Detecting/ Monitoring MySQL Server Issues

Use the InnoDB database engine. The old, non-transactional MyISAM engine is now deprecated

Monitoring MySQL: metrics and alerts

The rule of thumb :

- collect all possible/reasonable metrics that can help when troubleshooting,
- alert only on those that require an action from you.

Tweak the threshholds for the needs

MySQL process running

Metric	Comments	Suggested Alert
mysqld process count	Right binary daemon process running	When process count /usr/sbin/mysqld != 1

The top system resources to monitor on a database server

When you experience any issue or bottleneck, these are the first you need to have a look at.

Metric	Comments	Suggested Alert
Load	All in one high performance metric	When load is > factor x (number of cores).Our suggested factor is 4.
Swap usage	Emergencies only	When used, swap is > 128 MB
Disk usage	Make sure you always have free space for new data, temporary files, snapshot or backups.	When database, logs and temp is > 85% usage.
CPU Usage	A high CPU usage is not a	None

	bad thing as long as you don't reach the limit.	
Memory usage	Ideally your entire database should be stored in memory, but this is not always possible. Give MySQL as much as you can afford but leave enough for other processes to function.	None
Network bandwidth	Unless doing backups or transferring huge amounts of data, it shouldn't be the bottleneck.	None

Disk is one of the most common bottlenecks, so it's worth keeping an eye on some detailed metrics here. You can get those with iostat (for more info on iostat check out Monitoring IO performance using iostat & pt-diskstats).

Metric	Comments	Suggested Alert
Read/Write requests	IOPS (Input/Output operations per second)	None
IO Queue length	Tracks how many operations are waiting for disk access. If a query hits the cache, it doesn't create any disk operation. If a query doesn't hit the cache	None

	(i.e. a miss), it will create multiple disk operations.	
Average IO wait	Time that queue operations have to wait for disk access.	None
Average Read/Write time	Time it takes to finish disk access operations (latency).	None
Read/Write bandwidth	Data transfer from and towards your disk.	None

MySQL Metrics

Monitoring MySQL availability and connections

Let's start with the metrics that establish if your MySQL server is working.

Metric		Comments Suggested	Alert
Uptime	Seconds since the server was started. We can use this to detect respawns.	When uptime is < 180.	
Threads_connected	Number of clients currently connected. If		
none or too high, something is wrong.	None		

Max_used_connections	Max number of connections at a time		
since server started. (max_used_connections / max_connections) indicates if you could run out soon of connection slots.	When connections usage is > 85%.		
Aborted_connects	Number of failed connection attempts. When growing over a period of time either some credentials are wrong or we are being attacked.	When aborted connects/min > 3. (only on not public exposed servers, otherwise will generate noise)	

MySQL typical errors

Common failure points you need to keep an eye on.

Metric	Comments Suggested	Alert
(Errors)	Are there any errors on the mysql.log file?	None
(Log files size)	Are all log files being rotated?	None

(Deleted log files)	Were any log files deleted but the file descriptor is still	
open?	None	
(Backup space)	Do you have enough disk space for backups?	None

Monitoring MySQL queries

These metrics track whether our database does what it's meant to do.

Answer queries that is (!):

Metric	Comments Suggested	Alert
Questions (/s)	Number of statements sent by clients.	None
Queries	Number of executed statements (including stored procedures)	None
Read / Writes	Reads: selects \+ cache hits Writes: inserts \+ updates \+ deletes	None

These metrics keep track of the queries that are affecting your server's performance:

Metric	Comments Suggested	Alert

Slow_queries	Number of queries that took more than long_query_time seconds to execute. Slow queries generate excessive disk reads, memory and CPU usage. Check slow_query_log to find them.	None
Select_full_join	Number of full joins needed to answer queries. If too high, improve your indexing or database schema.	None
Created_tmp_disk_tables	Number of temporary tables (typically for joins) stored on slow spinning disks, instead of faster RAM.	None
(Full table scans) Handler_read%	Number of times the system reads the first row of a table index. Sequential reads might indicate a faulty index.	None

Monitoring MySQL caches, buffers, and locks

Here are some key metrics for optimizing caches and buffers, whilst detecting any locked transactions. We would love to hear about your suggested metrics as well.

Metric	Comments Suggested	Alert

Innodb_row_lock_waits	Number of times InnoDB had to wait before locking a row.	None
Innodb_buffer_pool_wait_free	Number of times InnoDB had to wait for memory pages to be flushed. If too high, innodb_buffer_pool_size is too small for current write load.	None
Open_tables	Number of tables currently open. If this is low and table_cache is high, we can reduce cache size. If opposite, we should increase it. If you increase table_cache you might have to increase available file descriptors for the mysql user.	None
(Long running transactions)	Tracks whether too many transactions are locked by other idle transactions, or because of a problem in InnoDB.	None
(Deadlocks)	Deadlocks happen when 2 transactions mutually hold. These are unavoidable in InnoDB and apps should deal with them.	None

Command-line Tools

1. Mytop

Mytop is one of the classic open source and free
console-based (non-gui) monitoring tool for MySQL database
was written using Perl language.
Mytop runs in a terminal and displays
statistics about threads, queries, slow queries, uptime, load, etc.
in tabular format, much similar to the Linux top program.
Which indirectly helps the administrators to
optimize and improve performance of MySQl
to handle large requests and decrease server load.

2. Innotop

Innotop is a real time advanced command line based investigation program
to monitor local and remote MySQL servers running under InnoDB engine.
Innotop includes many features and comes with different types of
modes/options,
which helps us to monitor various aspects of MySQL performance
to find out what's wrong going with MySQL server.

3. MysqlAdmin

MysqlAdmin is a default command line MySQL client that comes pre-
installed with
MySQL package for performing administrative operations such as
monitoring processes, checking server configuration, reloading privileges,
current status, setting root password, changing root password, create/drop
databases,
and much more.

Advanced GUI Toolkits

1. [MySQL Database ZenPack] (https://www.zenoss.com/product/zenpacks/mysql-monitor)

2. [Percona Toolkit] (https://www.percona.com/software/database-tools/percona-toolkit)

3. Apache BenchMark

create a simple script that does nothing but execute the query to improve. Then, run AB on it with different concurrency settings. The benefit is to get an idea on how the query scales, not just how fast it runs. Who cares if a query is fast for a single run. What you really care about is how fast in runs with load.

So: <?php $my = new MySQLi($host, $user, $pass); $my->query('SELECT foo');

Then, using AB:

ab -c 10 -n 10000 http://localhost/path/to/my/test.php

Transactional & Locking Issues

Note:

The `MyISAM` engine doesn't support transactions
InnoDB engine Supports transactions, row-level locking, and foreign keys

`mysql> show engines;`

Engine	Support	Comment	Transactions	XA	Savepoints
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
MyISAM	DEFAULT	Default engine as of MySQL 3.23 with great performance	YES	NO	YES
InnoDB	YES	Supports transactions, row-level locking, and foreign keys	YES	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO

Assumption: Using InnoDB MySQL engine

1. Have a list of all active connections to our target database.

`Select * from information_schema.processlist;`

You may Filter the result set by using the DB and/or USER column.
eg;

`Select ID from information_schema.processlist where DB='MY_TARGET_DB_NAME';`

2. access to a log of all queries executed against a particular database.

The general log can either be written to a log file or to a table.
Prefer writing to a table as it is easier to filter the log based on one’s current needs

```
Select * from mysql.general_log;
```

```
Note: Prior to MySQL 5.1.6 the general log could only be outputted to log files.

General logging to a table can be enabled by executing the following queries:

```

```
SET GLOBAL log_output = 'TABLE';
SET GLOBAL general_log = 'ON';

```

3. Filter the general log using a database name.

```
SELECT
  *
FROM
mysql.general_log
WHERE
thread_id IN (SELECT
      ID
  FROM
      information_schema.processlist
  WHERE
      DB = 'YOUR_DB_NAME');

```

4. As general logging is very verbose and one will eventually want to disable the logging.

General logging can be disabled using the following query:

```
SET global general_log = 'OFF';
```

Deletion of existing log entries

```
TRUNCATE table mysql.general_log;

```

5. MySQL Active Transactions and Locks

Get list of active transactions and locks that are currently executing against our target database.

INNODB_TRX provides us with information on the currently executing transactions.
INNODB_LOCKS gives us information on current lock while the INNODB_LOCK_WAITS provides information on who is actually waiting for locks.

The query below enables one to see active locks filtered by a particular database:

```
SELECT
tw_ps.DB waiting_trx_db,
r.trx_id waiting_trx_id,
r.trx_mysql_thread_id waiting_thread,
r.trx_query waiting_query,
bt_ps.DB blocking_trx_db,
b.trx_id blocking_trx_id,
b.trx_mysql_thread_id blocking_thread,
b.trx_query blocking_query
FROM
information_schema.innodb_lock_waits w
    INNER JOIN
information_schema.innodb_trx b ON b.trx_id = w.blocking_trx_id
    INNER JOIN
information_schema.innodb_trx r ON r.trx_id = w.requesting_trx_id
    INNER JOIN information_schema.processlist tw_ps ON tw_ps.ID = r.trx_id
INNER JOIN information_schema.processlist bt_ps ON bt_ps.ID =       b.trx_id
WHERE  r.trx_id in (Select ID FROM information_schema.processlist where
DB='YOUR_DB_NAME') ;

```

References: http://abiasforaction.net/debugging-transactional-and-locking-issues-in-mysql/

Using MySQL Tuner

- MySQL Tuner is a Perl script designed to assist you with the configuration and
performance tuning of a MySQL database server.

- This script will examine your MySQL server and then report its findings.

- You can then make suggested changes to your server to increase performance.

Troubleshooting MySQL Performance Issues

Why Database Performance Slows Down

The most common reason for slow database performance is based on this "equation":

```
(number of users) x (size of database) x (number of tables/views) x
(number of rows in each table/view) x (frequency of updates) x (processor) x
(network latency)
```

Following a Monitoring Plan

A sound database monitoring plan can help you stay on top of:

- Performance: By tracking performance-related metrics like query completion times you could spot any
underperformers and come up with possible fixes.

- Growth: Observe changes in terms of users and traffic. Database usage needs can evolve very quickly and lead to gridlock.

- Security: Ensure that adequate security measures are applied.

Although following the above advice won't eliminate performance issues, it may provide a better heads-up to issues as they develop.

Some Root Causes of Performance Issues

1 Hardware/OS/Network-related causes:

- Limited amount of available disk space and/or memory on the server or clients' machines.

- Out of date/obsolete Network Interface Cards (NIC) or other server/client hardware.

- Inappropriate OS (operating system) for running server or client applications.

- All sorts of malware.

- Network issues, excessive network traffic, network bottlenecks, or database traffic.

2 Native and 3rd party application interference:

- Scheduled jobs – backup, maintenance, AV-scan, application and system updates.

- Other applications and (3rd party) software installed on the machine.

Remember that all programs launched at startup are always running in the background.

3 Database Design:

- Large number of views in a database with queries using expensive subqueries and
poor indexing of tables can use extensive memory and cause swapping.

- Some database properties can slow performance while others improve performance.
Be careful when choosing character sets and datatypes in particular.

- Configuration of server and application framework does not allow access to all system resources
or require more resources than available. Both will reduce performance.

- Make sure that there are no database admin utilities such as backup, synchronization,
or compression are running during peak hours.

- Large attachments (BLOBS) in databases.

- Inefficient coding/programming of custom applications. Often it will be possible have application logic
in the application or in Stored Procedures, Triggers etc. Choose the right approach for the task.
Database corruption and fragmentation (could also be paired with other symptoms including incorrect results
from some queries).
Consider to run some check/repair/defragmentation routine periodically.

Database-related Performance Issues

Flaws that are only exposed once a certain usage threshold has been reached eg;

- normalization of tables
- improper indexing
- poorly constructed queries

MySQL comes equipped with a couple of excellent tools to help get at the root cause of slow MySQL performance,
namely the

- `Slow Log` and `Performance Schema`.

The MySQL Slow Query Log

- The **most common** internal cause of database slowdowns are **queries that monopolise system resources**

- Factors that contribute to poor query performance include

    - inadequate indexing,
    - fetching a data set that is very large,
    - complex joins,
    - and text matching.

The slow query log identify `queries that need optimization`.
It consists of `SQL statements that took more than a certain number of seconds to execute`.
The default value of ten seconds is stored in the **long_query_time** system variable.

`mysql> select @@long_query_time`

```
+-----------------+

| @@long_query_time |

+-----------------+

|        10.000000 |

+-----------------+
```

Ten seconds is probably too long as a cut-off for most production settings. To change it, use the SET statement:

`mysql> SET @@long_query_time = 5`

You can also `log queries that require MySQL to examine more than a certain number of rows` to the slow query log

That value is defined in the *min_examined_row_limit* system variable.

It only includes the time to resolve the query and not the time to acquire locks. It defaults to 0.

`mysql> select @@ min_examined_row_limit;`

```
+-------------------------+
```

```
| @@ min_examined_row_limit |

+--------------------------+

|        0                 |

+--------------------------+

``
```

To modify it, use the SET statement. It can contain any Integer between 0 and 4294967295 on 32-bit platforms and
18446744073709551615 on 64-bit platforms:

`mysql> SET @@min_examined_row_limit = 500000;`

The slow query log is disabled by default.

This is because logging can place a bit of a drag on performance. Therefore, it's best to enable the Slow Query log (preferably from MySQL configuration file) for a short period of time, e.g. 2-3 days, when your application performance is reduced for some reason and you wish to detect the slow queries.

You can check if the MySQL slow query log is enabled using the following statement
SHOW VARIABLES Like 'slow_query_log%';

Steps to enable the Slow Query Log:

1.Shut down any applications that are using MySQL.
2.Shut down MySQL itself.
3.Add the following configuration options to `my.ini` or `my.cnf`

```
 # What's the threshold for a slow query to be logged?

long_query_time=0.5

# Where should the queries be logged to?

slow_query_log_file=/path/to/logfile

# Enable slow query logging - note the dashes rather than underscores:

slow-query-log=1
```

1.Save the file, and restart MySQL.
2.Restart your connected applications.

Viewing the Slow Query Log

MySQL provides a tool called _mysqldumpslow_ which can be used to analyze the log file.

The following syntax will show you the top 10 queries sorted by average query time
(Remember to update the file names and paths to suit your environment)

```
mysqldumpslow -t 10 mysql-slow-query.log > mysqldumpslow.out
```

Here is some sample output:

```
    Reading mysql slow query log from /usr/local/mysql/data/mysqld51-apple-slow.log

    Count: 90  Time=56.73s (6183s) Lock=0.00s (0s)  Rows=0.0 (0), root[root]@localhost

    select * from t1
```

The important values to look at here are the __Count__ and the __Time__.
The __Count__ is the number of times this query ran within your log set.
The __Time__ is an average amount of time for each of those queries runs to complete.
With the number in parentheses, in this case 6183s, being the total (Count x Time) amount of time spent on running this query.

The Performance Schema

Monitor MySQL Server execution at a low level.
Provide details about server execution in structured way, accessible with SQL
PERFORMANCE_SCHEMA is a storage Engine that is only used for special tables in the performance_schema database.
That database contains 52 tables and no views, comprising of a mix of tables that encompass the following categories:

-Configuration Tables
-Object Tables
-Current Tables
-History Tables
-Summary Tables
-Other Tables

The PERFORMANCE_SCHEMA can be utilized to troubleshoot:

-Server bottlenecks, caused by locks, mutexes, IO
-Less-than optimal statements
-Most expensive operations
-Connection issues
-Memory usage

-Replication failures
-More…

Starting from MySQL 5.6.6 the Performance Schema is enabled by default.

Long-running Processes

The Performance Schema keeps track of events that take time as _instruments waits_.

These tables store wait events:

events_waits_current: Current wait events

events_waits_history: The most recent wait events per thread

events_waits_history_long: The most recent wait events globally (across all threads)

Once the performance schema is enabled, it will collect metrics on all the statements executed by the server.
Many of those metrics are summarized in the
events_statements_summary_by_digest table
available in MySQL 5.6 and later.

Metrics on

-query volume,

-latency,

-errors,

-time spent waiting for locks,

-index usage,

-and more are available for each normalized SQL statement executed. (Normalization here means stripping data values from the SQL statement and standardizing whitespace.)

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;
```

Output:

```
  digest_text_start  count_star avg_timer_wait
  ---------------------------------------------------------------------------------

  SHOW FULL TABLES FROM `sakila`                  1      1110825767786

  SHOW GLOBAL STATUS LIKE ?                      1      1038069287388

  SELECT `digest_text`, `count_star`, `avg_timer_w   1      945742257586

  SHOW FIELDS FROM `sakila` . `actor`             1      611721261340

  SELECT `digest_text` , `count_star` , `avg_timer_w  2      335116484794

  SHOW FIELDS FROM `sakila` . `actor_info` SELECT `a  1      221773712160

  SELECT NAME , TYPE FROM `mysql` . `proc` WHERE `Db  2
148939688506

  SHOW FIELDS FROM `vehicles` . `vehiclemodelyear`    1
144172298718

  SHOW SCHEMAS                                   2      132611131408

  SHOW FIELDS FROM `sakila` . `customer`          1      99954017212
```

Event Timer Units

Performance Schema displays event timer information in picoseconds (trillionths of a second) to normalize timing data to a standard unit. In the following example, TIMER_WAIT values are divided by 1000000000000 to show data in units of seconds. Values are also truncated to 6 decimal places.

```
digest_text_start                                count_star avg_timer_wait
-----------------------------------------------------------------------------------------


SHOW FULL TABLES FROM `sakila`                      1      1.110825

SHOW GLOBAL STATUS LIKE ?                          1      1.038069

SELECT `digest_text`, `count_star`, `avg_timer_w       1      0.945742
```

```
etc.
```

Unused Indexes

The **table_io_waits_summary_by_index_usage** table may be employed, not only to aggregate operations per index, but also to aggregate how many operations did not use indexes when accessing the tables.

This may be accomplished by including the **"INDEX_NAME column is NULL"** criteria in the WHERE clause.

```
SELECT object_schema AS schema_name,

      object_name   AS table_name,

      index_name,

      count_fetch

FROM performance_schema.table_io_waits_summary_by_index_usage

WHERE count_fetch > 0;
```

Result:

```
schema_name     table_name      index_name              count_fetch
-------------------------------------------------------------------------------
vehicles    vehiclemodelyear    U_VehicleModelYear_year_make_model
7273

sakila      film                (null)                  1001
```

Long-running Queries: one of the most common causes of slow database performance

1. The events_statements_history_long table contains

```
 - A lot of indicative fields on the subject of wait times and their
corresponding statements.

   The following query compiles some useful calculations on wait times
   to provide a clear picture of which statements are taking the longest to
execute

    SELECT left(digest_text, 64) AS digest_text_start

    , ROUND(SUM(timer_end-timer_start)/1000000000, 1) AS tot_exec_ms

    , ROUND(SUM(timer_end-timer_start)/1000000000/COUNT(*), 1) AS
```

```
avg_exec_ms

    , ROUND(MAX(timer_end-timer_start)/1000000000, 1) AS max_exec_ms

    , ROUND(SUM(timer_wait)/1000000000, 1) AS tot_wait_ms

    , ROUND(SUM(timer_wait)/1000000000/COUNT(*), 1) AS avg_wait_ms

    , ROUND(MAX(timer_wait)/1000000000, 1) AS max_wait_ms

    , ROUND(SUM(lock_time)/1000000000, 1) AS tot_lock_ms

    , ROUND(SUM(lock_time)/1000000000/COUNT(*), 1) AS avglock_ms

    , ROUND(MAX(lock_time)/1000000000, 1) AS max_lock_ms

    , COUNT(*) as count

   FROM events_statements_history_long

   JOIN information_schema.global_status AS isgs

   WHERE isgs.variable_name = 'UPTIME'

   GROUP BY LEFT(digest_text,64)

   ORDER BY tot_exec_ms DESC;
```

Result:

```
digest_text_start                avg_exec_ms    tot_wait_ms     max_wait_ms
avglock_ms      count

                      tot_exec_ms     max_exec_ms     avg_wait_ms     tot_lock_ms
max_lock_ms

-----------------------------------------------------------------------------------------------------------

SELECT * FROM `sakila` . `rent  240.3   240.3   240.3   240.3   240.3   240.3
0.0 0.0 0.0 1

SELECT * FROM `sakila` . `film  56.6    56.6    56.6    56.6    56.6    56.6    0.0
0.0 0.0 1

UPDATE `performance_schema` .   49.8    49.8    49.8    49.8    49.8    49.8
42.0    42.0    42.0    1

SELECT `st` . * FROM `performa  12.7    2.1 11.9    12.7    2.1 11.9    13.0
2.2 12.0    6
```

```
SELECT `st` . * FROM `performa  2.6 0.4 0.5 2.6 0.4 0.5 2.0 0.3 1.0 6

SELECT `st` . * FROM `performa  0.8 0.1 0.2 0.8 0.1 0.2 0.0 0.0 0.0 6

SHOW INDEX FROM `sakila` . `fi  0.8 0.8 0.8 0.8 0.8 0.8 0.0 0.0 0.0 1

SHOW INDEX FROM `sakila` . `re  0.6 0.6 0.6 0.6 0.6 0.6 0.0 0.0 0.0 1
```

2. Here's a way to find a long-running query using the threads table:

```
   SELECT PROCESSLIST_ID AS id,

       PROCESSLIST_USER AS user,

       PROCESSLIST_HOST AS host,

       PROCESSLIST_DB AS db,

       PROCESSLIST_COMMAND AS command,

       PROCESSLIST_TIME AS time,

       PROCESSLIST_STATE AS state,

       LEFT(PROCESSLIST_INFO, 80) AS info

 FROM performance_schema.threads

 WHERE PROCESSLIST_ID IS NOT NULL

 AND PROCESSLIST_COMMAND NOT IN ('Sleep', 'Binlog Dump')

 ORDER BY PROCESSLIST_TIME ASC;
```

Result:

```
Id  user  host      db            command  time  state      info

------------------------------------------------------------------------------------------------------------
------

2   root  localhost  sakila         Query    0      Creating sort index  SELECT
PROCESSLIST_ID AS id,PROCESSLIST_USER…
```

Memory Usage

1. To quickly ascertain how much RAM does a server have (Available since version 5.7,)

the **_sys_ tables** provide detailed information about memory usage of each server's allocated internal memory

```
SELECT * FROM sys.memory_global_total;
```

Result:

```
total_allocated

---------------

458.44 MiB
```

2. To get memory utilization by thread

```
SELECT thread_id,

    user,

    current_avg_alloc,

    current_allocated

FROM sys.memory_by_thread_by_current_bytes

WHERE thread_id IN (145, 146)\G
```

Result:

```
*********************** 1. row ***********************

   thread_id: 145

   user: sql/slave_io

   current_allocated: 1.04 GiB

   current_avg_alloc: 2.64 KiB

   *********************** 2. row ***********************

   thread_id: 146

   user: sql/slave_sql

   current_allocated: 1.79 MiB
```

```
    current_avg_alloc: 72 bytes

    2 rows in set (0.11 sec)
```

MySQL Configuration file Location

The location of the MySQL configuration file (either **my.ini** or **my.cnf**) depends on your OS.

MySQL will look at each location in order, and use the first file that it finds. Often, if the file does not exist, it must be created first.

Linux based MySQL systems will use configuration files in the following order of precedence
- /etc/my.cnf
- /etc/mysql/my.cnf
- SYSCONFDIR/my.cnf
- $MYSQL_HOME/my.cnf

SYSCONFDIR refers to the directory specified when MySQL was built; typically reverse to
the etc directory located under the compiled-in installation directory.

MYSQL_HOME is an environment variable referring to the path where my.cnf can be found.

Windows-based MySQL systems will use the configuration files in the following order of precedence

```
%PROGRAMDATA%\MySQL\<MySQL Server Version>\my.ini
%PROGRAMDATA%\MySQL\<MySQL Server Version>\my.cnf
%WINDIR%\my.ini
%WINDIR%\my.cnf
C:\my.ini
C:\my.cnf
INSTALLDIR\my.ini
INSTALLDIR\my.cnf
```

Fixing Issues and Performance Improvement

Summary/ Table of Contents (tl;dr)

1. Profile the workload

2. Understanding the four fundamental resources (CPU,memory,disk,network)

3. Don't Use MySQL as a queue.

4. Filter results by cheapest first (do cheap, imprecise work first, then the hard, precise work on the smaller, resulting set of data.)

5. Know the two scalability death traps: Avoid serialization and crosstalk, and your application will scale much better.

6. Don’t focus too much on configuration

7. Watch out for pagination queries

8. Save statistics eagerly, alert reluctantly

9. Learn the three rules of indexing.Design the index such a way that
   (i). Indexes let the server find groups of adjacent rows instead of single rows.
   (ii). Indexes avoid sorting which is costly and chooses reading rows in the desired order which is faster
   (iii). Indexes let the server satisfy entire queries from the index alone. (covering index or an index-only query)

10. Leverage the expertise of your peers (Ask in mailing lists, forums, Q&A websites)

1.Profiling the workload

- The best way to understand how your server spends its time
- You can expose the most expensive queries for further tuning
- Time is the most important metric here
- Preferred tool for workload profiling is `pt-query-digest` from the [Percona Toolkit](https://www.percona.com/software/database-tools/percona-toolkit)
- These tools capture queries the server executes and return a table of tasks sorted by decreasing order of response time
- The most expensive and time-consuming tasks is shown to the top
- Tools group similar queries together, allowing you to see the queries that are slow,
  as well as the queries that are fast but executed many times.

2. Proper functioning of 4 fundamental system resources (CPU,memory,disk,network)

- Weak, erratic, or overloaded resources affect performance
- Ensure good-performing components and balance them reasonably well against each other
- Adding memory is a cheap way of increasing performance by orders of magnitude,
  especially on workloads that are disk-bound.
- MySQL will perform well with fast CPUs because each query runs in a single thread
  and can’t be parallelized across CPUs.

3. Avoid Using MySQL as a queue

Queues cause problems for two major reasons:
They serialize your workload, preventing tasks from being done in parallel, and they often result in a table that contains work in process as well as historical data from jobs that were processed long ago.
Both add latency to the application and load to MySQL.

Many of them aren't really a problem in reality.
But the potential is always there, and it's hard to predict which things will become problems.

Job-queue design pattern can creep into an application's database through these following cases

- Storing a list of messages to send:
  whether it's emails, SMS messages, or friend requests,
  if you're storing a list of messages in a table and
  then looking through the list for messages that need to be sent,
  you've created a job queue.

- Moderation, token claims, or approval:
  do you have a list of pending articles, comments, posts, email validations, or users?
  If so, you have a job queue.

- Order processing:
  If your order-processing system looks for newly submitted orders, processes them,
  and updates their status, it's a job queue.

- Updating a remote service:
  Does your ad-management software compute bid changes for ads,
  and then store them for some other process to communicate with the advertising service?
  That's a job queue.

- Incremental refresh or synchronization:
  if you store a list of items that has changed and needs some background processing,
  such as files to sync for your new file-sharing service, that's a job queue too.

As time passes, the job queue table starts to either perform poorly, or cause other things to perform badly

Reasons: Polling, Locking, Data Growth

- Polling

Many of the job queue systems have one or more worker processes checking for something to do.
This starts to become a problem pretty quickly in a heavily loaded application.

- Locking

The specific implementation of the polling often looks like this:
run a SELECT FOR UPDATE to see if there are items to process;
if so, UPDATE them in some way to mark them as in-process;
then process them and mark them as complete.
There are variations on this, not necessarily involving SELECT FOR UPDATE,
but often something with similar effects.
The problem with SELECT FOR UPDATE is that it usually
creates a single synchronization point for all of the worker processes,
and you see a lot of processes waiting for the locks to be released with COMMIT.
Bad implementations of this (not committing until the workers have processed the items, for example)
are really horrible,
but even “good” implementations can cause serious pile-ups.

- Data Growth

Consider a Email list management applications that have a single huge emails table.
New emails go into the table with a “new” status, and then they get updated to mark them as sent.
As time passes, these email tables can grow into millions or even billions of rows.
Even though there might only be hundreds to thousands of new messages to send,
that big bloated table makes all the queries really, really slow.
This combined with polling and/or locking and lots of load on the server, this leads to disaster.

The solutions to these problems:

1) Avoiding polling

Since MySQL doesnt have listen/notify functionality, like the way Postgres, SQL Server etc. have
We can however simulate it in three ways
- use GET_LOCK() and RELEASE_LOCK(), or
- write a plugin to communicate, through [Spread!](http://www.spread.org/), or
- make the consumers run a SLEEP(100000) query,

and then kill these queries to “signal” to the worker that there’s something to do.

2) Avoiding locking

Instead of SELECT FOR UPDATE followed by UPDATE, just UPDATE with a LIMIT, and then see if any rows were affected.

The client protocol tells you that; there’s no need for another query to the database to check.

Make sure autocommit is enabled for this UPDATE, so that you don’t hold the resultant locks open
for longer than the duration of the statement.

If you don’t have autocommit enabled, the application must follow up with a COMMIT to release any locks,
and that is really no different from SELECT FOR UPDATE.

3) Avoiding storing your queue in the same table as other data.

To avoid the one-big-table syndrome.

Just create a separate table for new emails,
and when you’re done processing them, INSERT them into long-term storage and
then DELETE them from the queue table.

The table of new emails will typically stay very small and operations on it will be fast.
And if you do the INSERT before the DELETE, and use INSERT IGNORE or REPLACE,
you don’t even need to worry about using a transaction across the two tables, in case your app crashes between.

That further reduces locking and other overhead.
If you fail to execute the DELETE, you can just have a regular cleanup task retry and purge the orphaned row.

The same thing can be done for any type of queue. For example, articles or comments that are pending approval
can go into a separate table.
This is really required on a **large scale**, and you shouldn’t worry that your WordPress blog doesn’t do things this way.

Alternative solution

Use a real queueing system, such as Resque, ActiveMQ, RabbitMQ, or Gearman

NOTE: Be careful, however, that you don't enable persistence to a database and choose to use MySQL for that. Depending on the queue system, that can just reintroduce the problem in a generic way that's even less optimal.
Some queue systems use all of the database worst practices enumerated above.

4.Filter results by cheapest first

- Do the cheap, imprecise work first, then the hard, precise work on the smaller, resulting set of data.

5. Know the two scalability death traps:

Avoid serialization and crosstalk, and your application will scale much better.
The Universal Scalability Law explains scaling problems in terms of two fundamental costs: serialization and crosstalk.

Avoiding exclusive locks on rows.
Queues, tend to scale poorly for this reason.

6. Don't focus too much on configuration

The defaults that ship with MySQL are one-size-fits-none and badly outdated,
but you don't need to configure everything. It's better to get the fundamentals right and change other settings only if needed.
In most cases, you can get 95 percent of the server's peak performance by setting about 10 options correctly.

Server "tuning" tools aren't recommended because they tend to give guidelines that don't make sense for specific cases.
Some even have dangerous, inaccurate advice coded into them such as cache hit ratios and memory consumption formulas.
These were never right, and they have become even less correct as time has passed

7. Using Pagination Queries

Applications that paginate tend to bring the server to its knees.
In showing you a page of results, with a link to go to the next page, these applications typically group and sort in ways that can't use indexes, and they employ a LIMIT and offset that cause the server to do a lot of work generating, then discarding rows.

Optimizations can often be found in the user interface itself.
Instead of showing the exact number of pages in the results and links to each page individually, you can just show a link to the next page.
You can also prevent people from going to pages too far from the first page.

On the query side, instead of using LIMIT with offset,
you can select one more row than you need, and when the user clicks the “next page” link,
you can designate that final row as the starting point for the next set of results.
For example, if the user viewed a page with rows 101 through 120, you would select row 121 as well;
to render the next page, you’d query the server for rows greater than or equal to 121, limit 21.

8. Save statistics eagerly, alert reluctantly

- It’s very important to capture and save all the metrics
- It will be useful to figure out what changed in the system.
- Avoid tendency to alert way too much.
- Alert on things like the buffer hit ratio or the number of temporary tables created per second.
- The problem is that there is no good threshold for such a ratio.
- The right threshold is not only different from server to server,
  but from hour to hour as your workload changes.
- So, alert sparingly and only on conditions that indicate a definite, actionable problem.
- A low buffer hit ratio isn’t actionable, nor does it indicate a real issue,
- But a server that doesn’t respond to a connection attempt is an actual problem that needs to be solved

9. The three rules of indexing

Indexes, when properly designed, serve three important purposes in a database server.
If you can design your indexes and queries to exploit these three opportunities,
you can make your queries several orders of magnitude faster.

- Indexes let the server find groups of adjacent rows instead of single rows.
  Purpose of indexes is **not** to find individual rows
  Finding single rows leads to random disk operations, which is slow
  It’s much better to find groups of rows, all or most of which are required,
  than to find rows one at a time.

- Indexes let the server avoid sorting by reading the rows in a desired order.
  Sorting is costly.
  Reading rows in the desired order is much faster.

- Indexes let the server satisfy entire queries from the index alone,
  avoiding the need to access the table at all.

10. Leverage the expertise of your peers

- Build a network of MySQL-related resources
  - Toolsets, troubleshooting guides
  - Be active on mailing lists, forums, Q&A websites

More about Tools

Percona Configuration Wizard for MySQL, Percona Query Advisor for MySQL, and Percona Monitoring Plugins. (Note: You'll need to create a Percona account to access those first two links. It's free.) The configuration wizard can help you generate a baseline my.cnf file for a new server that's superior to the sample files that ship with the server. The query advisor will analyze your SQL to help detect potentially bad patterns such as pagination queries (No. 7). Percona Monitoring Plugins are a set of monitoring and graphing plugins to help you save statistics eagerly and alert reluctantly (No. 8). All of these tools are freely available.