

TOP METRICS TO MONITOR IN MYSQL DATABASES

BY SHREE NAIR

TABLE OF CONTENTS

Introduction	3
Benefits of following a monitoring plan	4
Database metric types	4
How often should one perform monitoring	5
Performance metric categories	6
Work metrics	6
Resource metrics	7
Monitoring performance	8
Best work metrics to monitor	8
Throughput	8
Throughput metrics in MySQL	9
The questions and queries status variable	9
The threads_running status variable	10
The slow_queries server variable	11
The sys_schema	13
Best resource metrics to monitor	14
Connections	14
Aborted client and connections	17
Connection errors	18
Buffer pool usage.....	21
Configuring the buffer pool	21
Important InnoDB buffer pool metrics	22
A word about the InnoDB buffer pool LRU algorithm	23
Some MySQL performance tuning tips	24
Converting buffer pool metrics to bytes	28
Conclusion	29
About the Author	32

INTRODUCTION

As tables increase in size and more and more users come online, it becomes necessary to fine-tune the database server from time to time. The secret to knowing what adjustments to make is to perform regular monitoring. Most databases offer dozens, if not hundreds, of performance metrics that one can assess.

As a database administrator, the top priority is to keep the databases and dependent applications running smoothly at all times. The best weapon is the careful monitoring of key performance metrics. In a perfect world, one would want to be up-to-date regarding every aspect of the activity of the database. One would want to know how many events occurred, how big they were, when they happened, and how long they took to complete.

There is no shortage of tools that can monitor resource consumption, provide instantaneous status snapshots, and generate wait analysis and graphs. The challenge is that some metrics can be expensive to measure, and perhaps they can require much work to analyze.

This whitepaper focuses on monitoring key performance metrics.

This whitepaper describes how to:

- Examine the benefits of performance monitoring.
- Outline the primary performance metric categories.
- List the monitoring tools provided by MySQL:
 - Server variables
 - Performance schema
 - Sys schema
- Learn how to monitor:
 - Transaction throughput
 - Query execution performance

Further, this whitepaper narrows down the field to the performance metrics that provide the most value for the effort. Also, this whitepaper presents some tangible ways to capture and study them. It is by tracking the most useful metrics and reviewing them in the most informative ways that one balance paranoid over-monitoring and unforeseen firefighting crises. This whitepaper focuses on monitoring database connections and buffer pool metrics.

BENEFITS OF FOLLOWING A MONITORING PLAN

A database, including MySQL, back most applications. It is crucial to monitor databases effectively to keep databases and applications running smoothly. A good database monitoring plan can help one stay on top of:

- **Performance:** The main subject of this whitepaper, monitoring how the database performs can help detect bottlenecks and other issues before catastrophe strikes. Beyond helping one avert emergencies, performance metrics can assist one in deciding whether a performance increase is necessary or workable. For instance, by keeping a track record of query execution times, one could spot any suboptimal SQL statements and come up with improvements.
- **Growth:** Observe requirements in terms of users and traffic. Database usage needs to evolve faster than expected.
- **Security:** Ensure that one applied adequate security measures.

DATABASE METRIC TYPES

Before going through identifying metrics to follow, perhaps one should start at the beginning and ask, What are metrics?

Metrics capture a value about the systems at a specific point in time. An example is the number of users logged into the database.

Therefore, systems collect metrics at regular intervals (such as once per second and one per minute), to monitor a system.

There are two essential categories of metrics: those that are most useful in identifying problems and those whose primary value is in investigating issues. This whitepaper covers what data to collect. So, one can:

1. Recognize potential issues before they occur.
2. Investigate and understand performance issues.

Beyond metrics, there are other types of database monitoring that this whitepaper does not address. These include tracking events and security.

HOW OFTEN SHOULD ONE PERFORM MONITORING?

How often one monitors different aspects of the database depends on how mission-critical it and the applications it supports are. A failure or disruption may cause a severe impact on the business operations and organization, or perhaps even result in catastrophe. Then, one needs to be on top of both performance and security issues at all times. One can reduce the need to monitor continually the performance dashboard to a weekly inspection. Accomplish this by setting up alerts to inform one of the critical issues in real-time.

Specialized tools that provide real-time and periodic alerts should monitor database performance metrics. Real-time alerts are a must for mission-critical databases and databases with sensitive information susceptible to attack. So, one can take care of urgent issues as soon as they occur. Real-time preventive measures can protect the database from certain types of attacks, even before one has time to respond.

The database administrators, Information Technology operations staff, and users shared responsibility in performance monitoring since some factors that affect database performance lie beyond the database itself. It also makes sense to include some application developers informed. So, they can investigate the application side of things.

Database administrators need not monitor the applications that interact with the database. However, they must possess a general understanding of how developers implement applications and the architecture of these applications.

PERFORMANCE METRIC CATEGORIES

The previous section described the two main uses of database metrics: problem identification and problem investigation. Likewise, there are two essential categories of metrics that pertain to performance: work metrics and resource metrics. For each system that is part of the software infrastructure, consider which work metrics and resource metrics apply and available. Also, collect whatever one can. One need not monitor every metric. However, some metrics may play a more significant role once one identifies performance issues (that is, during problem investigation).

The next two sections cover each of the two performance metric types in more detail.

WORK METRICS

Work metrics gauge the top-level health of the database by measuring its useful output. One may break down work metrics into four subtypes:

- **Throughput:** The amount of work the system is doing per unit of time. Systems usually record throughput as an absolute number. Examples include the number of transactions or queries per second.
- **Success:** Represents the percentage of work that the system executed successfully (that is, the number of successful queries).
- **Error:** Captures the number of erroneous results, usually expressed as a rate of errors per unit of time. This yields errors per unit of work. The system often captured error metrics separately from success metrics when there are several potential sources of error, some of which are more serious or actionable than others.
- **Performance:** Quantifies how efficiently a component is doing its work. The most common performance metric is latency. Latency represents the time required to complete a unit of work. One can express latency as an average and as a percentile, such as 99% of requests returned within 0.1s.

The above metrics provide high-level but telling data that can help one answer the most critical questions about the internal health and performance of a system. That is to say:

- Is the database available and doing what we designed it to do?
- How fast is it producing work?
- What is the quality of that work?

RESOURCE METRICS

Resources are hardware, software, and network components that the database requires to perform its job. Some resources are low level, such as physical components like CPU, memory, disks, and network interfaces. One can also consider higher-level resources such as the query cache and database waits as a resource and therefore monitor these higher-level resources.

Resource metrics are useful in helping one reconstruct a detailed picture of the state of the database, making them valuable for investigation and diagnosis of problems. Resource metrics cover four key areas:

1. Utilization: The percentage of time that the database is busy or the percentage of the capacity of the database in use.
2. Saturation: A measure of the amount of requested work that the database cannot yet service and waits in the queue.
3. Errors: Represents internal errors that may or may not be observable in the database's output.
4. Availability: Denotes the percentage of time that the database responded to requests.

MONITORING PERFORMANCE

Both work and resource metrics include two other types of metrics:

1. Work metrics:
 - Database, transaction or query throughput
 - Query execution performance
2. Resource metrics:
 - Connections
 - Buffer pool usage

BEST WORK METRICS TO MONITOR

THROUGHPUT

Throughput measures the speed of a database system. Systems typically express throughput as the number of transactions per second. Consider the following differences:

- Write transactions versus read transactions
- Sustained rates versus peak rates
- A 10-byte row versus a 1000-byte row

Because of these differences, it is best to measure:

- Database throughput (for the database)
- Transaction throughput (for any operation)
- Query throughput (for query execution)

Throughput metrics in MySQL

MySQL provides throughput metrics for all the above transaction types.

The questions and queries status variables

There are two general MySQL status variables for measuring query execution: questions and queries. Of the two, the client-centric view provided by the questions metric makes it easier to interpret than the queries counter. The latter also counts statements executed as part of stored programs, and commands such as PREPARE and DEALLOCATE PREPARE that run as part of server-side prepared statements.

```
SHOW GLOBAL STATUS LIKE "Questions"
Variable_name  Value
-----
Questions      66
SHOW GLOBAL STATUS LIKE "Queries";
Variable_name  Value
-----
Queries        149
```

One can also monitor the breakdown of read and write commands to understand the workload of the database better and identify potential bottlenecks. The com_select metric captures read queries. Writes increment one of three status variables, depending on the statement type:

```
Writes = Com_insert + Com_update + Com_delete
```

```
SHOW GLOBAL STATUS LIKE "Com_select";
Variable_name  Value
-----
Com_select     49
```

However, when are MySQL counters incremented? The MySQL documents list all the various counter variables. However, they do not describe when the system incremented each counter. That might seem like a trivial point. However, it is not trivial, especially if one captures metrics with high resolution to diagnose MySQL performance incidents.

For instance, one could count queries when they start. Then, a spike in the number of queries in a certain second could be because of an increase in traffic. However, if one measure queries at the completion, then some critical resources becoming available (which allow many queries to complete) can cause spikes. Such spikes often occur with table-level locks and row-level locks on InnoDB.

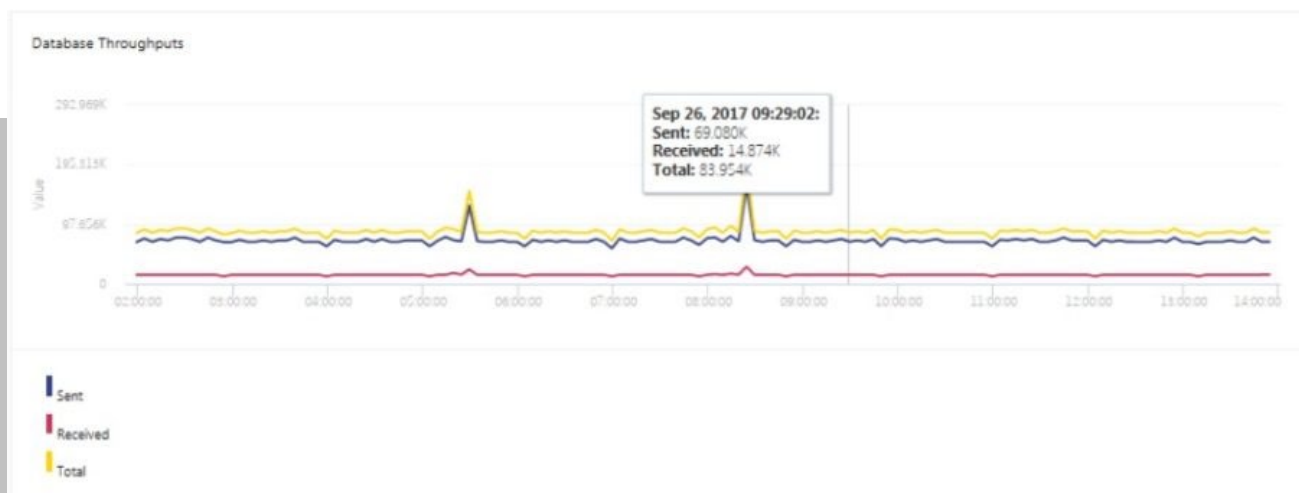
MySQL increments the questions and queries counters before executing the query. One may see a very uniform rate of questions when the system started many queries, but the system did not complete quickly because of waiting on some resource.

The threads_running status variable

It is instructive to look at the threads_running status variable to check for unusual numbers of queries running concurrently and struggling to complete in time.

```
SHOW GLOBAL STATUS LIKE "Threads_running";  
Variable_name  Value  
-----  
Threads_running 29
```

A professional monitoring tool can present throughput metrics as a graph to make peaks and valleys clearer:



Database throughputs



Transaction throughput

Measuring query execution performance is all about finding those queries that take too long to identify the required data or bring the data back. One of the best metrics to gauge query speed is latency. Latency is the time it takes a query to execute and return a result set. Latency is the time to make one round trip.

MySQL provides a few options for monitoring query latency, including built-in metrics and the performance schema. Enabled by default since version 5.6.6 of MySQL, the tables of the performance_schema database within MySQL store low-level statistics about server events and query execution.

The slow_queries server variable

It stores the number of queries that took more than long_query_time seconds. What is great about this counter is that it increments regardless of whether the system enabled the slow query log. That is a good thing because the system disabled the slow query log by default because logging can place a bit of a drag on performance.

```
SHOW GLOBAL STATUS LIKE "Slow_queries";
Variable_name  Value
-----
Slow_queries   99
```

The `events_statements_summary_by_digest` table of the performance schema contains a good deal of key metrics. That table captures information about query volume, latency, errors, time spent waiting for locks, and index usage. These metrics and more are available for each SQL statement executed. The system presents statements in a normalized form. The normalized form means that the system removes data values from the SQL and that the system standardized white space.

This query finds the top 10 statements by longest average run time:

```
SELECT substr(digest_text, 1, 50) AS digest_text_start
      , count_star
      , avg_timer_wait
  FROM performance_schema.events_statements_summary_by_digest
 ORDER BY avg_timer_wait DESC
LIMIT 10;
digest_text_start  count_star      avg_timer_wait
-----
SHOW FULL TABLES FROM `sakila`          11110825767786
SHOW GLOBAL STATUS LIKE ? 11038069287388
SELECT `digest_text`, `count_star`, `avg_timer_w1945742257586
SHOW FIELDS FROM `sakila` . `actor` 1611721261340
SELECT `digest_text`, `count_star`, `avg_timer_w          2335116484794
SHOW FIELDS FROM `sakila` . `actor_info` SELECT `a          1221773712160
SELECT NAME , TYPE FROM `mysql` . `proc` WHERE `Db          2148939688506
SHOW FIELDS FROM `vehicles` . `vehiclemodelyear`          1144172298718
SHOW SCHEMAS 2132611131408
SHOW FIELDS FROM `sakila` . `customer`          199954017212
```

Performance schemas display event timer information in picoseconds (that is, trillionths of a second) to present timing data in a standard unit. In the following example, the system divides `TIMER_WAIT` 1,000,000,000,000 to convert time data into seconds. The system also truncates values to six decimal places:

```

SELECT substr(digest_text, 1, 50) AS digest_text_start
      , count_star
      , TRUNCATE(avg_timer_wait/1000000000000,6)
FROM performance_schema.events_statements_summary_by_digest
ORDER BY avg_timer_wait DESC
LIMIT 10;
digest_text_start      count_star  avg_timer_wait
-----
SHOW FULL TABLES FROM `sakila`          11.110825
SHOW GLOBAL STATUS LIKE ? 11.038069
SELECT `digest_text`, `count_star`, `avg_timer_w          10.945742
etc.

```

Now one can see that the longest query took a little over one second to run.

The sys schema

Rather than write SQL statements against the performance schema, it is easier to use the sys schema. It contains interpretable tables for inspecting the performance data.

The sys schema comes installed with MySQL, starting with version 5.7.7. However, users of earlier versions can also install it. For instance, to install the sys schema on version 5.6 of MySQL, run the following commands:

```

git clone https://github.com/mysql/mysql-sys.git
cd mysql-sys/
mysql -u root -p < ./sys_56.sql

```

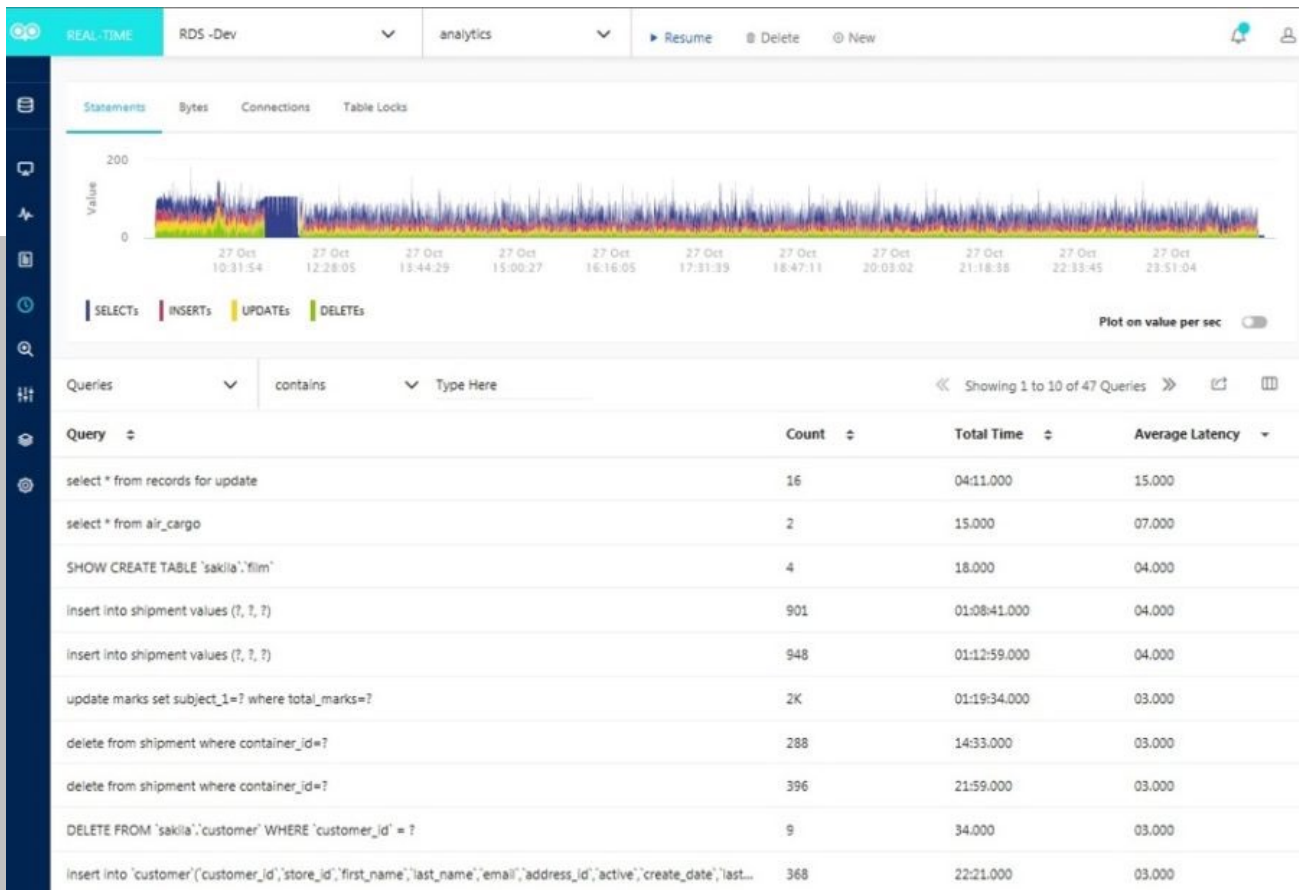
The sys schema provides an organized set of metrics in a more human-readable format, making the corresponding queries much simpler. For instance, to find the slowest statements (for example, those in the 95th percentile by runtime), run the following query:

```

SELECT * FROM sys.statements_with_runtimes_in_95th_percentile;

```

Here again, a professional monitoring tool can pay dividends by merging various performance metrics into one cohesive view:



Query execution performance

BEST RESOURCE METRICS TO MONITOR

CONNECTIONS

Connection manager threads handle client connection requests on the network interfaces to which the server listens. On all platforms, one manager thread handles transmission control protocol and Internet protocol (TCP/IP) connection requests. Connection manager threads

associate each client connection with a thread dedicated to it that handles authentication and request processing for that connection. Manager threads create a new thread when necessary. However, try to avoid doing that by consulting the thread cache first to see whether it contains a thread that it can use for the connection. When a connection ends, the system returns its thread to the thread cache (that is, if the cache is not full). In this connection thread model, there are as many threads as there are clients connected.

It is essential to monitor the client connections because the system refuses new client connections once the database server runs out of available connections.

The MySQL connection limit defaults to 151. However, one can change it using the SET statement. So, it is best not to assume anything. The @@max_connections variable stores the connection limit:

```
SELECT @@max_connections;
+-----+-----+
| Variable_name | Value |
+-----+-----+
| max_connections | 151 |
+-----+-----+
```

Set the connection limit like so:

```
SET GLOBAL max_connections = 200;
```

To set the connection limit such that it persists once the server restarts, add a line like this to the 'my.cnf' configuration file:

```
max_connections = 200
```

Do not hesitate to increase the number of max_connections. Per the MySQL documents, production servers should be able to handle connections in the high hundreds or thousands. Just remember there are some caveats when the server must handle many connections. For instance, thread creation and disposal become expensive when there are a lot of them. Also, each thread

requires server and kernel resources, such as stack space. Therefore, to accommodate many simultaneous connections, one needs to keep the stack size per thread small. That small stack can lead to a situation where the stack size is too small, or the server consumes large amounts of memory.

The takeaway here is that the database server should possess adequate amounts of processing power and memory to accommodate a large user base.

MySQL provides a few good metrics for monitoring the connections:

Variable	What it represents	Why one should monitor it
threads_connected	The total number of clients that possess open connections to the server.	Provides real-time information on how many clients connect to the server. That can help in analyzing traffic and in deciding the best time for a server restart.
threads_running	The number of threads that are not sleeping.	Good for isolating which connected threads are processing queries, as opposed to connections that are open but are idle.
connections	The number of connection attempts (that is, whether successful) to the MySQL server.	Can give one a good idea of how many people and applications are accessing the database. Over time, these numbers reveal busiest times and average usage numbers.
connection_errors_internal	The number of connections refused because of internal server errors (that is, failure to start a new thread or an out-of-memory condition).	Although MySQL exposes several metrics on connection errors, Connection_errors_internal is the most useful because it only increments when the error comes from the server itself. Internal errors can show an out-of-memory condition or an inability to start a new thread.

We can use the MySQL show status command to show MySQL variables and status information. Here are a few examples:

```
SHOW GLOBAL STATUS LIKE '%Threads_connected%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Threads_connected | 2    |
+-----+-----+
```

```
SHOW GLOBAL STATUS LIKE '%Threads_running%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Threads_running | 1    |
+-----+-----+
```

```
SHOW GLOBAL STATUS LIKE 'Connections';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Connections    | 20    |
+-----+-----+
```

Aborted client and connections

Every time a client cannot connect, the server increments the Aborted_connects status variable. Unsuccessful connection attempts can occur because:

- A client attempts to access a database but does not possess privileges for it.
- A client uses an incorrect password.
- A connection packet does not contain the right information.

It takes more than `connect_timeout` seconds to get a connect packet.

If these kinds of things happen, then it might show that someone is trying to break into the server. If the system enabled the general query log, then it logs messages for these types of problems.

If a client connects but later disconnects improperly or the system ended the connection, then the server increments the `aborted_clients` status variable. The server then logs an 'Aborted connection' message to the error log.

Here is how to view the number of aborted clients and connections:

```
mysql> SHOW GLOBAL STATUS LIKE 'Aborted_c%';
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| Aborted_clients        | 3     |
| Aborted_connects      | 8     |
+-----+-----+
```

Connection errors

MySQL does an outstanding job of breaking down metrics on connection errors into different status variables:

```
SHOW GLOBAL STATUS LIKE 'Connection_errors%';
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| Connection_errors_accept | 0     |
+-----+-----+
```

```

| Connection_errors_internal      | 0      |
+-----+-----+
| Connection_errors_max_connection | 0      |
+-----+-----+
| Connection_errors_peer_address  | 0      |
+-----+-----+
| Connection_errors_select        | 0      |
+-----+-----+
| Connection_errors_tcpwrap      | 0      |
+-----+-----+

```

Once all available connections are in use, attempting to connect to MySQL causes it to return a 'Too many connections' error and increment the `Connection_errors_max_connections` variable. The best bet in preventing this scenario is to monitor the number of open connections and ensure that it remains safely below the configured `max_connections` limit.

Fine-grained connection metrics such as `Connection_errors_max_connections` and `Connection_errors_internal` can help to pinpoint the source of the problem. The following statement fetches the value of `Connection_errors_internal`:

```

SHOW GLOBAL STATUS LIKE 'Connection_errors_internal';
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| Connection_errors_internal | 2     |
+-----+-----+

```

Here are two SQL Diagnostic Manager for MySQL screens that monitor Current Connections and the Connection History, respectively:

MONITOR GROUP	MONITORS	Master	Slave
Current Connections	Max allowed	4.88K	100
	Open connections	40	9
	Connection usage	0.60%	9.00%
	Currently running threads	1	1
	Highest no. of concurrent connections	114	12
	Idle time after which a client is disconn...	8 hrs	8 hrs
	Max number of interrupts before host is...	100	10
	Connect timeout	10 secs	10 secs
	Back log	900	50

SQL Diagnostic Manager for MySQL monitor: Current connection

MONITOR GROUP	MONITORS	Master	Slave
Connection History	Attempts	34.25K (0.099/sec)	255 (0.002/sec)
	Successful	32.35K (0.093/sec)	221 (0.001/sec)
	Percentage of max allowed reached	2.28%	12.00%
	Refused	1.91K (0.005/sec)	34 (0.000/sec)
	Percentage of refused connections	5.57%	13.33%
	Terminated abruptly	1.33K (0.004/sec)	103 (0.001/sec)
	Bytes received from all clients	539.08M (1.550K/sec)	9.20M (60.926/sec)
	Bytes sent to all clients	27.43G (80.771K/sec)	196.72M (1.273K/sec)

SQL Diagnostic Manager for MySQL monitor: Connection history

BUFFER POOL USAGE

The default storage engine of MySQL, InnoDB, uses a special storage area called the buffer pool to cache data for tables and indexes. The system categorizes buffer pool metrics as resource metrics. Their main value is in the investigation rather than the detection of performance issues.

Configuring the buffer pool

You can configure various aspects of the InnoDB buffer pool to improve performance.

The buffer pool defaults to a small 128MB., one should increase the size of the buffer pool to as high a value as is practical. At the same time, one should leave sufficient memory for other processes on the server to run without excessive paging. That reserved memory typically amounts to about 80% of physical memory on a dedicated database server. The larger the buffer pool, the more InnoDB acts like an in-memory database, reading data from disk once and then accessing the data from memory during subsequent reads.

Please note:

- The memory overhead of InnoDB can increase the memory footprint by about 10 percent beyond the allotted buffer pool size.
- Once the system exhausted the physical memory, the system resorts to paging. Performance suffers. Hence, database performance degrades while the disk input and output rises. Then, it might be time to expand the buffer pool.

The system resizes the buffer pool operations in chunks. Also, one must set the size of the buffer pool to a multiple of the chunk size times the number of instances:

```
innodb_buffer_pool_size = N * innodb_buffer_pool_chunk_size
                        * innodb_buffer_pool_instances
```

The chunk size defaults to 128MB but is configurable as of version 5.7.5 of MySQL. Check the value of both parameters:

```
SHOW GLOBAL VARIABLES LIKE "innodb_buffer_pool_chunk_size";  
SHOW GLOBAL VARIABLES LIKE "innodb_buffer_pool_instances";
```

If querying `innodb_buffer_pool_chunk_size` returns no results, the parameter is not tunable in the version of MySQL. One can assume the parameter to be 128MB.

To set the buffer pool size and the number of instances at server startup, invoke `mysqld.exe` with the following parameters:

```
$ mysqld --innodb_buffer_pool_size=8G  
--innodb_buffer_pool_instances=16
```

As of version 5.7.5 of MySQL, one can also resize the buffer pool on-the-fly via a `SET` command specifying the desired size in bytes. For instance, with two buffer pool instances, one could set each to 4GB by setting the total size to 8GB:

```
SET GLOBAL innodb_buffer_pool_size=8589934592;
```

Important InnoDB buffer pool metrics

One can access InnoDB Standard Monitor output using `SHOW ENGINE INNODB STATUS`. That output provides several metrics about operating the InnoDB buffer pool under the `BUFFER POOL AND MEMORY` section. Here is some typical content:

```
-----  
BUFFER POOL AND MEMORY  
-----
```

```
Total large memory allocated 2198863872  
Dictionary memory allocated 776332  
Buffer pool size 131072  
Free buffers 124908  
Database pages 5720  
Old database pages 2071  
Modified db pages 910  
Pending reads 0  
Pending writes: LRU 0, flush list 0, single page 0  
Pages made young 4, not young 0  
0.10 youngs/s, 0.00 non-youngs/s  
Pages read 197, created 5523, written 5060  
0.00 reads/s, 190.89 creates/s, 244.94 writes/s  
Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not  
0 / 1000  
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read  
ahead 0.00/s  
LRU len: 5720, unzip_LRU len: 0  
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
```

A word about the InnoDB buffer pool LRU algorithm

To better understand what the above metrics mean, one should review how the InnoDB Buffer Pool LRU Algorithm works.

InnoDB manages the buffer pool as a list, using a variation of the least recently used (LRU) algorithm. When the system needs space to add a new page to the pool, InnoDB evicts the least recently used page and adds the new page to the middle of the list. This 'midpoint insertion strategy' treats the list as two sublists:

1. At the head, a sublist of 'new' (or 'young') pages that the system accessed recently.
2. At the tail, a sublist of 'old' pages that the system accessed less recently.

This algorithm keeps pages that the system uses heavily by queries in the new sublist. The old sublist contains less-used pages. Such less-used pages are candidates for eviction.

With that in mind, here are more critical fields in the InnoDB Standard Monitor output:

- Old database pages: The number of pages in the old sublist of the buffer pool.
- Pages made young, not young: The number of old pages that the system moved to the head of the buffer pool (that is, the new sublist). Also, the number of pages that remained in the old sublist without being made new.
- Youngs per second non-youngs per second: The number of accesses to old pages that resulted in making them young or not. This metric differs from that of the previous item in two ways. First, it relates only to old pages. Second, the system bases it on the number of accesses to pages and not the number of pages. There can be multiple access events to a page. The system counts these access events.
- Young-making rate: Hits that cause blocks to move to the head of the buffer pool.
- Not: Hits that do not cause blocks to move to the head of the buffer pool.

The young-making rate and not rate rarely add up to the overall buffer pool hit rate.

Some MySQL performance tuning tips

One may see low values of youngs per second without performing large table scans. That may be a sign that one needs to reduce the delay time for a block that the system moves from the old to the new sublist. Alternatively, one may need to increase the percentage of the buffer pool used for the old sublist.

If one does not see many non-youngs per second while performing large table scans (and lots of youngs per second), then try tuning the delay value to be higher.

The `innodb_old_blocks_time` global variable specifies how long in milliseconds (ms) a page inserted into the old sublist must stay there after its first access event before the system can move it to the new sublist. If the value is 0, then a page inserted into the old sublist moves to the new sublist the first time the system accesses it. That occurs no matter how soon the access event occurs after insertion. If the value is greater than 0, then pages remain in the old sublist until an access event occurs at least that many milliseconds after the first access. For example, a value of 1000 causes pages to stay in the old sublist for 1 second after the first access event before they become eligible to move to the new sublist.

The following statement sets the `innodb_old_blocks_time` to zero:

```
SET GLOBAL innodb_old_blocks_time = 0;
```

The `innodb_old_blocks_pct` global variable specifies the approximate percentage of the buffer pool that InnoDB uses for the old block sublist. Increasing the old sublist percentage makes it larger. So, blocks in that sublist take longer to move to the tail, and the system evicts them. That increases the likelihood that the system accesses them again and makes them young. The range of values is 5 to 95. The default value is 37 (that is, $\frac{3}{8}$ of the pool).

When scanning small tables that fit into memory, there is less overhead for moving pages around within the buffer pool. So, one can leave `innodb_old_blocks_pct` at its default value, or even higher, such as `innodb_old_blocks_pct=50`.

There are many other global status variables one can examine besides `innodb_old_blocks_time` and `innodb_old_blocks_pct`:

```
SHOW GLOBAL STATUS LIKE 'Innodb_buffer_pool%';
```

```
Variable_nameValue
```

```
-----  
Innodb_buffer_pool_dump_status, not started  
Innodb_buffer_pool_dump_status          not started  
Innodb_buffer_pool_load_status           not started  
Innodb_buffer_pool_pages_data460  
Innodb_buffer_pool_bytes_data7536640  
Innodb_buffer_pool_pages_dirty          0  
Innodb_buffer_pool_bytes_dirty          0  
Innodb_buffer_pool_pages_flushed1  
Innodb_buffer_pool_pages_free7730  
Innodb_buffer_pool_pages_misc2  
Innodb_buffer_pool_pages_total          8192  
Innodb_buffer_pool_read_ahead_rnd0  
Innodb_buffer_pool_read_ahead0  
Innodb_buffer_pool_read_ahead_evicted    0  
Innodb_buffer_pool_read_requests15397  
Innodb_buffer_pool_reads461  
Innodb_buffer_pool_wait_free0  
Innodb_buffer_pool_write_requests1
```

Of these, some metrics are more useful to one than others. Standouts include:

- Metrics tracking the total size of the buffer pool
- How much is in use
- How effectively the buffer pool is serving reads

The metrics `innodb_buffer_pool_read_requests` and `Innodb_buffer_pool_reads` are integral to gauging buffer pool utilization. `Innodb_buffer_pool_read_requests` are the number of requests to read a row from the buffer pool. `Innodb_buffer_pool_reads` is the number of times InnoDB needs to perform read data from disk to fetch required data pages. Reading from memory is much faster than reading from a disk. So, keep an eye out for increasing `Innodb_buffer_pool_reads` numbers.

One can calculate the buffer pool efficiency using the formula:

$$\text{innodb_buffer_pool_reads} / \text{innodb_buffer_pool_read_requests} * 100 = 0.001$$

Here is an example:

```
mysql> SHOW GLOBAL STATUS LIKE 'innodb_buffer_pool_rea%';
Variable_name      Value
-----
Innodb_buffer_pool_read_requests | 2905072850 |
Innodb_buffer_pool_reads          | 1073291394 |
```

To calculate the InnoDB buffer pool efficiency:

$$(107329139 / 2905072850 * 100) = 37$$

Here, InnoDB is doing more disk reads. So, the InnoDB buffer pool is not sufficiently large.

Buffer pool utilization is another useful metric to check. The utilization metric is not available. However, one can calculate it:

$$\frac{(\text{Innodb_buffer_pool_pages_total} - \text{Innodb_buffer_pool_pages_free})}{\text{Innodb_buffer_pool_pages_total}}$$

Here is an example:

```
SHOW GLOBAL STATUS LIKE 'Innodb_buffer_pool_pages%';
Variable_name      Value
-----
Innodb_buffer_pool_pages_data460
Innodb_buffer_pool_pages_dirty      0
Innodb_buffer_pool_pages_flushed1
Innodb_buffer_pool_pages_free7730
Innodb_buffer_pool_pages_misc2
Innodb_buffer_pool_pages_total      8192
```

Enter the numbers into our formula:

$$(8192 - 7730) / 8192 = 0.056396484375$$

We can convert that into a percentage by multiplying by 100:

$$0.056396484375 * 100 = 5.64\% \quad (\text{Quite low!})$$

That the database is serving many reads from disk while the buffer pool is near empty is not a cause for celebration. Perhaps the system cleared the cache, and it is still refilling. However, should this condition continue for an extended amount of time, it is likely that there is plenty of memory to accommodate the dataset.

High buffer pool utilization is not necessarily a bad thing either, as long as the system ages the old data out of the cache according to the LRU policy.

Only when read operations are overpowering the buffer pool, should one think seriously about scaling up the cache.














Converting buffer pool metrics to bytes

The system reports most of the buffer pool metrics as a count of memory pages. That is not all that useful. It is possible to convert page counts to bytes. Using bytes makes it a lot easier to determine the actual size of the buffer pool. For instance, this simple formula gives us the total size of the buffer pool in bytes:

```
Innodb_buffer_pool_pages_total * innodb_page_size
```

One can retrieve the `innodb_page_size` using a `SHOW VARIABLES` query:

```
SHOW VARIABLES LIKE "innodb_page_size"
```

MONITORS		Master	
Percentage of full table scans	 	99.6%	 
Buffer for full table scans (per client)		128K	
SELECTs requiring full table scan		4.74M (13.951/sec)	
Buffer for joins requiring full table scan ...		256K	
Joins requiring full scan of second and ...		704.35K (2.026/sec)	
Joins that reevaluate index selection fo...		0	

SQL Diagnostic Manager for MySQL offers the most useful buffer pool metrics at a glance

CONCLUSION

This whitepaper presents the top MySQL performance metrics. In particular, this whitepaper showed that:

- It is crucial to monitor them effectively to keep the databases and applications running smoothly.
- There are two essential categories of metrics: those that are most useful in identifying problems and those whose primary value is in investigating problems.

- How often one monitors different aspects of the database depends on how mission-critical it and the applications it supports are.
- There are two essential categories of metrics that pertain to performance: work metrics and resource metrics.
- Both work and resource metrics include two types of metrics:
 - Work metrics:
 - Throughput
 - Query execution performance
 - Resource metrics:
 - Connections
 - Buffer pool usage
- There are two general MySQL status variables for measuring query execution: questions and queries. Of the two, the client-centric view provided by the questions metric makes it easier to interpret than the queries counter.
- MySQL provides a few options for monitoring query latency, including its built-in metrics and performance schema. The tables of the performance_schema database within MySQL, enabled by default since version 5.6.6 of MySQL, store low-level statistics about server events and query execution.
- The events_statements_summary_by_digest table of the performance schema contains a good deal of vital metrics. That table captures information about query volume, latency, errors, time spent waiting for locks, and index usage.
- The sys schema contains interpretable tables for inspecting the performance data.

Further, this whitepaper focuses on database connections and buffer pool metrics. This whitepaper showed how to capture and study MySQL metrics that provide the most value for the effort. The main points covered include:

- The connection manager handles client connection requests on the network interfaces to which the server listens.
- It is essential to monitor the client connections because (once the database server runs out of available connections) the system refuses new client connections.

- Every time a client cannot connect, the server increments the `Aborted_connects` status variable.
- Fine-grained connection metrics such as `Connection_errors_max_connections` and `Connection_errors_internal` can help to pinpoint the source of connection problems.
- The default storage engine of MySQL, InnoDB, uses a particular storage area called the buffer pool to cache data for tables and indexes.
- The system categorizes buffer pool metrics as resource metrics.
- One can configure various aspects of the InnoDB buffer pool to improve performance.
- InnoDB Standard Monitor output provides several metrics about operating the InnoDB buffer pool.
- The LRU algorithm uses a midpoint insertion strategy that treats the pages as old and new sublists.
- One can tune the LRU algorithm using the `innodb_old_blocks_time` and `innodb_old_blocks_pct` global variables.

By tracking the most useful metrics and reviewing them in the most informative ways, one balances over-monitoring and unforeseen firefighting crises.

ABOUT THE AUTHOR

Shree Nair is Director of Technology Partnerships at IDERA. He has an exceptional track record in innovation, partner management, solution development, and global team leadership. Shree is the former product manager of Webyog's MySQL tools. Under his direction, Webyog performed a complete overhaul of the product and business systems to target a wide range of industries worldwide and expand the customer base beyond 30,000 customers. Shree believes monitoring MySQL should be easy. He seeks opportunities to make MySQL user's life easy. He earned a Master's at Coventry University, England.

SQL DIAGNOSTIC MANAGER FOR MYSQL

With [SQL Diagnostic Manager for MySQL](#), monitor MySQL and MariaDB performance in real-time. This powerful tool helps database administrators pinpoint the cause of MySQL performance problems in physical, virtual, and cloud environments.

Proactively find and fix MySQL performance problems:

- Improve performance by optimizing bad SQL queries.
- Gain visibility into overall health and performance.
- Alert proactively on potential performance problems.
- Take action before MySQL powered systems run out of resources.
- Get a high ROI with increased DBA productivity and server performance.

“SQL Diagnostic Manager for MySQL is very intuitive, and it makes database administration easy.”
Olu Efonwoye, Database Administrator, ICF Corporation (Small Business, Telecommunications Services, USA, 10 to 24 MySQL databases)

Unlike its competition, SQL Diagnostic Manager for MySQL provides:

- Agentless monitoring with no additional load on servers
- Over 600 monitors and advisors
- Custom dashboards, charts, and monitors
- Real-time tracking of locked and long-running SQL queries
- Display of top 10 problematic SQL queries across servers
- Monitoring and comparison of configuration changes
- File-based log monitoring for Amazon RDS for MySQL

Start for FREE

