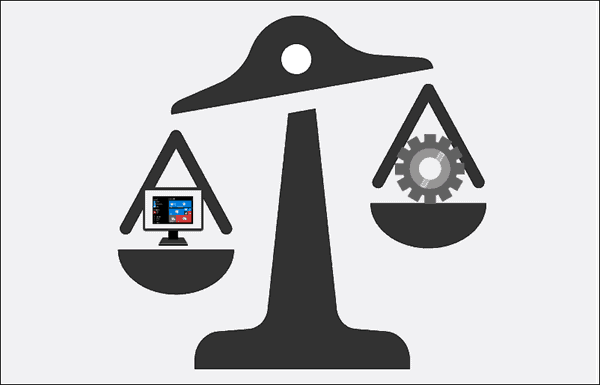
**System MySQL Performance Tuning**

At the system level, you’ll adjust hardware and software options to improve MySQL performance.

**1. Balance the Four Main Hardware Resources**



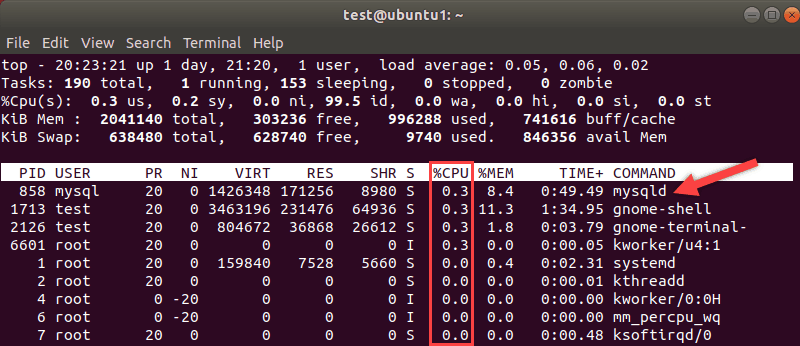
**Storage**

Take a moment to evaluate your storage. If you’re using traditional hard disk drives (HDD), you can upgrade to [solid-state drives (SSD) for a performance improvement](https://phoenixnap.com/kb/nvme-vs-sata-vs-m-2-comparison).

Use a tool like **iotop** or **sar** from the **sysstat** package to monitor your disk input/output rates. If disk usage is much higher than usage of other resources, consider adding more storage or upgrading to faster storage.

**Processor**

Processors are usually considered the measure of how fast your system is. Use the [Linux **top** command](https://phoenixnap.com/kb/linux-commands-check-memory-usage) for a breakdown of how your resources are used. Pay attention to the MySQL processes and the percentage of processor usage they require.



Processors are more expensive to upgrade, but if your CPU is a bottleneck, an upgrade might be necessary.

**Memory**

Memory represents the total amount of RAM in your MySQL [database storage server](https://phoenixnap.com/servers/database). You can adjust the memory cache (more on that later) **to improve performance**. If you don’t have enough memory, or if the existing memory isn’t optimized, you can end up damaging your performance instead of improving it.

Like other bottlenecks, if your server is constantly running out of memory, you can upgrade by adding more. If you run short of memory, your server will cache data storage (like a hard drive) to act as memory. Database caching slows down your performance.

**Network**

It’s important to [monitor network traffic](https://phoenixnap.com/kb/linux-network-bandwidth-monitor-traffic) to make sure you have sufficient infrastructure to manage the load.

Overloading your network can lead to latency, dropped packets, and even server outages. Make sure you have enough [network bandwidth](https://phoenixnap.com/kb/website-bandwidth) to accommodate your normal levels of database traffic.

**2. Use InnoDB, Not MyISAM**

**MyISAM** is an older database style used for some MySQL databases. It is a less efficient database design. The newer **InnoDB** supports more advanced features and has in-built optimization mechanics.

InnoDB uses a clustered index and keeps data in pages, which are stored in consecutive physical blocks. If a value is too large for a page, InnoDB moves it to another location, then indexes the value. This feature helps keep relevant data in the same place on the storage device, meaning it takes the physical hard drive less time to access the data.

**3. Use the Latest Version of MySQL**

Using the latest version is not always feasible for older and legacy databases. But whenever possible, you should [check the version of MySQL](https://phoenixnap.com/kb/how-to-check-mysql-version) in use and upgrade to the latest.

A part of the ongoing development includes performance enhancements. Some common performance adjustments may be rendered obsolete by newer versions of MySQL. In general, it’s always better to use native MySQL performance enhancement over scripting and configuration files.

**Software MySQL Performance Tuning**

SQL performance tuning is the process of maximizing query speeds on a [relational database](https://phoenixnap.com/kb/what-is-a-relational-database). The task usually involves multiple tools and techniques.

These methods involve:

* Tweaking the MySQL configuration files.
* Writing more efficient database queries.
* Structuring the database to retrieve data more efficiently.

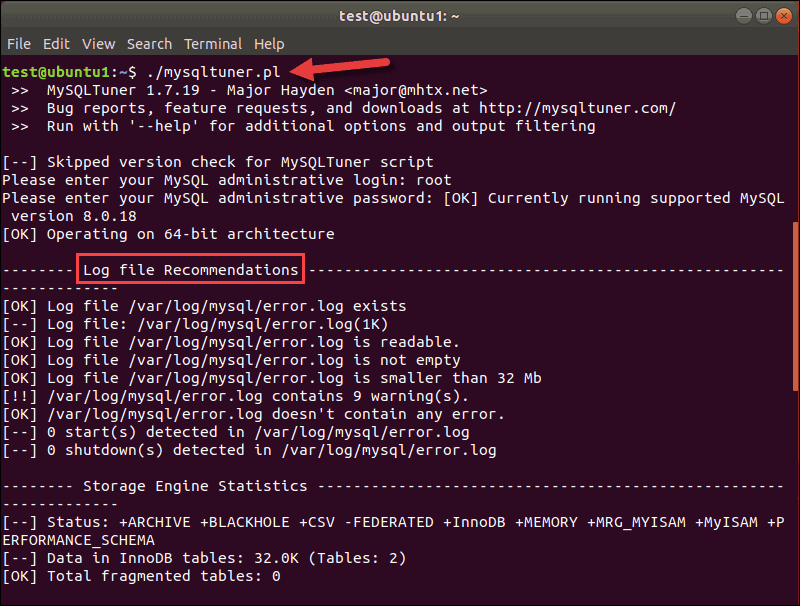
**Note:** When adjusting configuration settings, it’s best to make small incremental adjustments. A major adjustment may overburden another value and degrade performance. Also, it is recommended that you make one change at a time and then test. It’s easier to track errors or misconfigurations when you only change one variable at a time.

**4. Consider Using an Automatic Performance Improvement Tool**

As with most software, not all tools work on all versions of MySQL. We will examine three utilities to evaluate your MySQL database and recommend changes to improve performance.

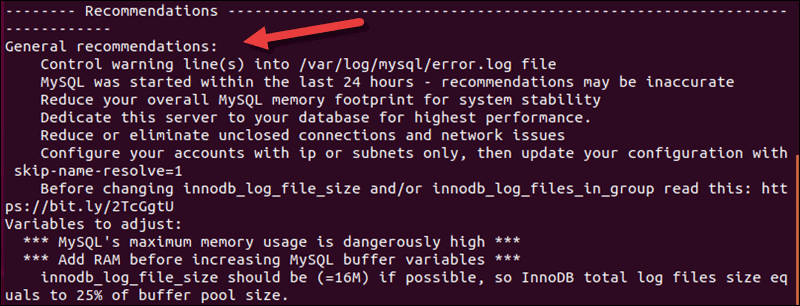
The first is [tuning-primer](https://github.com/RootService/tuning-primer). This tool is a bit older, designed for MySQL 5.5 – 5.7. It can analyze your database and suggest settings to improve performance. For example, it may suggest that you raise the ***query\_cache\_size*** parameter if it feels like your system can’t process queries quickly enough to keep the cache clear.

The second tuning tool, useful for most modern SQL databases, is [MySQLTuner](https://github.com/major/MySQLTuner-perl" \t "_blank). This script (***mysqltuner.pl***) is written in Perl. Like tuning-primer, it analyzes your database configuration looking for bottlenecks and inefficiencies. The output shows metrics and recommendations:



At the top of the output, you can see the version of the MySQLTuner tool and your database.

The script works with MySQL 8.x. Log file recommendations are the first on the list, but if you scroll to the bottom, you can see general recommendations for improving MySQL performance.



The third utility, which you may already have, is the **phpMyAdmin Advisor**. Like the other two utilities, it evaluates your database and recommends adjustments. If you’re already using phpMyAdmin, the Advisor is a helpful tool you can use within the GUI.

**Note:** Check out our list of [top SQL query optimization tools](https://phoenixnap.com/kb/sql-query-optimization-tool) and use our in-depth analysis of each one to find the best one for your tasks.

**5. Optimize Queries**

A query is a coded request to search the database for data that matches a certain value. There are some query operators that, by their very nature, take a long time to run. SQL performance tuning techniques help optimize queries for better run times.

Detecting queries with poor execution time is one of the main tasks performance tuning. Commonly implemented queries on large datasets are slow and occupy databases. The tables are therefore unavailable for any other tasks.

**Note:** Consider looking into the [data warehouse architecture](https://phoenixnap.com/kb/data-warehouse-architecture-explained), which separates production databases from analytical.

For example, an [OLTP database](https://phoenixnap.com/kb/oltp-database) requires fast transactions and effective query processing. Running an inefficient query blocks the use of the database and stalls information updates.

If your environment relies on automated queries such as [triggers](https://phoenixnap.com/kb/mysql-trigger) , they may be impacting performance. Check and [terminate MySQL processes](https://phoenixnap.com/kb/how-to-kill-mysql-process) that may pile up in time.

**6. Use Indexes Where Appropriate**

Many database queries use a structure similar to this:

SELECT … WHERE

These queries involve evaluating, filtering, and retrieving results. You can restructure these by adding [a small set of indexes](https://phoenixnap.com/kb/mysql-create-index) for the related tables. The query can be directed at the index to speed up the query.

**7. Functions in Predicates**

Avoid using a function in the predicate of a query. For example:

SELECT \* FROM MYTABLE WHERE UPPER(COL1)='123'Copy

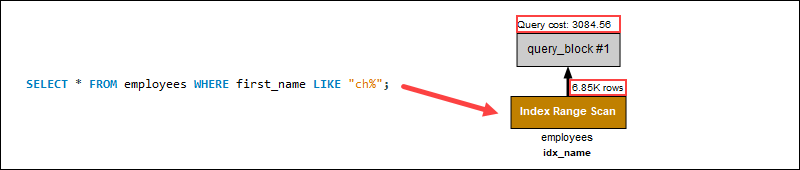
The **UPPER** notation creates a function, which must operate during the **SELECT** operation. This doubles the work the query is doing, and you should avoid it if possible.

**8. Avoid % Wildcard in a Predicate**

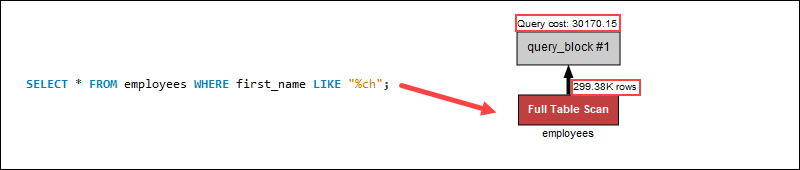
When searching through textual data, wildcards help make a wider search. For example, to select all names that start with *ch*, create an index on the name column and run:

SELECT \* FROM person WHERE name LIKE "ch%"

The query scans the indexes, making the query cost low:



However, doing a search for names using the wildcards in the beginning increases the query cost significantly because an indexing scan does not apply to ends of strings:



A wildcard at the beginning of a search does not apply indexing. Instead, a full table scan searches through each row individually, increasing the query cost in the process. In the example query, using a wildcard at the end helps reduce the query cost due to going through fewer table rows.

**Note:** Check out our [MySQL Commands Cheat Sheet](https://phoenixnap.com/kb/mysql-commands-cheat-sheet" \t "_blank), which features the indexing commands.

A way to search ends of strings is to reverse the string, index the reversed strings and look at the starting characters. Placing the wildcard at the end now searches for the beginning of the reversed string, making the search more efficient.

**9. Specify Columns in SELECT Function**

A commonly used expression for analytical and exploratory queries is **SELECT \***. Selecting more than you need results in unnecessary performance loss and redundancy. If you specify the columns you need, your query won’t need to scan irrelevant columns.

If all columns are needed, there is no other way to go about it. However, most business requirements do not need all columns available within a dataset. Consider selecting specific columns instead.

To summarize, avoid using:

SELECT \* FROM table

Instead, try:

SELECT column1, column2 FROM table

**10. Use ORDER BY Appropriately**

The **ORDER BY** expression sorts results by the specified column. It can be used to sort by two columns at once. These should be sorted in the same order, ascending or descending.

If you try to sort different columns in different order, it will slow down performance. You may combine this with an index to speed up the sorting.

**11. GROUP BY Instead of SELECT DISTINCT**

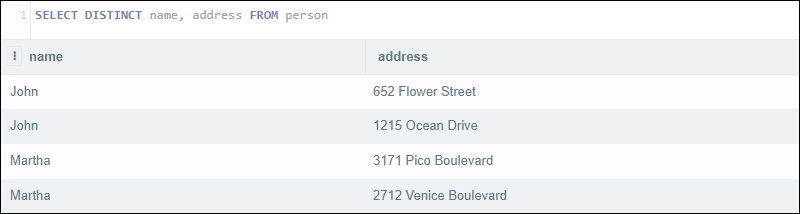
The **SELECT DISTINCT** query comes in handy when trying to get rid of duplicate values. However, the statement requires a large amount of processing power.

Whenever possible, avoid using **SELECT DISTINCT**, as it is very inefficient and sometimes confusing. For example, if a table lists information about customers with the following structure:

| **id** | **name** | **lastName** | **address** | **city** | **state** | **zip** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | John | Smith | 652 Flower Street | Los Angeles | CA | 90017 |
| **1** | John | Smith | 1215 Ocean Boulevard | Los Angeles | CA | 90802 |
| **2** | Martha | Matthews | 3104 Pico Boulevard | Los Angeles | CA | 90019 |
| **3** | Martha | Jones | 2712 Venice Boulevard | Los Angeles | CA | 90019 |

Running the following query returns four results:

SELECT DISTINCT name, address FROM person



The statement seems like it should return a list of distinct names along with their address. Instead, the query looks at **both** the name and address column. Although there are two pairs of customers with the same name, their addresses are different.

To filter out duplicate names and return the addresses, try using the **GROUP BY** statement:

SELECT name, address FROM person GROUP BY name

The result returns the first distinct name along with the address, making the statement less ambiguous. To group by unique addresses, the **GROUP BY** parameter would just change to address and return the same result as the **DISTINCT** statement faster.

To summarize, avoid using:

SELECT DISTINCT column1, column2 FROM table

Instead, try using:

SELECT column1, column2 FROM table GROUP BY column1

**12. JOIN, WHERE, UNION, DISTINCT**

Try to use an inner join whenever possible. An outer join looks at additional data outside the specified columns. That’s fine if you need that data, but it’s a waste of performance to include data that won’t be required.

Using **INNER JOIN** is the standard approach to joining tables. Most database engines accept using **WHERE** as well. For example, the following two queries output the same result:

SELECT \* FROM table1 INNER JOIN table2 ON table1.id = table2.id

Compared with:

SELECT \* FROM table1, table2 WHERE table1.id = table2.id

In theory, they have the same runtime as well.

The choice on whether to use **JOIN** or **WHERE** query depend on the database engine. While most engines have the same runtime for the two methods, in some database systems one runs faster than the other.

**Note:** Learn more about [MySQL JOINS](https://phoenixnap.com/kb/mysql-join" \t "_blank) and how to use them.

The **UNION** and **DISTINCT** commands are sometimes included in queries. Like an outer join, it’s fine to use these expressions if they are necessary. However, they add additional sorting and reading of the database. If you don’t need them, it’s better to find a more efficient expression.

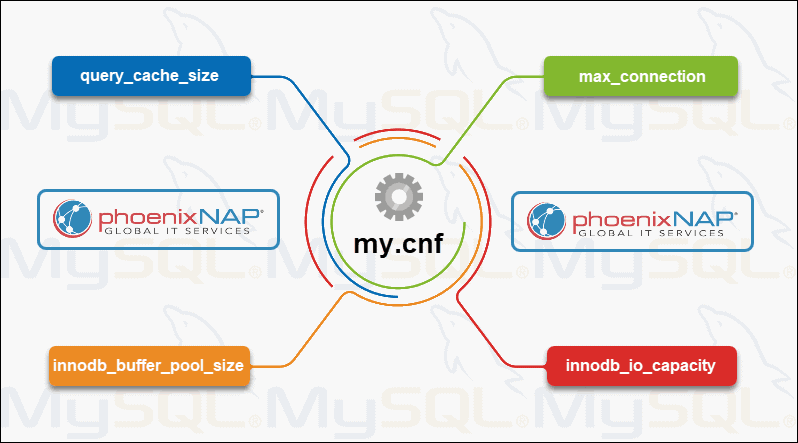
**13. Use the EXPLAIN Function**

Modern MySQL databases include an **EXPLAIN** function.

Appending the **EXPLAIN** expression to the beginning of a query will read and evaluate the query. If there are inefficient expressions or confusing structures, **EXPLAIN** can help you find them. You can then adjust the phrasing of your query to avoid unintentional table scans or other performance hits.

**14. MySQL Server Configuration**

This configuration involves making changes to your **/etc/mysql/my.cnf** file. Proceed with caution and make minor changes at a time.



**query\_cache\_size** – Specifies the size of the cache of MySQL queries waiting to run. The recommendation is to start with small values around 10MB and then increase to no more than 100-200MB. With too many cached queries, you can experience a cascade of queries “Waiting for cache lock.” If your queries keep backing up, a better procedure is to use **EXPLAIN** to evaluate each query and find ways to make them more efficient.

**max\_connection** – Refers to the number of connections allowed into the database. If you’re getting errors citing “*Too many connections,*” increasing this value may help.

**innodb\_buffer\_pool\_size**– This setting allocates system memory as a data cache for your database. If you have large chunks of data, increase this value. Take note of the RAM required to run other system resources.

**innodb\_io\_capacity**– This variable sets the rate for input/output from your storage device. This is directly related to the type and speed of your storage drive. A 5400-rpm HDD will have a much lower capacity than a [high-end SSD or Intel Optane](https://phoenixnap.com/kb/optane-memory-vs-ssd-vs-ram). You can adjust this value to better match your hardware.

**Conclusion**

You should now know how to improve MySQL performance and tune your database.

Look for bottlenecks (hardware and software), queries that are doing more work than needed, and consider using automated tools and the **EXPLAIN** function to evaluate your database.

Optimizing MySQL tables helps reorder information in a dedicated storage server to improve data input and output speeds. Check out our guide on how to [optimize MySQL tables](https://phoenixnap.com/kb/mysql-optimize-table).