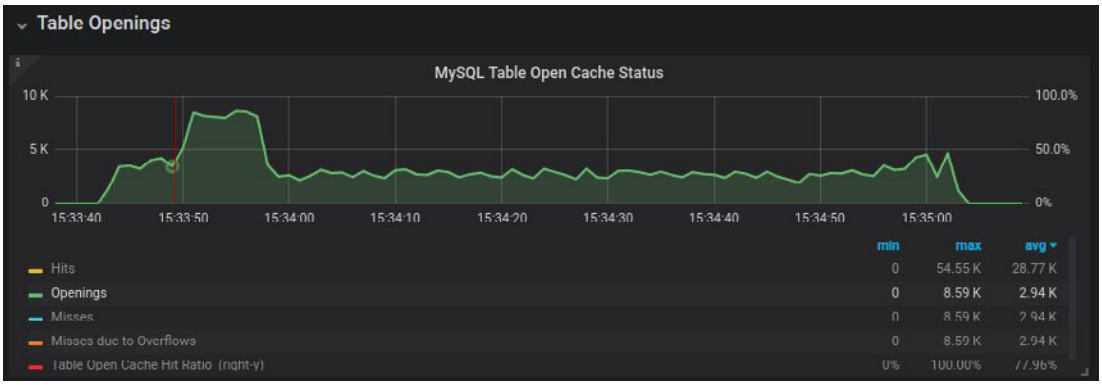
This can also be seen using Percona Monitoring and Management (PMM) and checking the "MySQL overview" dashboard ->"MySQL table open cache status" graphic.

When running only sysbench, there is a high number of cache hits (~99.98%) and a low number of "misses," which means the table_cache is big enough for the workload.



Whereas while running sysbench + I_S queries, there is a decrease in the number of cache hits at ~15:33:40 when FK checks executed until ~15:35:00, which also shows an increase in (table) "Openings" and "Misses" for the duration of the test.





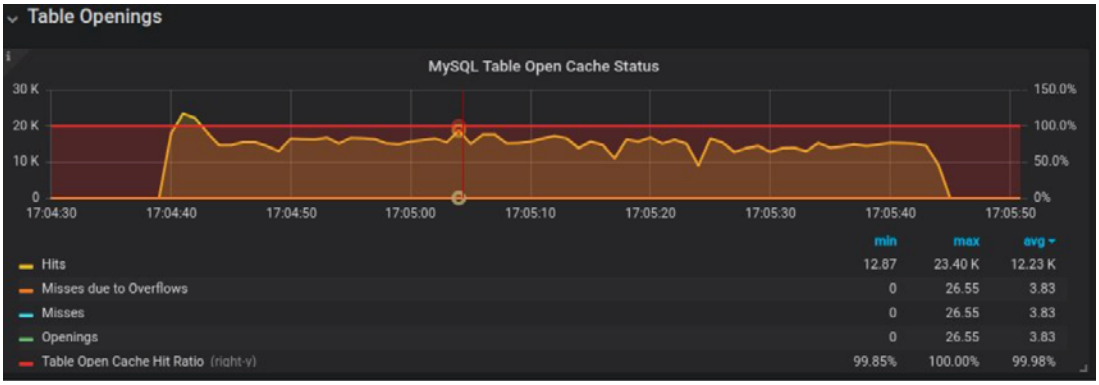
Performance degradation when querying against I_S can be minimized if:

- table_open_cache was increased
- using a "WHERE" condition on certain columns to avoid a datadir/database folder scan. You can read some optimization examples in the official doc: https://dev.mysql.com/doc/refman/5.7/en/information-schema-optimization.html .

Results for Percona Server for MySQL 8.0

For MySQL and Percona Server for MySQL 8, executing sysbench + I_S queries shows there almost no cache misses, as can be seen in the following snapshot:

MySQL 8 improved data dictionary access, which avoids having to open all tables .frm files. This improvement has been ported to Percona Server for MySQL 8.



Conclusion

In conclusion, checking table information from 5.7 I_S relies on opening .frm files from disk to retrieve table information, which can cause a performance drop when the amount of opened tables does not fit in the table_cache.

If you rely heavily on queries against information_schema, then your workload will benefit from upgrading to MySQL/Percona Server for MySQL 8 because of the new data dictionary format. While you are on 5.7, you can also consider increasing table_open_cache to avoid table_misses or at least use some filters on the “WHERE” clause to avoid opening all .frm files and limit the query to a subset of the tables for a limited performance impact.

Partitioning

Partitioning allows for dividing a database into smaller, more manageable pieces, thereby improving query performance, simplifying maintenance tasks, and facilitating faster data access. This chapter, compiled from some recommended strategies of our experts, not only includes an ultimate guide to partitioning but also showcases how partitioning can be applied to archive data seamlessly and efficiently.

Far from a one-stop shop for all things MySQL partitioning, it is a great place to start for those looking to implement partitioning in their databases to optimize performance, ensure scalability, and streamline data archival processes.

The Ultimate Guide to MySQL Partitions

By Manjot Singh

Manjot worked as a Database & Infrastructure Architect at Percona from 2016 to 2018. He is a veteran of startup and Fortune 50 enterprise companies alike with a few years spent in government, education, and hospital IT.

So, what is MySQL partitioning?

Partitioning is a way in which a database (MySQL in this case) splits its actual data down into separate tables but still gets treated as a single table by the SQL layer.

When partitioning in MySQL, it's a good idea to find a natural partition key. You want to ensure that table lookups go to the correct partition or group of partitions. This means that all SELECT, UPDATE, and DELETE should include that column in the WHERE clause. Otherwise, the storage engine does a scatter-gather and queries ALL partitions in a UNION that is not concurrent.

Generally, you must add the partition key into the primary key along with the auto-increment, i.e., PRIMARY KEY (part_id, id). If you don't have well-designed and small columns for this composite primary key, it could enlarge all of your secondary indexes.

What are the different types of MySQL partitions?

Horizontal and vertical MySQL partitions are techniques used to divide tables into smaller sections for improved performance and management. Horizontal partitioning splits a table into smaller tables with identical columns but distinct rows, while Vertical partitioning divides a table into separate tables with related columns. Various partition types, like RANGE, LIST, HASH, and KEY, are used for specific needs, from range-based data to custom criteria, to ensure efficient data handling and the optimization of queries.

RANGE partitioning

RANGE partitioning in MySQL is a data partitioning technique where a large table is divided into smaller partitions based on a specified range of column values like dates or numeric intervals. Each partition holds data that falls within a specific range, optimizing data handling and query speed.

HASH partitioning

HASH partitioning in MySQL divides a table into partitions based on the hash value of a designated column's contents. Unlike range or list partitioning, where you manually designate the partition for specific column values, hash partitioning automatically assigns values to partitions based on hashing. This method distributes data evenly across partitions to achieve balanced storage and optimal query performance.

LIST partitioning

LIST partitioning in MySQL shares similarities with range partitioning. As with range partitioning, each partition is explicitly defined, but in list partitioning, partitions are created and assigned based on including a column value in predefined value lists rather than the contiguous ranges of values used in range partitioning.

COLUMNS partitioning

COLUMNS partitioning in MySQL is a technique that involves dividing a table into partitions based on specific columns' values. Unlike other partitioning methods focusing on the entire row, column partitioning separates columns into different partitions. This approach is helpful when working with tables with many columns or when specific columns are frequently updated.

KEY partitioning

KEY partitioning is similar to HASH partitioning, except that only one or more columns to be evaluated are specified, and the MySQL server provides its own hashing function. These columns can contain other than integer values since the hashing function supplied by MySQL guarantees an integer result regardless of the column data type.

MySQL partitioning in Version 5.7

MySQL version 5.7 introduced various features and capabilities for partitioning, enhancing its utility in managing large datasets. It enabled dividing large tables into smaller, manageable segments based on defined criteria. This facilitates improved data organization, query optimization, and maintenance.

In version 5.7, MySQL partitioning supports multiple partitioning types, including RANGE, LIST, HASH, KEY, and COLUMNS. Each type caters to different data distribution needs.

Using partitioning in a MySQL 5.7 environment offers several practical benefits. It significantly improves query performance by reducing the amount of data scanned during queries, which is especially helpful

when dealing with large tables. Partition pruning, a feature in MySQL 5.7, ensures that only relevant partitions are accessed, further enhancing query efficiency. Additionally, partitioning aids in maintenance tasks like archiving and purging old data, as operations can be performed on individual partitions instead of the entire table.

MySQL partitioning in MySQL 8.0

MySQL 8.0 brought substantial advancements and enhancements to partitioning, significantly elevating its capabilities. This version introduces key features and optimizations that address limitations from previous iterations.

One major enhancement is the support for subpartitioning. MySQL 8.0 allows you to create subpartitions within existing partitions, providing an additional level of data segmentation. This feature facilitates even more precise data organization and management, allowing for complex use cases involving multi-level data hierarchies.

Additionally, MySQL 8.0 introduces automated list partitioning, simplifying partition creation through by enabling the database to determine the partition based on the values inserted automatically.
This version also notably integrates native backing for range and list partitioning of spatial indexes, amplifying geospatial query speed for substantial datasets. Enhancements to the query optimizer improve partition pruning for both single-level and subpartitioned tables, leading to improved query performance.

To sum it up, MySQL 8.0 significantly advances partitioning with features like subpartitioning, automatic list partitioning, and improved query optimization. These enhancements address limitations from previous versions, allowing for more complex data organization, streamlined management, and optimized query performance.

What are the benefits of MySQL partitions?

MySQL partitioning offers several advantages in terms of query performance and maintenance:

Enhanced query performance: Partitioning improves query performance by minimizing the amount of data scanned during queries. As the data is distributed into smaller partitions, the database engine only needs to scan relevant partitions, leading to faster query responses.

Optimized resource utilization: Partitioning enables parallelism in query execution across partitions. This means that multiple partitions can be processed simultaneously, making better use of available hardware resources and further enhancing query performance.

Data retention and deletion: Partitioning simplifies the archiving or deleting of old data by targeting specific partitions, and enhancing data retention policies.

Reduced overhead: Partitioning can significantly reduce the overhead of managing large tables. For example, when inserting or deleting data, the database engine only needs to modify the relevant partitions, which can be much faster than performing these operations on the entire table.

Streamlined maintenance: Partitioning simplifies maintenance operations. For example, you can perform maintenance tasks like index rebuilds, statistics updates, or data archiving on specific partitions rather than the entire table, minimizing downtime and optimizing resource utilization.

Data lifecycle management: Partitioning supports efficient data lifecycle management. Old or infrequently accessed data can be stored in separate partitions or even archived, allowing for better control over data retention and optimization of storage resources.

Enhanced scalability: Partitioning enhances the database's ability to scale, as data can be distributed across different storage devices.

In summary, MySQL partitioning brings substantial advantages to both query performance and maintenance. It improves data retrieval speed, enhances resource utilization, streamlines maintenance operations, optimizes storage management, and reduces overheads associated with large tables. These benefits collectively contribute to a more efficient database environment.

What are the challenges and limitations of MySQL partitions?

While there are lots of positives about using MySQL partitioning, there can also be challenges and limitations that users should be aware of:

Query optimization complexity: Although partitioning can enhance query performance, it requires queries to be designed with partitioning key considerations in mind. Inappropriately designed queries may not fully utilize partitioning benefits, leading to poor performance.

Limited key choices: Not all columns are suitable for partitioning keys. Choosing a proper partitioning key is crucial, and inappropriate selections can result in uneven data distribution across partitions, impacting performance.

Suboptimal partitioning strategies: Choosing the wrong partitioning strategy or key can lead to performance degradation. For instance, using partitioning on a table with a small number of rows may not provide significant benefits and can even worsen performance due to increased complexity.

Limited parallelism: While partitioning allows for parallel processing, there might be limitations on how many partitions can be processed concurrently based on hardware resources, potentially impacting query performance.

Data skewing: In some scenarios, data might not be uniformly distributed across partitions, causing "data skew." This can lead to uneven resource utilization and slower performance for certain partitions.

Replication and backup issues: MySQL partitioning might impact the way data replication and backups are performed. Special considerations are needed to ensure these processes still work seamlessly after partitioning.

So, while MySQL partitioning does offer advantages, it also brings challenges and limitations related to complexity, maintenance, query optimization, and performance. Careful planning and continuous monitoring are crucial to facing these challenges and achieving optimal performance.

Performance optimization with MySQL partitioning

MySQL partitioning enhances query performance by enabling the database to focus on relevant data partitions during queries. This reduces the amount of data that needs to be scanned, resulting in faster data retrieval. For example, when querying a large table for specific date ranges, partitioning allows the engine to scan only relevant partitions containing data within those ranges.

Query execution plans are positively impacted by partitioning. The query optimizer recognizes partitioning schemes and generates execution plans that use partition pruning. This means the optimizer can skip unnecessary partitions, resulting in optimized query plans that use fewer resources and execute more quickly.

Partitioning influences indexing strategies by narrowing the scope of indexing. Instead of indexing the entire table, partitioning allows for more focused indexing. This minimizes index size and boosts efficiency, leading to faster query performance.

In scenarios where partitioning aligns with natural data distribution, such as time-series data or geographical regions, query execution time is significantly reduced. Queries that involve specific partitions can bypass irrelevant data; for instance, when searching for transactions within a certain date range, partitioning enables the database to search only the relevant partition.

Best practices for implementing MySQL partitioning

With these best practices, you can ensure that your MySQL partitioning setup is efficient, well-maintained, and improves database performance.

Choose the correct partition key: Select a partition key that aligns with your data distribution and query patterns. Common choices include time-based or range-based values.

Monitor query performance: Continuously monitor query performance after partitioning. Use tools like EXPLAIN to assess query execution plans.

Watch for bloat: Over time, partitions can accumulate large amounts of data, leading to slow queries.

Proper indexing: Partitioned tables benefit from proper indexing. Ensure that the chosen partition key is part of the primary or unique key. Additionally, consider indexing frequently queried columns to improve performance further.

Regular maintenance: Perform routine maintenance tasks, such as purging old data from partitions, optimizing indexes, and rebuilding partitions.

Backup and restore: As we mentioned earlier, partitioning can impact backup and restore strategies. Ensure your backup and restore procedures account for partitioned data to prevent data loss and ensure reliable recovery.

Test, test, and test again: Before implementing partitioning in production, thoroughly test it in a controlled environment. This helps identify potential issues and fine-tune the partitioning strategy.

Documentation: Always be documenting! Be sure to include your partitioning strategy, why certain partition keys are used, and your maintenance procedures.

Talk to experts: If you're new to partitioning or dealing with complex scenarios, consider consulting with experts.

Choosing the right partitioning strategy

Selecting the appropriate partitioning strategy in MySQL involves carefully considering various factors, including:

Understanding your data's nature and distribution. For range-based data, consider range partitioning, while list partitioning is suitable for discrete values. Hash partitioning evenly distributes data.

Analyzing query patterns to align the strategy with how data is accessed. Time-based queries benefit from range partitioning, while hash partitioning suits equally accessed values.

Matching the strategy to your database requirements. For archiving historical data, consider range-based on time. High-write workloads might favor hash or key partitioning for load balancing.

Watching for changes in data patterns. As data grows, a previously effective strategy might become less optimal. Periodically review and adjust.

Any partitioning strategy should improve query performance, not lead to suboptimal queries — test and benchmark strategies before implementation.

Ensuring the strategy aligns with maintenance operations. For example, rebuilding large partitions might often impact uptime. Select a strategy that minimizes disruptions.

Continuously monitoring query performance after implementation. Be ready to adjust your strategy as needed.

Evaluating how your chosen strategy accommodates future growth, as some strategies scale better with larger datasets.

Choosing the right partitioning strategy is pivotal to database performance. By aligning the strategy with your data's characteristics and specific requirements, you ensure that your MySQL partitioning delivers the desired results.

Quick Data Archival in MySQL Using Partitions

By Smit Arora

Space constraint has been an endless and painstaking process for many of us, especially in systems that have a high number of transactions and data growth exceeding hundreds of GBs in a matter of days. In this article, I will share a solution to remove this space and remove rows from tables in a few seconds regardless of the size of a table without causing any additional load on the database using table partitions.

The first approach that comes to anyone's mind for deleting the row is using a DELETE query in SQL. Suppose, one wants to delete rows from a table that are older than one year—the query for such operations would be like this:

```
DELETE FROM salaries WHERE from_date <DATE_SUB(NOW(),INTERVAL 1 YEAR);
```

The above query is pretty straightforward but there are a few caveats:

- Server business will grow exponentially and could impact the usual traffic on the server.
- To speed up the above query we need to create an appropriate index so that the query can be executed in minimal time and have less impact on server performance.
- If we are using binlog_format as ROW, a huge number of bin logs would be created which could choke I/O of servers and require extra cleanup.
- Even after deleting all the rows, space won't be freed. MySQL won't shrink tablespace and storage won't be released to the file system. To release it to the file system, we need to recreate the table by running ANALYZE or an ALTER.

One way to get around this is using the Percona pt-archiver. But the archiver process will take time as it also considers system load, replica lag, and specified parameters to throttle the archiving process without affecting ongoing traffic.

What I propose here is using MySQL partitioning, which is a much faster approach.

What is partitioning?

In MySQL, the InnoDB storage engine has long supported the notion that a tablespace and the MySQL Server, even prior to the introduction of partitioning, could be configured to employ different physical directories for storing different databases. Partitioning takes this notion a step further, allowing users to save portions of the table according to a user-defined rule. The user-selected rule by which data can be divided is known as a partitioning function, which could be a simple rule against a set of ranges or value lists, an internal hashing function, or a linear hashing function.

When we partition the table data-file is split across multiple partitions of smaller data-files. The operation we do against that specific range of data, will not affect the whole table as only one data file is touched.

Table without partition:

```
centos: employees # ls -lh salaries*#*
-rw-r-----. 1 mysql mysql 104G Oct 11 05:47 salaries.ibd
```

Table with partition:

```
centos: employees # ls -lh salaries*#*
-rw-r-----. 1 mysql mysql 9.0G Oct 11 05:47 salaries#P#p01.ibd
-rw-r-----. 1 mysql mysql 9.0G Oct 11 05:47 salaries#P#p02.ibd
-rw-r-----. 1 mysql mysql  10G Oct 11 05:47 salaries#P#p03.ibd
-rw-r-----. 1 mysql mysql  11G Oct 11 05:47 salaries#P#p04.ibd
-rw-r-----. 1 mysql mysql  11G Oct 11 05:47 salaries#P#p05.ibd
-rw-r-----. 1 mysql mysql  12G Oct 11 05:47 salaries#P#p06.ibd
-rw-r-----. 1 mysql mysql  12G Oct 11 05:47 salaries#P#p07.ibd
-rw-r-----. 1 mysql mysql  13G Oct 11 05:47 salaries#P#p08.ibd
-rw-r-----. 1 mysql mysql  14G Oct 11 05:47 salaries#P#p09.ibd
-rw-r-----. 1 mysql mysql  14G Oct 11 05:47 salaries#P#p10.ibd
-rw-r-----. 1 mysql mysql  15G Oct 11 05:47 salaries#P#p11.ibd
-rw-r-----. 1 mysql mysql  15G Oct 11 05:47 salaries#P#p12.ibd
-rw-r-----. 1 mysql mysql  16G Oct 11 05:47 salaries#P#p13.ibd
-rw-r-----. 1 mysql mysql  16G Oct 11 05:47 salaries#P#p14.ibd
-rw-r-----. 1 mysql mysql  17G Oct 11 05:47 salaries#P#p15.ibd
-rw-r-----. 1 mysql mysql  17G Oct 11 05:47 salaries#P#p16.ibd
-rw-r-----. 1 mysql mysql  16G Oct 11 05:47 salaries#P#p17.ibd
-rw-r-----. 1 mysql mysql  13G Oct 11 05:47 salaries#P#p18.ibd
-rw-r-----. 1 mysql mysql  96M Oct 11 05:45 salaries#P#p19.ibd
```

How will partitioning help in quickly deleting rows and releasing space?

To archive old data in a partitioned table, we will create an empty table that is identical to the original table in structure but does not have multiple partitions like the original table. Once this table is created we will swap the newly created empty table with one of the partitions of the original table in a matter of seconds.

In this example, we are using a table partitioned on the basis of a date range.

```
mysql>show create table salaries\G
*************************** 1. row ***************************
       Table: salaries
Create Table: CREATE TABLE `salaries` (
  `emp_no` int(11) NOT NULL,
  `salary` int(11) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date NOT NULL,
  `response_code` blob NOT NULL,
  PRIMARY KEY (`emp_no`,`from_date`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(from_date)
(PARTITION p01 VALUES LESS THAN (‘1985-12-31’) ENGINE = InnoDB,
 PARTITION p02 VALUES LESS THAN (‘1986-12-31’) ENGINE = InnoDB,
 PARTITION p03 VALUES LESS THAN (‘1987-12-31’) ENGINE = InnoDB,
 PARTITION p04 VALUES LESS THAN (‘1988-12-31’) ENGINE = InnoDB,
 PARTITION p05 VALUES LESS THAN (‘1989-12-31’) ENGINE = InnoDB,
 PARTITION p06 VALUES LESS THAN (‘1990-12-31’) ENGINE = InnoDB,
 PARTITION p07 VALUES LESS THAN (‘1991-12-31’) ENGINE = InnoDB,
 PARTITION p08 VALUES LESS THAN (‘1992-12-31’) ENGINE = InnoDB,
 PARTITION p09 VALUES LESS THAN (‘1993-12-31’) ENGINE = InnoDB,
 PARTITION p10 VALUES LESS THAN (‘1994-12-31’) ENGINE = InnoDB,
 PARTITION p11 VALUES LESS THAN (‘1995-12-31’) ENGINE = InnoDB,
 PARTITION p12 VALUES LESS THAN (‘1996-12-31’) ENGINE = InnoDB,
 PARTITION p13 VALUES LESS THAN (‘1997-12-31’) ENGINE = InnoDB,
 PARTITION p14 VALUES LESS THAN (‘1998-12-31’) ENGINE = InnoDB,
 PARTITION p15 VALUES LESS THAN (‘1999-12-31’) ENGINE = InnoDB,
 PARTITION p16 VALUES LESS THAN (‘2000-12-31’) ENGINE = InnoDB,
 PARTITION p17 VALUES LESS THAN (‘2001-12-31’) ENGINE = InnoDB,
 PARTITION p18 VALUES LESS THAN (‘2002-12-31’) ENGINE = InnoDB,
 PARTITION p19 VALUES LESS THAN (MAXVALUE) ENGINE = InnoDB) */
1 row in set (0.00 sec)bd
```

The size of this table is around 105 GB.

```
mysql>select table_schema, table_name, table_rows, round(data_length / 1024 / 1024 / 1024 ) DATA_
MB, round(index_length / 1024 / 1024 / 1024 ) INDEX_GB, round(data_free / 1024 / 1024 / 1024)
FREE_MB, round(data_length / 1024 / 1024 / 1024 )+round(index_length / 1024 / 1024 )+round(data_
free / 1024 / 1024 /1024 ) TOTAL_MB from information_schema.tables where table_schema='employees'
and table_name='salaries';
+--------------+------------+------------+---------+----------+---------+---------+
| table_schema | table_name | table_rows | DATA_GB | INDEX_GB | FREE_GB | TOTAL_GB|
+--------------+------------+------------+---------+----------+---------+---------+
| employees    | salaries   |    2845404 |     105 | 
      0 |       0 |     105 |
+--------------+------------+------------+---------+----------+---------+---------+

1 row in set (0.00 sec)
```

We will create a table that is identical to the partitioned table but we will remove partitioning from this newly created table.

```
mysql> CREATE TABLE salaries_swap_p3 LIKE salaries;
Query OK, 0 rows affected (0.23 sec)

mysql> ALTER TABLE salaries_swap_p3 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.32 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

At this moment, this partition of the table has all the data and is occupying the space in the file system.

```
mysql> SELECT count(*) FROM salaries PARTITION (p03);
+----------+
| count(*) |
+----------+
|    57395 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT count(*) FROM salaries_swap_p3;
+----------+
| count(*) |
+----------+
|        0 |
+----------+
1 row in set (0.02 sec)
```

Space reserved by the file system:

```
[root@ip-172-31-83-227 employees]# ls -lrth *salaries*3*
-rw-r-----. 1 mysql mysql  96K Nov 21 03:54 salaries_swap_p3.ibd
-rw-r-----. 1 mysql mysql 8.5K Nov 21 03:56 salaries_swap_p3.frm
-rw-r-----. 1 mysql mysql  10G Nov 21 03:56 salaries#P#p03.ibd
```

To exchange partitions we need to execute a query that requires metadata lock on the table and is done almost instantaneously.

```
mysql> ALTER TABLE salaries EXCHANGE PARTITION p03 WITH TABLE salaries_swap_p3;
```

Since this operation requires metadata lock and is very fast, it is advised to run this operation in low-traffic periods.

```
mysql> show processlist;
+----+------+-----------+-----------+---------+------+---------------------------------+---------
-------------------------------------------------------------------+-----------+---------------+
| Id | User | Host      | db        | Command | Time | State                           | Info
| Rows_sent | Rows_examined |
+----+------+-----------+-----------+---------+------+---------------------------------+---------
-------------------------------------------------------------------+-----------+---------------+
|  4 | root | localhost | employees | Query   |   0  | Waiting for table metadata lock | ALTER
TABLE salaries EXCHANGE PARTITION p03 WITH TABLE salaries_swap_p3 |         0 |             0 |
+----+------+-----------+-----------+---------+------+---------------------------------+---------
-------------------------------------------------------------------+-----------+---------------+
1 row in set (0.00 sec)
```

After the swapping, the new table has all the data of the partition and the partition of the table is empty.

```
mysql>  SELECT count(*) FROM salaries PARTITION (p03);

+----------+
| count(*) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)

mysql>  SELECT count(*) FROM salaries_swap_p3;
+----------+
| count(*) |
+----------+
|    57395 |
+----------+
1 row in set (0.02 sec)
```

Space freed by filesystem:

```
[root@ip-172-31-83-227 employees]# ls -lrth *salaries*3*
-rw-r-----. 1 mysql mysql  10G Nov 21 03:54 salaries_swap_p3.ibd
-rw-r-----. 1 mysql mysql 8.5K Nov 21 03:56 salaries_swap_p3.frm
-rw-r-----. 1 mysql mysql  96K Nov 21 03:56 salaries#P#p03.ibd
```

Now, you can proceed to drop the swapped table without locking the main table or can use this technique as well to speed up the drop: Speed Up Your Large Table Drops in MySQL.

```
mysql> drop table if exists salaries_swap_p3;
Query OK, 0 rows affected (5.23 sec)
```

Conclusion

What we did here is use the MySQL partitioning-related functions to answer a customer challenge of purging data older than x days, sooner. But, partitioning is not a one size fits all solution, partitioning comes with its own caveats. If there are a lot of secondary indexes on a table and the search queries are not only limited to the partition key, the query performance could deteriorate exponentially. If partitioning the table is not an option, pt-archiver is a great tool that would automate the DELETE statement on your databases with minimal monitoring.

Schema design

As we dig even deeper into MySQL optimization tips, it's time to take a comprehensive look at the performance schema, an instrumental feature within MySQL designed to collect and provide detailed performance metrics. This chapter shows how to make complete use of it, guiding readers on how to harness its full potential to fine-tune MySQL databases for optimal performance.

Our expert explores the necessary steps to configure and query the performance schema effectively, identify what metrics and information it requires, and how to interpret these data points to make informed decisions. You will learn how to diagnose performance issues, monitor database operations in real time, and implement strategies for performance improvement.

Deep Dive Into MySQL Performance Schema

By Ankit Kapoor

Ankit was a Senior Database Architect (Professional Services) at Percona from 2022 to 2023, specializing in MySQL databases, MyROCKS, and distributed systems. He was previously a database architect with Alibaba Cloud.

Recently I was working with a customer wherein our focus was to carry out a performance audit of their multiple MySQL database nodes. We started looking into the stats of the **performance schema**. While working, the customer raised two interesting questions: how can he make complete use of the performance schema, and how can he find what he requires? I realized that it is important to understand the insights of the performance schema and how we can make effective use of it. This article should make it easier to understand for everyone.

The performance schema is an engine in MySQL which can easily be checked whether enabled or not using SHOW ENGINES. It is entirely built upon various sets of instruments (also can be called event names) each serving different purposes.

Instruments are the main part of the performance schema. It is useful when I want to investigate a problem and its root causes. Some of the examples are listed below (but not limited to) :

1. **Which IO operation is causing MySQL to slow down?**
2. **Which file a process/thread is mostly waiting for?**
3. **At which execution stage is a query taking time, or how much time will an alter command will take?**
4. **Which process is consuming most of the memory or how to identify the cause of memory leakage?**

What is an instrument in terms of performance schema?

Instruments are a combination of different sets of components like wait, io, sql, binlog, file, etc. If we combine these components, they become a meaningful tool to help us troubleshoot different issues. For example, wait/io/file/sql/binlog is one of the instruments providing information regarding the wait and I/O details on binary log files. Instruments are being read from left and then components will be added with delimiter "/". The more components we add to the instrument, the more complex or more specific it becomes, i.e. the more lengthy the instrument is, the more complex it goes.

You can locate all instruments available in your MySQL version under table setup_instruments. It is worth noting that every version of MySQL has a different number of instruments.

```
select count(1) from performance_schema.setup_instruments;

+----------+
| count(1) |
+----------+
|     1269 |
+----------+
```

For easy understanding, instruments can be divided into seven different parts as shown below. The **MySQL version I am using here is 8.0.30.** In earlier versions, we used to have only four, so expect to see different types of instruments in case you are using different/lower versions.

```
select distinct(substring_index(name,'/',1)) from performance_schema.setup_instruments;
+------------------------------+
| (substring_index(name,'/',1)) |
+------------------------------+
| wait                         |
| idle                         |
| stage                        |
| statement                    |
| transaction                  |
| memory                       |
| error                        |
+------------------------------+

7 rows in set (0.01 sec)
```

1. **Stage** - Instrument starting with 'stage' provides the execution stage of any query like reading data, sending data, altering table, checking query cache for queries, etc. For example stage/sql/altering table.
2. **Wait** - Instrument starting with 'wait' falls here. Like mutex waiting, file waiting, I/O waiting, and table waiting. Instrument for this can be wait/io/file/sql/map.

3. **Memory** - Instrument starting from “memory” providing information regarding memory usage on a per-thread basis. For example memory/sql/MYSQL_BIN_LOG
4. **Statement** - Instruments starting with “statement” provide information about the type of SQL, and stored procedures.
5. **Idle** - provide information on socket connection and information related to a thread.
6. **Transaction** - Provide information related to the transactions and have only one instrument.
7. **Error** - This single instrument provides information related to the errors generated by user activities. There are no further components attached to this instrument.

The total number of instruments for these seven components is listed below. You can identify these instruments starting with these names only.

```
select distinct(substring_index(name,'/',1)) as instrument_name,count(1) from performance_schema.
setup_instruments group by instrument_name;

+-----------------+----------+
| instrument_name | count(1) |
+-----------------+----------+
| wait            |      399 |
| idle            |        1 |
| stage           |      133 |
| statement       |      221 |
| transaction     |        1 |
| memory          |      513 |
| error           |        1 |
+-----------------+----------+
```

How to find which instrument you need

I do remember that a customer asked me since there are thousands of instruments available, how can he find out which one he requires. As I mentioned before that instruments are being read from left to right, we can find out which instrument we require and then find its respective performance.

For example - I need to observe the performance of redo logs (log files or WAL files) of my MySQL instance and need to check if threads/connections need to wait for the redo log files to be flushed before further writing and if so then how much.

```
select * from setup_instruments where name like '%innodb_log_file%';

+---------------------------------------+---------+-------+------------+------------+---------------+
| NAME                                  | ENABLED | TIMED | PROPERTIES | VOLATILITY | DOCUMENTATION |
+---------------------------------------+---------+-------+------------+------------+---------------+
| wait/synch/mutex/innodb/log_files_mutex | NO      | NO    |            |          0 | NULL          |
| wait/io/file/innodb/innodb_log_file     | YES     | YES   |            |          0 | NULL          |
+---------------------------------------+---------+-------+------------+------------+---------------+
```

Here you see that I have two instruments for redo log files. One is for the mutex stats on the redo log files and the second is for the IO wait stats on the redo log files.

Example two - You need to find out those operations or instruments for which you can calculate the time required i.e. how much time a bulk update will take. Below are all the instruments that help you to locate the same.

```
select * from setup_instruments where PROPERTIES='progress';

+-------------------------------------------------------+---------+-------+------------+------------+---------------
+
| NAME                                                  | ENABLED | TIMED | PROPERTIES | VOLATILITY | DOCUMENTATION
|
+-------------------------------------------------------+---------+-------+------------+------------+---------------
+
| stage/sql/copy to tmp table                           | YES     | YES   | progress   |          0 | NULL          |
| stage/sql/Applying batch of row changes (write)       | YES     | YES   | progress   |          0 | NULL          |
| stage/sql/Applying batch of row changes (update)      | YES     | YES   | progress   |          0 | NULL          |
| stage/sql/Applying batch of row changes (delete)      | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (end)                        | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (flush)                      | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (insert)                     | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (log apply index)            | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (log apply table)            | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (merge sort)                 | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter table (read PK and internal sort)  | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/alter tablespace (encryption)            | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/buffer pool load                         | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/clone (file copy)                        | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/clone (redo copy)                        | YES     | YES   | progress   |          0 | NULL          |
| stage/innodb/clone (page copy)                        | YES     | YES   | progress   |          0 | NULL          |
+-------------------------------------------------------+---------+-------+------------+------------+---------------+
```

The above instruments are the ones for which progress can be tracked.

How to prepare these instruments to troubleshoot the performance issues

To take advantage of these instruments, they need to be enabled first to make the performance schema log-related data. In addition to logging the information of running threads, it is also possible to maintain the history of such threads (statement/stages or any particular operation). Let's see, by default, how many instruments are enabled in the version I am using. I have not enabled any other instrument explicitly.
select count(*) from setup_instruments where ENABLED='YES';

```
+----------+
| count(*) |
+----------+
|      810 |
+----------+

1 row in set (0.00 sec)
```

The below query lists the top 30 enabled instruments for which logging will take place in the tables.
select * from performance_schema.setup_instruments where enabled='YES' limit 30;

```
+---------------------------------------+---------+-------+------------+------------+---------------+
| NAME                                  | ENABLED | TIMED | PROPERTIES | VOLATILITY | DOCUMENTATION |
+---------------------------------------+---------+-------+------------+------------+---------------+
| wait/io/file/sql/binlog               | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/binlog_cache         | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/binlog_index         | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/binlog_index_cache   | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/relaylog             | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/relaylog_cache       | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/relaylog_index       | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/relaylog_index_cache | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/io_cache             | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/casetest             | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/dbopt                | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/ERRMSG               | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/select_to_file       | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/file_parser          | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/FRM                  | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/load                 | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/LOAD_FILE            | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/log_event_data       | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/log_event_info       | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/misc                 | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/pid                  | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/query_log            | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/slow_log             | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/tclog                | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/trigger_name         | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/trigger              | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/init                 | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/SDI                  | YES     | YES   |            |          0 | NULL          |
| wait/io/file/sql/hash_join            | YES     | YES   |            |          0 | NULL          |
| wait/io/file/mysys/proc_meminfo       | YES     | YES   |            |          0 | NULL          |
+---------------------------------------+---------+-------+------------+------------+---------------+
```

As I mentioned previously, it is also possible to maintain the history of the events. For example, if you are running a load test and want to analyze the performance of queries after its completion, you need to activate the below consumers if not activated yet.

```
select * from performance_schema.setup_consumers;

+----------------------------------+---------+
| NAME                             | ENABLED |
+----------------------------------+---------+
| events_stages_current            | YES     |
| events_stages_history            | YES     |
| events_stages_history_long       | YES     |
| events_statements_cpu            | YES     |
| events_statements_current        | YES     |
| events_statements_history        | YES     |
| events_statements_history_long   | YES     |
| events_transactions_current      | YES     |
| events_transactions_history      | YES     |
| events_transactions_history_long | YES     |
| events_waits_current             | YES     |
| events_waits_history             | YES     |
| events_waits_history_long        | YES     |
| global_instrumentation           | YES     |
| thread_instrumentation           | YES     |
| statements_digest                | YES     |
+----------------------------------+---------+
```

Note - The top 15 records in the above rows are self-explanatory, but the last one for digest means to allow the digest text for SQL statements. By digest I mean, grouping similar queries and showing their performance. This is being done by hashing algorithms.

Let's say, you want to analyze the stages of a query that is spending most of the time in, you need to enable the respective logging using the below query.

```
MySQL> update performance_schema.setup_consumers set ENABLED='YES' where NAME='events_stages_
current';

Query OK, 1 row affected (0.00 sec)

Rows matched: 1  Changed: 1  Warnings: 0
```

How to take advantage of the performance schema?

Now that we know what instruments are, how to enable them, and the amount of data we want to store in, it's time to understand how to make use of these instruments. To make it easier to understand I have taken the output of a few instruments from my test cases as it won't be possible to cover all as there are more than a thousand instruments.

Please note that to generate the fake load, I used sysbench (if you are not familiar with it, read about it here) to create read and write traffic using the below details :

```
lua : oltp_read_write.lua

Number of tables : 1

table_Size : 100000

threads : 4/10

rate - 10
```

As an example, think about a case when you want to find out where memory is getting utilized. To find out this, let's execute the below query in the table related to the memory.

```
select * from memory_summary_global_by_event_name order by SUM_NUMBER_OF_BYTES_ALLOC desc limit
3\G;

*************************** 1. row ***************************
                  EVENT_NAME: memory/innodb/buf_buf_pool
                 COUNT_ALLOC: 24
                  COUNT_FREE: 0
   SUM_NUMBER_OF_BYTES_ALLOC: 3292102656
    SUM_NUMBER_OF_BYTES_FREE: 0
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 24
             HIGH_COUNT_USED: 24
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 3292102656
   HIGH_NUMBER_OF_BYTES_USED: 3292102656
*************************** 2. row ***************************
                  EVENT_NAME: memory/sql/THD::main_mem_root
                 COUNT_ALLOC: 138566
                  COUNT_FREE: 138543
   SUM_NUMBER_OF_BYTES_ALLOC: 2444314336
    SUM_NUMBER_OF_BYTES_FREE: 2443662928
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 23
             HIGH_COUNT_USED: 98
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 651408
   HIGH_NUMBER_OF_BYTES_USED: 4075056
*************************** 3. row ***************************
                  EVENT_NAME: memory/sql/Filesort_buffer::sort_keys
                 COUNT_ALLOC: 58869
                  COUNT_FREE: 58868
   SUM_NUMBER_OF_BYTES_ALLOC: 2412676319
    SUM_NUMBER_OF_BYTES_FREE: 2412673879
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 1
             HIGH_COUNT_USED: 13
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 2440
   HIGH_NUMBER_OF_BYTES_USED: 491936
```

Above are the top three records, showing where the memory is getting mostly utilized.

Instrument **memory/innodb/buf_buf_pool** is related to the **buffer pool** which is utilizing 3 GB and we can fetch this information from **SUM_NUMBER_OF_BYTES_ALLOC**. Another data that is also important for us to consider is **CURRENT_COUNT_USED** which tells us how many blocks of data have been currently allocated and once work is done, the value of this column will be modified. Looking at the stats of this record, consumption of 3GB is not a problem since MySQL uses a buffer pool quite frequently (for example, while writing data, loading data, modifying data, etc.). But the problem rises, when you have memory leakage issues or the buffer pool is not getting used. In such cases, this instrument is quite useful to analyze.

Looking at the second instrument memory/sql/THD::main_mem_root which is utilizing 2G, is related to the sql (that's how we should read it from the very left). THD::main_mem_root is one of the thread classes. Let us try to understand this instrument:

THD represent thread

main_mem_root is a class of mem_root. MEM_ROOT is a structure being used to allocate memory to threads while parsing the query, during execution plans, during execution of nested queries/sub-queries and other allocations while query execution. Now, in our case we want to check which thread/host is

consuming memory so that we can further optimize the query. Before digging down further, let's understand the 3rd instrument first which is an important instrument to look for.

memory/sql/filesort_buffer::sort_keys - As I mentioned earlier, instrument names should be read starting from left. In this case, it is related to memory allocated to sql. The next component in this instrument is **filesort_buffer::sort_keys** which is responsible for sorting the data (it can be a buffer in which data is stored and needs to be sorted. Various examples of this can be index creation or normal order by clause)

It's time to dig down and analyze which connection is using this memory. To find out this, I have used table **memory_summary_by_host_by_event_name** and filtered out the record coming from my application server.

```
select * from memory_summary_by_host_by_event_name where HOST='10.11.120.141' order by SUM_
NUMBER_OF_BYTES_ALLOC desc limit 2\G;

*************************** 1. row ***************************
                       HOST: 10.11.120.141
                 EVENT_NAME: memory/sql/THD::main_mem_root
                COUNT_ALLOC: 73817
                 COUNT_FREE: 73810
   SUM_NUMBER_OF_BYTES_ALLOC: 1300244144
    SUM_NUMBER_OF_BYTES_FREE: 1300114784
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 7
             HIGH_COUNT_USED: 39
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 129360
   HIGH_NUMBER_OF_BYTES_USED: 667744
*************************** 2. row ***************************
                       HOST: 10.11.120.141
                 EVENT_NAME: memory/sql/Filesort_buffer::sort_keys
                COUNT_ALLOC: 31318
                 COUNT_FREE: 31318
   SUM_NUMBER_OF_BYTES_ALLOC: 1283771072
    SUM_NUMBER_OF_BYTES_FREE: 1283771072
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 0
             HIGH_COUNT_USED: 8
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 0
   HIGH_NUMBER_OF_BYTES_USED: 327936
```

Event name **memory/sql/THD::main_mem_root** has consumed more than 1G memory (sum) by the host 11.11.120.141 which is my application host at the time of executing this query. Now since we know that this host is consuming memory, we can dig down further to find out the queries like nested or subquery and then try to optimize it.

Similarly, if we see the memory allocation by f**ilesort_buffer::sort_keys** is also more than 1G (total) at the time of execution. Such instruments signal us to refer to any queries using sorting i.e. order by clause.

Time to join all dotted lines

Let's try to find out the culprit thread in one of the cases where most of the memory is being utilized by the file sort. The **first query** helps us in finding the host and event name (instrument):

```
select * from memory_summary_by_host_by_event_name order by SUM_NUMBER_OF_BYTES_ALLOC desc limit
1\G;

*************************** 1. row ***************************
                      HOST: 10.11.54.152
                EVENT_NAME: memory/sql/Filesort_buffer::sort_keys
               COUNT_ALLOC: 5617297
                COUNT_FREE: 5617297
   SUM_NUMBER_OF_BYTES_ALLOC: 193386762784
    SUM_NUMBER_OF_BYTES_FREE: 193386762784
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 0
             HIGH_COUNT_USED: 20
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 0
   HIGH_NUMBER_OF_BYTES_USED: 819840
```

Ahan, this is my application host, and let's find out which user is executing and its respective thread id.

```
select * from memory_summary_by_account_by_event_name where HOST='10.11.54.152' order by SUM_
NUMBER_OF_BYTES_ALLOC desc limit 1\G;

*************************** 1. row ***************************
                      USER: sbuser
                      HOST: 10.11.54.152
                EVENT_NAME: memory/sql/Filesort_buffer::sort_keys
               COUNT_ALLOC: 5612993
                COUNT_FREE: 5612993
   SUM_NUMBER_OF_BYTES_ALLOC: 193239513120
    SUM_NUMBER_OF_BYTES_FREE: 193239513120
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 0
             HIGH_COUNT_USED: 20
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 0
   HIGH_NUMBER_OF_BYTES_USED: 819840

select * from memory_summary_by_thread_by_event_name where EVENT_NAME='memory/sql/Filesort_
buffer::sort_keys' order by SUM_NUMBER_OF_BYTES_ALLOC desc limit 1\G;

*************************** 1. row ***************************
                 THREAD_ID: 84
                EVENT_NAME: memory/sql/Filesort_buffer::sort_keys
               COUNT_ALLOC: 565645
                COUNT_FREE: 565645
   SUM_NUMBER_OF_BYTES_ALLOC: 19475083680
    SUM_NUMBER_OF_BYTES_FREE: 19475083680
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 0
             HIGH_COUNT_USED: 2
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 0
   HIGH_NUMBER_OF_BYTES_USED: 81984
```

Now, we have the complete details of the user and its thread id. Let's see which sort of queries are being executed by this thread.

```
select * from events_statements_history where THREAD_ID=84 order by SORT_SCAN desc\G;

*************************** 1. row ***************************
              THREAD_ID: 84
               EVENT_ID: 48091828
           END_EVENT_ID: 48091833
             EVENT_NAME: statement/sql/select
                 SOURCE: init_net_server_extension.cc:95
            TIMER_START: 145083499054314000
              TIMER_END: 145083499243093000
             TIMER_WAIT: 188779000
              LOCK_TIME: 1000000
               SQL_TEXT: SELECT c FROM sbtest2 WHERE id BETWEEN 5744223 AND 5744322 ORDER BY c
                 DIGEST: 4f764af1c0d6e44e4666e887d454a241a09ac8c4df9d5c2479f08b00e4b9b80d
            DIGEST_TEXT: SELECT `c` FROM `sbtest2` WHERE `id` BETWEEN ? AND ? ORDER BY `c`
         CURRENT_SCHEMA: sysbench
            OBJECT_TYPE: NULL
          OBJECT_SCHEMA: NULL
            OBJECT_NAME: NULL
  OBJECT_INSTANCE_BEGIN: NULL
            MYSQL_ERRNO: 0
      RETURNED_SQLSTATE: NULL
           MESSAGE_TEXT: NULL
                 ERRORS: 0
               WARNINGS: 0
          ROWS_AFFECTED: 0
              ROWS_SENT: 14
          ROWS_EXAMINED: 28
CREATED_TMP_DISK_TABLES: 0
     CREATED_TMP_TABLES: 0
       SELECT_FULL_JOIN: 0
 SELECT_FULL_RANGE_JOIN: 0
           SELECT_RANGE: 1
     SELECT_RANGE_CHECK: 0
            SELECT_SCAN: 0
      SORT_MERGE_PASSES: 0
             SORT_RANGE: 0
              SORT_ROWS: 14
              SORT_SCAN: 1
          NO_INDEX_USED: 0
     NO_GOOD_INDEX_USED: 0
       NESTING_EVENT_ID: NULL
     NESTING_EVENT_TYPE: NULL
    NESTING_EVENT_LEVEL: 0
           STATEMENT_ID: 49021382
               CPU_TIME: 185100000
       EXECUTION_ENGINE: PRIMARY
```

I have pasted one record only as per rows_scan (which refers to the table scan) here but you can find similar other queries in your case and then try to optimize it either by creating an index or some other suitable solution.

Example two

Let's try to find out the situation of table locking i.e. which lock i.e. read lock, write lock, etc., has been put on the user table and for what duration (displayed in pico seconds).

Lock a table with write lock:

```
mysql> lock tables sbtest2 write;

Query OK, 0 rows affected (0.00 sec)
mysql> show processlist;

+----+--------+---------------------+--------------------+-------------+--------+-----------------------------
-----------------------------------+------------------+-----------+-----------+---------------+
| Id | User   | Host                | db                 | Command     | Time   | State
| Info             | Time_ms    | Rows_sent | Rows_examined |
+----+--------+---------------------+--------------------+-------------+--------+-----------------------------
-----------------------------------+------------------+-----------+-----------+---------------+
|  8 | repl   | 10.11.139.171:53860 | NULL               | Binlog Dump | 421999 | Source has sent all binlog
to replica; waiting for more updates | NULL             | 421998368 |         0 |             0 |
|  9 | repl   | 10.11.223.98:51212  | NULL               | Binlog Dump | 421998 | Source has sent all binlog
to replica; waiting for more updates | NULL             | 421998262 |         0 |             0 |
| 25 | sbuser | 10.11.54.152:38060  | sysbench           | Sleep       |  65223 |
| NULL             |   65222573 |         0 |             1 |
| 26 | sbuser | 10.11.54.152:38080  | sysbench           | Sleep       |  65222 |
| NULL             |   65222177 |         0 |             1 |
| 27 | sbuser | 10.11.54.152:38090  | sysbench           | Sleep       |  65223 |
| NULL             |   65222438 |         0 |             0 |
| 28 | sbuser | 10.11.54.152:38096  | sysbench           | Sleep       |  65223 |
| NULL             |   65222489 |         0 |             1 |
| 29 | sbuser | 10.11.54.152:38068  | sysbench           | Sleep       |  65223 |
| NULL             |   65222527 |         0 |             1 |
| 45 | root   | localhost           | performance_schema | Sleep       |   7722 |
| NULL             |    7722009 |        40 |           348 |
| 46 | root   | localhost           | performance_schema | Sleep       |   6266 |
| NULL             |    6265800 |        16 |          1269 |
| 47 | root   | localhost           | performance_schema | Sleep       |   4904 |
| NULL             |    4903622 |         0 |            23 |
| 48 | root   | localhost           | performance_schema | Sleep       |   1777 |
| NULL             |    1776860 |         0 |             0 |
| 54 | root   | localhost           | sysbench           | Sleep       |    689 |
| NULL             |     688740 |         0 |             1 |
| 58 | root   | localhost           | NULL               | Sleep       |     44 |
| NULL             |      44263 |         1 |             1 |
| 59 | root   | localhost           | sysbench           | Query       |      0 | init
| show processlist |          0 |         0 |             0 |
+----+--------+---------------------+--------------------+-------------+--------+-----------------------------
-----------------------------------+------------------+-----------+-----------+---------------+
```

Now, think of a situation wherein you are not aware of this session and you are trying to read this table and thus waiting for the meta data locks. In this situation, we need to take the help of instruments (to find out which session is locking this table) related to the lock i.e. wait/table/lock/sql/handler (table_handles is the table responsible for table lock instruments):

```
mysql> select * from table_handles where object_name='sbtest2' and OWNER_THREAD_ID is not null;

+-------------+---------------+-------------+-----------------------+-----------------+----------
------+---------------+-----------------+
| OBJECT_TYPE | OBJECT_SCHEMA | OBJECT_NAME | OBJECT_INSTANCE_BEGIN | OWNER_THREAD_ID | OWNER_
EVENT_ID | INTERNAL_LOCK | EXTERNAL_LOCK   |
+-------------+---------------+-------------+-----------------------+-----------------+----------
------+---------------+-----------------+
| TABLE       | sysbench      | sbtest2     |       140087472317648 |             141 |
77 | NULL          | WRITE EXTERNAL |
+-------------+---------------+-------------+-----------------------+-----------------+----------
------+---------------+-----------------+
mysql> select * from metadata_locks;
+---------------+--------------------+-----------------+-------------+-----------------------+--
--------------------+---------------+-------------+-------------------+-----------------+--------
--------+
| OBJECT_TYPE   | OBJECT_SCHEMA      | OBJECT_NAME     | COLUMN_NAME | OBJECT_INSTANCE_BEGIN |
LOCK_TYPE           | LOCK_DURATION | LOCK_STATUS | SOURCE            | OWNER_THREAD_ID | OWNER_
EVENT_ID |
+---------------+--------------------+-----------------+-------------+-----------------------+--
--------------------+---------------+-------------+-------------------+-----------------+--------
--------+
| GLOBAL        | NULL               | NULL            | NULL        |       140087472151024
| INTENTION_EXCLUSIVE  | STATEMENT     | GRANTED     | sql_base.cc:5534  |             141 |
77 |
| SCHEMA        | sysbench           | NULL            | NULL        |       140087472076832
| INTENTION_EXCLUSIVE  | TRANSACTION   | GRANTED     | sql_base.cc:5521  |             141 |
77 |
| TABLE         | sysbench           | sbtest2         | NULL        |       140087471957616
| SHARED_NO_READ_WRITE | TRANSACTION   | GRANTED     | sql_parse.cc:6295 |             141 |
77 |
| BACKUP TABLES | NULL               | NULL            | NULL        |       140087472077120
| INTENTION_EXCLUSIVE  | STATEMENT     | GRANTED     | lock.cc:1259      |             141 |
77 |
| TABLESPACE    | NULL               | sysbench/sbtest2 | NULL       |       140087471954800
| INTENTION_EXCLUSIVE  | TRANSACTION   | GRANTED     | lock.cc:812       |             141 |
77 |
| TABLE         | sysbench           | sbtest2         | NULL        |       140087673437920
| SHARED_READ          | TRANSACTION   | PENDING     | sql_parse.cc:6295 |             142 |
77 |
| TABLE         | performance_schema | metadata_locks  | NULL        |       140088117153152
| SHARED_READ          | TRANSACTION   | GRANTED     | sql_parse.cc:6295 |             143 |
970 |
| TABLE         | sysbench           | sbtest1         | NULL        |       140087543861792
| SHARED_WRITE         | TRANSACTION   | GRANTED     | sql_parse.cc:6295 |             132 |
156 |
+---------------+--------------------+-----------------+-------------+-----------------------+--
--------------------+---------------+-------------+-------------------+-----------------+--------
--------+
```

From here we know that thread id 141 is holding the lock “SHARED_NO_READ_WRITE” on sbtest2 and thus we can take the corrective step i.e. either commit the session or kill it, once we realize its requirement. We need to find the respective processlist_id from the threads table to kill it.

```
mysql> kill 63;

Query OK, 0 rows affected (0.00 sec)
mysql> select * from table_handles where object_name='sbtest2' and OWNER_THREAD_ID is not null;

Empty set (0.00 sec)
```

Example three

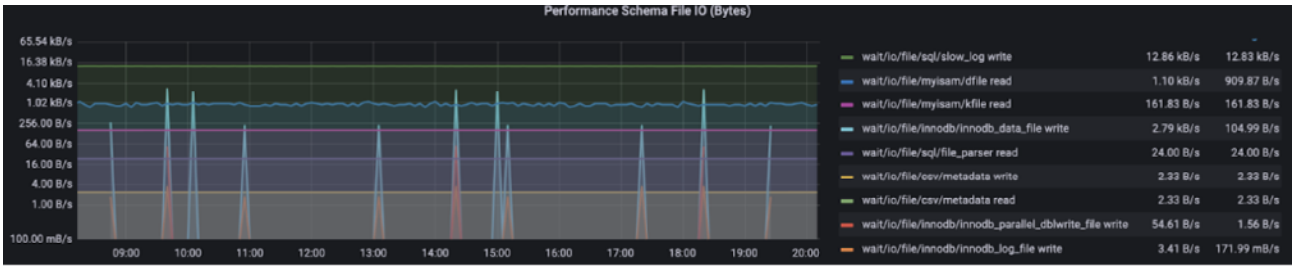
In some situations, we need to find out where our MySQL server is spending most of the time waiting so that we can take further steps :

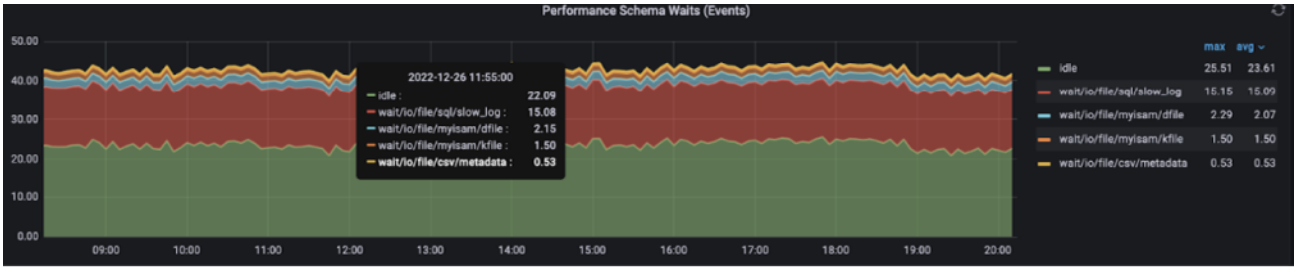
```
mysql> select * from events_waits_history order by TIMER_WAIT desc limit 2\G;

*************************** 1. row ***************************
            THREAD_ID: 88
             EVENT_ID: 124481038
         END_EVENT_ID: 124481038
           EVENT_NAME: wait/io/file/sql/binlog
               SOURCE: mf_iocache.cc:1694
          TIMER_START: 356793339225677600
            TIMER_END: 420519408945931200
           TIMER_WAIT: 63726069720253600
                SPINS: NULL
        OBJECT_SCHEMA: NULL
          OBJECT_NAME: /var/lib/mysql/mysqld-bin.000009
           INDEX_NAME: NULL
          OBJECT_TYPE: FILE
OBJECT_INSTANCE_BEGIN: 140092364472192
     NESTING_EVENT_ID: 124481033
   NESTING_EVENT_TYPE: STATEMENT
            OPERATION: write
      NUMBER_OF_BYTES: 683
                FLAGS: NULL
*************************** 2. row ***************************
            THREAD_ID: 142
             EVENT_ID: 77
         END_EVENT_ID: 77
           EVENT_NAME: wait/lock/metadata/sql/mdl
               SOURCE: mdl.cc:3443
          TIMER_START: 424714091048155200
            TIMER_END: 426449252955162400
           TIMER_WAIT: 1735161907007200
                SPINS: NULL
        OBJECT_SCHEMA: sysbench
          OBJECT_NAME: sbtest2
           INDEX_NAME: NULL
          OBJECT_TYPE: TABLE
OBJECT_INSTANCE_BEGIN: 140087673437920
     NESTING_EVENT_ID: 76
   NESTING_EVENT_TYPE: STATEMENT
            OPERATION: metadata lock
      NUMBER_OF_BYTES: NULL
                FLAGS: NULL
2 rows in set (0.00 sec)
```

In the above example, bin log file has waited most of the time (timer_wait in pico seconds) to perform IO operations in mysqld-bin.000009. It may be because of several reasons, for example, storage is full. The next records show the details of example two I explained previously.

What else?

To make life more convenient and easy to monitor these instruments, Percona Monitoring and Management plays an important role. For example, see the below snapshots.





We can configure almost all instruments and instead of querying, we can just make use of these graphs.

Obviously, knowing about performance schema helps us a lot but also enabling all of them incurs additional costs and impacts performance. Hence, in many cases, Percona Toolkit is helpful without impacting the DB performance. For example, pt-index-usage, pt-online schema change, pt-query-digest.

Some important points

1. History table loads after a while, not instantly. Only after completion of a thread activity.
2. Enabling all instruments may impact the performance of your MySQL as we are enabling more writes to these in-memory tables. Also, it will impose additional money on your budget. Hence enable as per requirements only.
3. PMM contains most of the instruments and is also possible to configure more as per your requirements.
4. You don't need to remember the name of all the tables. You can just use PMM or use joins to create the queries. This article hashes the entire concept into smaller chunks and thus didn't use any joins so that readers can understand it.
5. The best method of enabling multiple instruments is in the staging environment and then optimize your findings and then moving to the productions.

Conclusion

Performance schemas are a great help while troubleshooting the behavior of your MySQL server. You need to find out which instrument you need. Should you be still struggling with the performance, please don't hesitate to reach us and we will be more than happy to help you.

Cost optimization

As a bonus chapter for our eBook on MySQL optimization techniques, this chapter presents Peter Zaitsev's expert tips on reducing operational costs without compromising database performance. Now more than ever, organizations are seeking ways to minimize expenses while maintaining or enhancing their database systems' efficiency and reliability. This is particularly important for those operating MySQL databases in cloud environments, where cost dynamics are transparent, and adjustments can lead to immediate financial benefits.

Through this chapter, Peter equips you with seven strategies to significantly lower the costs associated with running MySQL databases. By applying his guidance, you'll not only become a cost-saving hero in your organization but also ensure that your MySQL databases are optimized for both performance and financial efficiency.

Seven Ways To Reduce MySQL Costs in the Cloud

By Peter Zaitsev

Your organization will love you if you find ways to reduce the costs of running their MySQL databases. This is especially true if you run MySQL in the cloud, as it often allows you to see the immediate effect of those savings, which is what this article will focus on.

1. Optimize your schema and queries

While optimizing schema and queries is only going to do so much to help you to save on MySQL costs in the cloud, it is a great thing to start with. Suboptimal schema and queries can require a much larger footprint, and it is also something "fully managed" Database as a Service (DBaaS) solutions from major cloud vendors do not really help you with. It is not uncommon for a system with suboptimal schema and queries to require 10x or more resources to run than an optimized system.

At Percona, we've built Percona Monitoring and Management (PMM), which helps you to find out which of the queries need attention and how to optimize it. If you need more help, Percona Professional Services are often able to get your schema and queries in shape in a matter of days, providing long-term saving opportunities with a very reasonable upfront cost.

2. Tune your MySQL configuration

Optimal MySQL configuration depends on the workload, which is why I recommend tuning your queries and schema first. While gains from MySQL configuration tuning are often smaller than from fixing your queries, it is still significant. We have an old but still very relevant article on the basic MySQL settings you'll want to tune, which you can check out. You can also consider using tools like Releem or Ottertune to help you to get a better MySQL configuration for your workload.

More advanced tuning might include exploring alternative storage engines, such as MyRocks (included in Percona Distribution for MySQL). MyRocks can offer fantastic compression and minimize required IO, thereby drastically reducing storage costs. For a more in-depth look at MyRocks performance in the cloud, check out the blog post Scaling IO-Bound Workloads for MySQL in the Cloud.

3. Implement caching

Caching is cheating — and it works great! The most state-of-the-art caching for MySQL is available by rolling out ProxySQL. It can provide some additional performance benefits, such as connection pooling and read-write splitting, but I think caching is most generally useful for MySQL cost reduction. ProxySQL is fully supported by Percona with a Percona Platform subscription and is included in Percona Distribution for MySQL.

Enabling query cache for your heavy queries can be truly magical — often enough it is the heaviest queries that do not need to provide the most up-to-date information and can have their results cached for a significant time.

Read more about how to configure ProxySQL caching:

- Scaling With ProxySQL Query Cache
- How to Optimize MySQL Queries With ProxySQL Caching on Ubuntu 16.04
- ProxySQL Series: Query Cache With ProxySQL

4. Rightsize your resources

Once you have optimized your schema and queries and tuned MySQL configuration, you can check what resources your MySQL instances are using and where you can put them on a diet without negatively impacting performance. CPU, memory, disk, and network are four primary resources that impact MySQL performance and often can be managed semi-independently in the cloud. For example, if your workload needs a lot of CPU power but does not need a lot of memory for caching, you can consider CPU-intensive instances. PMM has some great tools to understand which resources are most demanded by your workload.

You can also read some additional resource-specific tips in my other article on MySQL performance.

In our experience, due to the simplicity of "scaling with credit cards", many databases in the cloud become grossly over-provisioned over time, and there can be a lot of opportunity for savings with instance size reduction!

5. Ditch DBaaS for Kubernetes

The price differential between DBaaS and comparable resources continues to grow. With the latest Graviton instances, you will pay double for Amazon RDS compared to the cost of the underlying instance. Amazon Aurora, while offering some wonderful features, is even more expensive. If you are deploying just a couple of small nodes, this additional cost beats having to hire people to deploy and manage "do it yourself" solutions. If you're spending tens of thousands of dollars a month on your DBaaS solution, the situation may be different.

A few years ago, building your own database service using building blocks like EC2, EBS, or using a DBaaS solution such as Amazon RDS for MySQL were the only solutions. Now, another opportunity has emerged — using Kubernetes.

You can get Amazon EKS - Managed Kubernetes Service, Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS) for a relatively small premium as infrastructure costs. Then, you can use Percona's MySQL Operator to deploy and manage your database at a fraction of the complexity of traditional deployments. If you're using Kubernetes for your apps already or use an infrastructure as code (IaC) deployment approach, it may even be handier than DBaaS solutions and 100% open source.

6. Consider lower-cost alternatives

A few years ago, only major cloud providers (in the U.S.: AWS, GCP, Azure) had DBaaS solutions for MySQL. Now the situation has changed, with MySQL DBaaS being available from second-tier and typically lower-cost providers, too. You can get MySQL DBaaS from Linode, Digital Ocean, and Vultr at a significantly lower cost (though, with fewer features). You can also get MySQL from independent providers like Aiven.

If you're considering deploying databases on a cloud vendor different than the one your application uses, make sure you're using a close location for deployment and make sure to check the network latency between your database and your application — poor network latency or reliability issues can negate all the savings you've achieved.

7. Let experts manage your MySQL database

If you're spending more than $20,000 a month on your cloud database footprint, or if your application is growing or changing rapidly, you'll often have better database performance, security, and reliability at lower cost by going truly "fully managed" instead of "cloud fully managed." The latter relies on "shared responsibility" in many key areas of MySQL best practices.

"Cloud fully managed" will ensure the database infrastructure is operating properly (if scaled with a credit card), but will not fully ensure you have optimal MySQL configuration and optimized queries, are following best security practices, or picked the most performant and cost-effective instance for your MySQL deployment.

Percona Managed Services is a solution from Percona to consider, though there are many other experts on the market that can take better care of your MySQL needs than DBaaS at major clouds.

Summary

If the costs of your MySQL infrastructure in the cloud are starting to bite, do not despair. There are likely plenty of savings opportunities available, and I hope some of the above tips will apply to your environment. It's also worth noting that my above advice is not theory, but is based on the actual work we've done at Percona. We've helped many leading companies realize significant cost savings when running MySQL in the cloud, including Patreon, which saved more than 50% on their cloud database infrastructure costs with Percona.

Conclusion

Equipped with this advice from Percona's MySQL experts, you now have the insights and strategies to tackle some of the most challenging aspects of database optimization. From the intricacies of query fine-tuning to the critical considerations of InnoDB configuration and beyond, our hope is that this ebook has provided you with a solid overview of what it takes to build and manage high-performing MySQL databases.

We encourage you to revisit these pages whenever you need inspiration or guidance and to stay connected with the vibrant community of MySQL professionals. And for further exploration of performance optimization strategies and expert advice, we invite you to **visit and subscribe to our blog**. Our tips, insights, and detailed guides are designed to help you navigate the complexities of database management and optimization. Whether you're looking for advanced techniques or practical advice, you'll find a wealth of knowledge to help enhance your database's performance and efficiency.

If you're interested in a comprehensive assessment of your database's health and performance, please consider visiting our **Database Performance Assessment** page. By opting for a Database Wellness Exam, you not only avoid the high costs associated with poor database performance but also gain a partner in ensuring your database environment is optimized for peak efficiency. Our team of experts is ready to provide you with tailored recommendations to improve system resource utilization, resolve query bottlenecks, enhance scalability, and much more. Whether you choose to implement these recommendations on your own or with our continued support, Percona is here to help every step of the way.

Databases Run Better
with Percona

percona.com