How To Troubleshoot Issues in MySQL

Posted December 5, 2018 8.5k MYSQL DATABASES UBUNTU UBUNTU 18.04

By: Mark Drake

Introduction

MySQL is an open-source relational database management system (RDBMS), the most popular of its kind in the world. As is the case when working with any software, both newcomers and experienced users can run into confusing error messages or difficult-to-diagnose problems.

This guide will serve as a troubleshooting resource and starting point as you diagnose your MySQL setup. We'll go over some of the issues that many MySQL users encounter and provide guidance for troubleshooting specific problems. We will also include links to DigitalOcean tutorials and the official MySQL documentation that may be useful in certain cases.

Please note that this guide assumes the setup described in How To Install MySQL on Ubuntu 18.04, and the linked tutorials throughout the guide reflect this configuration. If your server is running another distribution, however, you can find a guide specific to that distro in the Tutorial Version Menu at the t

tutorials when one is available.

How To Get Started with MySQL

The place where many first-time users of MySQL run into a problem is during the installation and configuration process. Our guide on How To Install MySQL on Ubuntu 18.04 provides instructions on how to set up a basic configuration and may be helpful to those new to MySQL.

Another reason some users run into issues is that their application requires database features that are only present in the latest releases, but the version of MySQL available in the default repositories of some Linux distributions — including Ubuntu — isn't the latest version. For this reason, the MySQL developers maintain their own software repository, which you can use to install the latest version and keep it up to date. Our tutorial "How To Install the Latest MySQL on Ubuntu 18.04" provides instructions on how to do this.

How to Access MySQL Error Logs

Oftentimes, the root cause of slowdowns, crashes, or other unexpected behavior in MySQL can be determined by analyzing its error logs. On Ubuntu systems, the default location for the MySQL is `/var/log/mysql/error.log`. In many cases, the error logs are most easily read with the `less` program, a command line utility that allows you to view files but not edit them:

```
$ sudo less /var/log/mysql/error.log
```

If MySQL isn't behaving as expected, you can obtain more information about the source of the trouble by running this command and diagnosing the error based on the log's contents.

Resetting the root MySQL User's Password

If you've set a password for your MySQL installation's **root** user but have since forgotten it, you could be locked out of your databases. As long as you have access to the server on which your database is hosted, though, you should be able to reset it.

This process differs from resetting the password for a standard Linux username. Check out our guide on How To Reset Your MySQL or MariaDB Root Password to walk through and understand this process.

Troubles with Queries

Sometimes users run into problems once they begin issuing queries on their data. In some database systems, including MySQL, query statements in must end in a semicolon (`;`) for the query to complete, as in the following example:

```
mysql> SHOW * FROM table_name;
```

If you fail to include a semicolon at the end of your query, the prompt will continue on a new line until you complete the query by entering a semicolon and pressing `ENTER`.

Some users may find that their queries are exceedingly slow. One way to find which query statement is the cause of a slowdown is to enable and view MySQL's slow query log. To do this, open your `mysqld.cnf` file, which is used to configure options for the MySQL server. This file is typically stored within the `/etc/mysql/mysql.conf.d/` directory:

```
$ sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

Scroll through the file until you see the following lines:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
#slow_query_log         = 1
#slow_query_log_file    = /var/log/mysql/mysql-slow.log
#long_query_time = 2
#log-queries-not-using-indexes
. . .
```

These commented-out directives provide MySQL's default configuration options for the slow query log. Specifically, here's what each of them do:

- `slow-query-log`: Setting this to `1` enables the slow query log.
- `slow-query-log-file`: This defines the file where MySQL will log any slow queries. In this case, it points to the `/var/log/mysql-slow.log` file.
- `long_query_time`: By setting this directive to `2`, it configures MySQL to log any queries that take longer than 2 seconds to complete.
- `log_queries_not_using_indexes`: This tells MySQL to also log any queries that run without indexes to the `/var/log/mysql-slow.log` file. This setting isn't required for the slow query log to function, but it can be helpful for spotting inefficient queries.

Uncomment each of these lines by removing the leading pound signs (`#`). The section will now look like this:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
slow_query_log = 1
slow_query_log_file = /var/log/mysql-slow.log
long_query_time = 2
log_queries_not_using_indexes
. . .
```

Note: If you're running MySQL 8+, these commented lines will not be in the `mysqld.cnf` file by default. In this case, add the following lines to the bottom of the file:

```
. . .
slow_query_log = 1
slow_query_log_file = /var/log/mysql-slow.log
long_query_time = 2
log_queries_not_using_indexes
```

After enabling the slow query log, save and close the file. Then restart the MySQL service:

```
$ sudo systemctl restart mysql
```

With these settings in place, you can find problematic query statements by viewing the slow query log. You can do so with `less`, like this:

```
$ sudo less /var/log/mysql_slow.log
```

Once you've singled out the queries causing the slowdown, you may find our guide on How To Optimize Queries and Tables in MySQL and MariaDB on a VPS to be helpful with optimizing them.

Additionally, MySQL includes the `EXPLAIN` statement, which provides information about how MySQL executes queries. This page from the official MySQL documentation provides insight on how to use `EXPLAIN` to highlight inefficient queries.

For help with understanding basic query structures, see our Introduction to MySQL Queries.

Allowing Remote Access

Many websites and applications start off with their web server and database backend hosted on the same machine. With time, though, a setup like this can become cumbersome and difficult to scale. A common solution is to separate these functions by setting up a remote database, allowing the server and database to grow at their own pace on their own machines.

One of the more common problems that users run into when trying to set up a remote MySQL database is that their MySQL instance is only configured to listen for local connections. This is MySQL's default setting, but it won't work for a remote database setup since MySQL must be able to listen for an *external* IP address where the server can be reached. To enable this, open up your `mysqld.cnf` file:

```
$ sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

Navigate to the line that begins with the `bind-address` directive. It will look like this:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
lc-messages-dir = /usr/share/mysql
skip-external-locking
#
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
bind-address            = 127.0.0.1
. . .
```

By default, this value is set to `127.0.0.1`, meaning that the server will only look for local connections. You will need to change this directive to reference an external IP address. For the purposes of troubleshooting, you could set this directive to a wildcard IP address, either `*`, `::`, or `0.0.0.0`:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
lc-messages-dir = /usr/share/mysql
skip-external-locking
#
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
bind-address            = 0.0.0.0
. . .
```

Note: If you're running MySQL 8+, the `bind-address` directive will not be in the `mysqld.cnf` file by default. In this case, add the following highlighted line to the bottom of the file:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
[mysqld]
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
datadir         = /var/lib/mysql
log-error       = /var/log/mysql/error.log
bind-address            = 0.0.0.0
```

After changing this line, save and close the file and then restart the MySQL service:

```
$ sudo systemctl restart mysql
```


Following this, try accessing your database remotely from another machine:

```
$ mysql -u user -h database_server_ip -p
```

If you're able to access your database, it confirms that the `bind-address` directive in your configuration file was the issue. Please note, though, that setting `bind-address` to `0.0.0.0` is insecure as it allows connections to your server from any IP address. On the other hand, if you're still unable to access the database remotely, then something else may be causing the issue. In either case, you may find it helpful to follow our guide on How To Set Up a Remote Database to Optimize Site Performance with MySQL on Ubuntu 18.04 to set up a more secure remote database configuration.

MySQL Stops Unexpectedly or Fails to Start

The most common cause of crashes in MySQL is that it stopped or failed to start due to insufficient memory. To check this, you will need to review the MySQL error log after a crash.

First, attempt to start the MySQL server by typing:

```
$ sudo systemctl start mysql
```

Then review the error logs to see what's causing MySQL to crash. You can use `less` to review your logs, one page at a time:

```
$ sudo less /var/log/mysql/error.log
```

Some common messages that would indicate an insufficient amount of memory are `Out of memory` or `mmap can't allocate`.

Potential solutions to an inadequate amount of memory are:

- **Optimizing your MySQL configuration**. A great open-source tool for this is MySQLtuner. Running the MySQLtuner script will output a set of recommended adjustments to your MySQL configuration file (`mysqld.cnf`). Note that the longer your server has been running before using MySQLTuner, the more accurate its suggestions will be. To get a memory usage estimate of both your current settings and those proposed by MySQLTimer, use this MySQL Calculator.
- **Reducing your web application's reliance on MySQL for page loads**. This can usually be done by adding static caching to your application. Examples for this include Joomla, which has caching as a built-in feature that can be enabled, and WP Super Cache, a WordPress plugin that adds this kind of functionality.
- **Upgrading to a larger VPS**. At minimum, we recommend a server with at least 1GB of RAM for any server using a MySQL database, but the size and type of your data can significantly affect memory requirements.

Take note that even though upgrading your server is a potential solution, it's only recommended after you investigate and weigh all of your other options. An upgraded server with more resources will likewise cost more money, so you should only go through with resizing if it truly ends up being your b

note that the MySQL documentation includes a number of other suggestions for diagnosing and preventing crashes.

Corrupted Tables

Occasionally, MySQL tables can become corrupted, meaning that an error has occurred and the data held within them is unreadable. Attempts to read from a corrupted table will usually lead to the server crashing.

Some common causes of corrupted tables are:

- The MySQL server stops in middle of a write.
- An external program modifies a table that's simultaneously being modified by the server.
- The machine is shut down unexpectedly.
- The computer hardware fails.
- There's a software bug somewhere in the MySQL code.

If you suspect that one of your tables has been corrupted, you should make a backup of your data directory before troubleshooting or attempting to fix the table. This will help to minimize the risk of data loss.

First, stop the MySQL service:

```
$ sudo systemctl stop mysql
```

Then copy all of your data into a new backup directory. On Ubuntu systems, the default data directory is `/var/lib/mysql/` :

```
$ cp -r /var/lib/mysql /var/lib/mysql_bkp
```

After making the backup, you're ready to begin investigating whther the table is in fact corrupted. If the table uses the MyISAM storage engine, you can check whether it's corrupted by running a `CHECK TABLE` statement from the MySQL prompt:

```
mysql> CHECK TABLE table_name;
```

A message will appear in this statement's output letting you know whether or not it's corrupted. If the MyISAM table is indeed corrupted, it can usually be repaired by issuing a `REPAIR TABLE` statement:

```
mysql> REPAIR TABLE table_name;
```

Assuming the repair was successful, you will see a message like the following in your ou

Output

```
+--------------------------+--------+----------+----------+
| Table                    | Op     | Msg_type | Msg_text |
+--------------------------+--------+----------+----------+
| database_name.table_name | repair | status   | OK       |
+--------------------------+--------+----------+----------+
```

If the table is still corrupted, though, the MySQL documentation suggests a few alternative methods for repairing corrupted tables.

On the other hand, if the corrupted table uses the InnoDB storage engine, then the process for repairing it will be different. InnoDB is the default storage engine in MySQL as of version 5.5, and it features automated corruption checking and repair operations. InnoDB checks for corrupted pages by performing checksums on every page it reads, and if it finds a checksum discrepancy it will automatically stop the MySQL server.

There is rarely a need to repair InnoDB tables, as InnoDB features a crash recovery mechanism that can resolve most issues when the server is restarted. However, if you do encounter a situation where you need to rebuild a corrupted InnoDB table, the MySQL documentation recommends using the "Dump and Reload" method. This involves regaining access to the corrupted table, using the `mysqldump` utility to create a *logical backup* of the table, which will retain the table structure and the data within it, and then reloading the table back into the database.

With that in mind, try restarting the MySQL service to see if doing so will allow you access to the server:

```
$ sudo systemctl restart mysql
```

If the server remains crashed or otherwise inaccessible, then it may be helpful to enable InnoDB's `force_recovery` option. You can do this by editing the `mysqld.cnf` file:

```
$ sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

In the `[mysqld]` section, add the following line:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
[mysqld]
. . .
innodb_force_recovery=1
```

Save and close the file, and then try restarting the MySQL service again. If you can successfully access the corrupted table, use the `mysqldump` utility to dump your table data to a new file. You can name this file whatever you like, but here we'll name it `out.sql`:


```
$ mysqldump database_name table_name > out.sql
```

Then drop the table from the database. To avoid having to reopen the MySQL prompt, you can use the following syntax:

```
$ mysql -u user -p --execute="DROP TABLE database_name.table_name"
```

Following this, restore the table with the dump file you just created:

```
$ mysql -u user -p < out.sql
```

Note that the InnoDB storage engine is generally more fault-tolerant than the older MyISAM engine. Tables using InnoDB *can* still be corrupted, but because of its auto-recovery features the risk of table corruption and crashes is decidedly lower.

Socket Errors

MySQL manages connections to the database server through the use of a *socket file*, a special kind of file that facilitates communications between different processes. The MySQL server's socket file is named `mysqld.sock` and on Ubuntu systems it's usually stored in the `/var/run/mysqld/` directory. This file is created by the MySQL service automatically.

Sometimes, changes to your system or your MySQL configuration can result in MySQL being unable to read the socket file, preventing you from gaining access to your databases. The most common socket error looks like this:

```
Output
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/var/run/mysqld/mysqld.sock'
```

There are a few reasons why this error may occur, and a few potential ways to resolve it.

One common cause of this error is that the MySQL service is stopped or did not start to begin with, meaning that it was unable to create the socket file in the first place. To find out if this is the reason you're seeing this error, try starting the service with `systemctl`:

```
$ sudo systemctl start mysql
```

Then try accessing the MySQL prompt again. If you still receive the socket error, double check the location where your MySQL installation is looking for the socket file. This information can be found in the `mysqld.cnf` file:

```
$ sudo nano /etc/mysql/mysql.conf.d/mysql.cnf
```

Look for the `socket` parameter in the `[mysqld]` section of this file. It will look like this:

/etc/mysql/mysql.conf.d/mysqld.cnf

```
. . .
[mysqld]
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
. . .
```

Close this file, then ensure that the `mysqld.sock` file exists by running an `ls` command on the directory where MySQL expects to find it:

```
$ ls -a /var/run/mysqld/
```

If the socket file exists, you will see it in this command's output:

```
Output
.  ..  mysqld.pid  mysqld.sock  mysqld.sock.lock
```

If the file does not exist, the reason may be that MySQL is trying to create it, but does not have adequate permissions to do so. You can ensure that the correct permissions are in place by changing the directory's ownership to the **mysql** user and group:

```
$ sudo chown mysql:mysql /var/run/mysqld/
```

Then ensure that the **mysql** user has the appropriate permissions over the directory. Setting these to `775` will work in most cases:

```
$ sudo chmod -R 755 /var/run/mysqld/
```

Finally, restart the MySQL service so it can attempt to create the socket file again:

```
$ sudo systemctl restart mysql
```

Then try accessing the MySQL prompt once again. If you still encounter the socket error, there's likely a deeper issue with your MySQL instance, in which case you should review the error log to see if it can provide any clues.

Conclusion

MySQL serves as the backbone of countless data-driven applications and websites. With so many use cases, there are as many potential causes of errors. Likewise, there are also many different ways to resolve such errors. We've covered some of the most frequently encountered errors in this guide, but there are many more that could come up depending on how your own application works with MySQL.

If you weren't able to find a solution to your particular problem, we hope that this guide will at least give you some background into MySQL troubleshooting and help you find the source of your errors. For more information, you can look at the official MySQL documentation, which covers the topics we have discussed here as well as other troubleshooting strategies.

Additionally, if your MySQL database is hosted on a DigitalOcean Droplet, you can contact our Support team for further assistance.

By: Mark Drake

Related Tutorials

How To Ensure Code Quality with SonarQube on Ubuntu 18.04

How To Sync and Share Your Files with Seafile on Ubuntu 18.04

How To Set Up a Remote Database to Optimize Site Performance with MySQL on Ubuntu 18.04

How To Set Up Laravel, Nginx, and MySQL with Docker Compose

How To Install and Configure pgAdmin 4 in Server Mode

0 Comments

Leave a comment...

Log In to Comment