

# Chapter 9. Using Option Files

Almost every piece of software is capable of being configured, or even must be configured. MySQL is not much different in this regard. While the default configuration will probably suit an astonishing number of installations, more likely than not, you will end up needing to configure the server, or a client.

MySQL provides two ways to configure itself: through command-line argument options, and through the configuration file. Since this file contains only the options that could be specified on the command line, it's also called the *option* file.

The option file is not exclusive to MySQL Server. It's also not strictly correct to talk about the *option file*, as pretty much every installation of MySQL will have multiple option files. Most MySQL software supports inclusion in the option files, and we'll cover that, too.

Knowing your way around an option file—understanding its sections and option precedence—is an important part of efficiently working with MySQL Server and related software. After going through this chapter, you should feel comfortable configuring MySQL Server and other programs that use option files. This chapter will focus on the files themselves. The configuration of the server itself and some tuning ideas are discussed in depth in Chapter 11.

## Structure of the Option File

Configuration files in MySQL follow the ubiquitous INI file scheme. In short, they are regular text files that are intended to be edited manually. Of course, you can automate the editing process, but the structure of these files is purposefully very simple. Almost every MySQL configuration file can be created and modified with any text editor. There are just two exceptions to this rule, reviewed in “[Special Option Files](#)”.

To give you an idea of the file structure, let's take a look at a configuration file shipped with MySQL 8 on Fedora Linux (note that the exact contents of the

option files on your system may be different). We've redacted a few lines for brevity:

```
$ cat /etc/my.cnf
...
[mysqld]
#
# Remove leading # and set to the amount of RAM for the
# cache in MySQL. Start at 70% of total RAM for dedicated
# innodb_buffer_pool_size = 128M
...
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

log-error=/var/log/mysqld.log
pid-file=/run/mysqld/mysqld.pid
```

---

**TIP**

On some Linux distributions, such as Ubuntu, the `/etc/my.cnf` configuration file doesn't exist in a default MySQL installation. Look for `/etc/mysql/my.cnf` on those systems, or refer to "[Search Order for Option Files](#)" for a way to get a full list of option files `mysqld` reads.

---

There are a few main parts to the file structure:

*Section (group) headers*

These are the values in square brackets preceding the configuration parameters. All programs using option files look for parameters in one or more named sections. For example, `[mysqld]` is a section used by the MySQL server, and `[mysql]` is used by the `mysql` CLI program. The name of the sections are, strictly speaking, arbitrary, and you can put anything there. However, if you change `[mysqld]` to `[section]`, your MySQL server will ignore all the options following that header.

MySQL documentation calls sections *groups*, but both terms can be used interchangeably.

Headers control how the files are parsed, and by which programs.

Each option after a section header and before the next section header will be attributed to the first header. An example will make this clearer:

```
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
[mysql]
default-character-set=latin1
```

Here, the `datadir` and `socket` options are under (and will be attributed to) the `[mysqld]` section, whereas the `default-character-set` option is under `[mysql]`. Note that some MySQL programs read multiple sections; but we'll talk about that later.

Section headers can be intertwined. The following example is completely valid:

```
[mysqld]
datadir=/var/lib/mysql
[mysql]
default-character-set=latin1
[mysqld]
socket=/var/lib/mysql/mysql.sock
[mysqld_safe]
core-file-size=unlimited
[mysqld]
core-file
```

Such a configuration might be difficult for a person to read, but programs that parse the file will not care about the order. Still, it's probably best to keep the configuration files as human-readable as possible.

### *Option-value pairs*

This is the main part of the option file, consisting of the configuration variables themselves and their values. Each of these pairs is defined on a new line and follows one of two general patterns. In addition to the `option=value` pattern shown in the previous example, there's also

just the *option* pattern. For example, the same standard MySQL 8 configuration file has the following lines:

```
# Remove the leading "# " to disable binary loggi
# Binary logging captures changes between backups
# default. Its default setting is log_bin=binlog
# disable_log_bin
```

`disable_log_bin` is an option without a value. If we uncomment it, MySQL Server will apply the option.

With the *option=value* pattern, you can add spaces around the equals sign for readability if you prefer. Any whitespace preceding and following the option name and value will be truncated automatically.

Option values can also be enclosed in single or double quote characters. That is useful if you're not sure whether the value is going to be interpreted correctly. For example, on Windows, paths contain the `\` symbol, which is treated like an escape symbol. Thus, you should put paths on Windows in double quotes (although you could also escape each `\` by doubling it as `\\"`). Quoting the option values is required when the value includes the `#` symbol, which would otherwise be treated as indicating the start of a comment.

The rule of thumb that we recommend is to use quotes when you're not sure. Here are some valid option/value pairs that illustrate the previous points:

```
slow_query_log_file = "C:\mysql\data\query.log"
slow_query_log_file=C:\\mysql\\data\\\\query.log
innodb_temp_tablespaces_dir=".#/innodb_temp/"
```

When setting values for numerical options, like the sizes of different buffers and files, working with bytes can get tedious. To make life easier, MySQL understands a variety of suffixes standing in for different units. For example, the following are all equivalent and define a buffer pool of the same size (268,435,456 bytes):

```
innodb_buffer_pool_size = 268435456
innodb_buffer_pool_size = 256M
innodb_buffer_pool_size = 256MB
innodb_buffer_pool_size = 256MiB
```

You can also specify `G`, `GB`, and `GiB` for gigabytes and `T`, `TB`, and `TiB` for terabytes if you have a server large enough. Of course, `K` and other forms are also accepted. MySQL always uses base-2 units: 1 GB is 1,024 MB, not 1,000 MB.

You cannot specify fractional values for options. So, for example, `0.25G` is an incorrect value for the `innodb_buffer_pool_size` variable. Also, unlike when setting values from the `mysql` CLI or another client connection, you cannot use mathematical notation for option values. You can run `SET GLOBAL max_heap_table_size=16*1024*1024;`, but you cannot put the same value in the option file.

You can even configure the same option multiple times, as we did with `innodb_buffer_pool_size`. The last setting will take precedence over the previous ones, and the files are scanned from top to bottom. Option files have a global order of precedence as well; we'll talk about that in [“Search Order for Option Files”](#).

A very important thing to remember is that setting an incorrect option name will lead to programs not starting. Of course, if the incorrect option is under a section that particular program doesn't read, it's fine. But `mysqld` will fail if it finds an option it doesn't know under `[mysqld]`. In MySQL 8.0 you can validate some of the changes you make in option files by using the `--validate-config` command-line argument with `mysqld`. However, that validation will cover only core server functionality and won't verify storage engine options.

Sometimes you'll need to set an option that MySQL doesn't know at startup time. For example, this can be useful when configuring plugins that may be loaded after startup. You can prepend such options with the `Loose-` prefix (or `--loose-` on the command line), and MySQL will only output a warning when it sees these but not fail to start. Here's an example with an unknown option:

```
# mysqld --validate-config
2021-02-11T08:02:58.741347Z 0 [ERROR] [MY-000067]
... unknown variable audit_log_format=JSON.
2021-02-11T08:02:58.741470Z 0 [ERROR] [MY-010119]
```



After the option is changed to `loose-audit_log_format`, we see the following instead. No output means that all the options were successfully validated:

```
# mysqld --validate-config
#
```

### *Comments*

An often overlooked but important feature of MySQL option files is the ability to add comments. Comments allow you to include arbitrary text, usually a description of why the setting is here, that will not be parsed by any MySQL programs. As you saw in the `disable_log_bin` example, comments are lines starting with `#`. You can also create comments that start with semicolon (`;`); either is accepted. You don't necessarily need to have a whole line dedicated to a comment: they can also appear at the end of a line, although in this case they must start with a `#`, not a `;`. Once MySQL finds a `#` on a line (unless it's escaped), everything beyond that point is treated as a comment. The following line is a valid configuration:

```
innodb_buffer_pool_size = 268435456 # 256M
```

### *Inclusion directives*

Configuration files (and whole directories) can be included within other config files. This can make it easier to manage complex configurations, but it also makes reading the options more difficult, because humans, unlike programs, can't really merge the files together easily. Still, it's useful to be able to separate the configurations of different MySQL programs. The `xtrabackup` utility (see [Chapter 10](#)), for example, doesn't have any special config file and reads standard system option files. With inclusion, you can have `xtrabackup`'s configs neatly organized in a dedicated file and

de clutter your main MySQL option file. You can then include it as follows:

```
$ cat /etc/my.cnf
!include /etc/mysql.d/xtrabackup.cnf
...
```

You can see that */etc/my.cnf* includes the */etc/mysql.d/xtrabackup.cnf* file, which in turn has a few configuration options listed in the **[xtrabackup]** section.

It is not necessary to have different sections in different files, though. For example, Percona XtraDB Cluster has `wsrep` library configuration options under the **[mysqld]** section. There are plenty of such configurations, and they aren't necessarily useful to have in your *my.cnf*. You could create a separate file—for example, */etc/mysql.d/wsrep.conf*—and list the `wsrep` variables under the **[mysqld]** section there. Any program reading the main *my.cnf* file will also read all of the included files and only then parse the variables under the different sections.

When a lot of such extra configuration files are created, you may want to just go ahead and include the whole directory or directories that contain them instead of including each individual option file. That's done with another directive, `includedir`, that expects a directory path as an argument:

```
!includedir /etc/mysql.d
```

MySQL programs will understand that path as a directory and try to include every option file in that directory's tree. On Unix-like systems, *.cnf* files are included; on Windows, both *.cnf* and *.ini* files are included.

Usually, inclusions are defined at the beginning of a particular config file, but that isn't mandatory. You can think of the inclusion as appending the contents of the included file or files to the parent file; wherever an inclusion is defined in the file, the included file's contents will be placed right under that inclusion. In reality, things are a bit

more complicated, but this mental model works when, for example, thinking about option precedence, which we cover in “[Search Order for Option Files](#)”.

Each included file must have at least one configuration section defined. For example, it may have `[mysqld]` at the beginning.

#### *Empty lines*

There’s no meaning to empty lines in the option files. You can use them to separate the sections or individual options visually to make the file easier to read.

## Scope of Options

We can talk about option scope in MySQL from two perspectives. First, each individual option can have global scope, session scope, or both, and can be set dynamically or statically. Second, we can talk about how options set in option files are scoped through sections and what the scope and order of precedence of the option files themselves is.

We mentioned that section headers define which particular program (or programs, as nothing prevents one from reading multiple sections) is intended to be reading the options under a particular header. Some configuration options do not make sense outside of their sections, but some can be defined under multiple sections and do not necessarily need to be set equally.

Let’s consider an example where we have a MySQL server configured with the `latin1` character set for legacy reasons. However, there are now newer tables with the `utf8mb4` charset. We want our `mysqldump` logical dumps to just be in UTF-8, so we want to override the charset for this program. Conveniently, `mysqldump` reads its own configuration section, so we can write an option file like this:

```
[mysqld]
character_set_server=latin1
[mysqldump]
default_character_set=utf8mb4
```

This small example shows how options can be set on different levels. In this particular case we used different options, but it could be the same one in different scopes. For example, suppose we want to limit the future size of `BLOB` and `TEXT` values (see “[String types](#)”) to 32 MiB, but we already have rows of up to 256 MiB in size. We can add an artificial barrier for local clients using a configuration like this:

```
[mysqld]
max_allowed_packet=256M
[client]
max_allowed_packet=32M
```

The MySQL server’s `max_allowed_packet` value will be set on a global scope and will act as a hard limit on the maximum query size (and also on `BLOB` or `TEXT` field size). The client’s value will be set on a session scope and will act as a soft limit. If a particular client requires a larger value (to read an old row, for example), it can use the `SET` statement to go up to the server’s limit.

The option files themselves also have different scopes. MySQL option files can be divided by scope into a few groups: global, client, server, and extra. Global option files are read by all or most MySQL programs, whereas client and server files are only read by client programs and `mysqld`, respectively. Since it’s possible to specify an extra configuration file to be read by a program, we’re also listing the “extra” category.

Let’s outline the option files installed and read on Linux and Windows by a regular MySQL 8.0 installation. We’ll start with Windows, in [Table 9-1](#).

Table 9-1. MySQL option files on Windows

Filename	Scope and purpose
<code>%WINDIR%\my.ini,</code> <code>%WINDIR%\my.cnf</code>	Global options read by all programs
<code>C:\my.ini, C:\my.cnf</code>	Global options read by all programs
<code>BASEDIR\my.ini, BASEDIR\my.cnf</code>	Global options read by all programs
Extra config file	File optionally specified with <code>--defaults-extra-file</code>
<code>%APPDATA%\MySQL\mylogin.cnf</code>	Login path config file
<code>DATADIR\mysqld-auto.cnf</code>	Option file for persisted variables

[Table 9-2](#) breaks down the option files for a typical installation on Fedora Linux.

Table 9-2. MySQL option files on Fedora Linux

Filename	Scope and purpose
<code>/etc/my.cnf</code> , <code>/etc/mysql/my.cnf</code> , <code>/usr/etc/my.cnf</code>	Global options read by all programs
<code>\$MYSQL_HOME/my.cnf</code>	Server options, only read if the variable is set
<code>~/.my.cnf</code>	Global options read by all programs run by a particular OS user
Extra config file	File optionally specified with <code>--defaults-extra-file</code>
<code>~/.mylogin.cnf</code>	Login path config file under a particular OS user
<code>DATADIR/mysql-auto.cnf</code>	Option file for persisted variables

With Linux, it's difficult to identify a universal, complete list of configuration files, as MySQL packages for different Linux distributions may read slightly different files or locations. As a rule of thumb, `/etc/my.cnf` is a good starting point on Linux, and either `%WINDIR%\my.cnf` or `BASEDIR\my.cnf` on Windows.

A couple of the configuration files we've listed may differ in their paths between different systems. `/usr/etc/my.cnf` can be also written as `SYSCONFIGDIR/my.cnf`, and the path is defined at compilation time. `$MYSQL_HOME/my.cnf` is only read if the variable is set. The default packaged `mysqld_safe` program (used to start the `mysqld` daemon) will set `$MYSQL_HOME` to `BASEDIR` before running `mysqld`. You won't find `$MYSQL_HOME` set for any of the OS users, and setting that variable is relevant only if you're starting `mysqld` manually—in other words, not using the `service` or `systemctl` commands.

There's one peculiar difference between Windows and Linux. On Linux, MySQL programs read some configuration files located under the given OS

user's home directory. In [Table 9-2](#), the home directory is represented by `~`. MySQL on Windows lacks this ability. One frequent use case for such config files is controlling options for clients based on their OS user. Usually, they will contain credentials. However, the login path facility described in [“Special Option Files”](#) makes this redundant.

An extra config file, specified on the command line with `--defaults-extra-file`, will be read after every other global file is read, according to its position in the table. This is a useful option when you want to do a one-off run of a program to test new variables, for example. Overusing this option, though, can lead to trouble in understanding the current set of options in effect (see [“Determining the Options in Effect”](#)). The `--defaults-extra-file` option is not the only one that alters option file handling. `--no-defaults` prevents the program from reading *any* configuration files at all. `--defaults-file` forces the program to read a single file, which can be useful if you have your custom configuration all in one place.

By now you should have a firm grasp on what option files most installations of MySQL use. The next section talks more about how different programs read different files, in which order, and what specific group or groups they read from those files.

## Search Order for Option Files

At this point you should know the structure of an option file and where to find them. Most MySQL programs read one or more option files, and it's important to know in which specific order a program searches for these files and reads them. This section covers the topics of search order and options precedence and discusses their importance.

If a MySQL program reads any option files, you can find the specific files it reads, as well as the order in which it reads them. The general order of the configuration files read will be either exactly the same or very similar to that outlined in Tables [9-1](#) and [9-2](#). You can use the following command to see the exact order:

```
$ mysql --verbose --help | grep "Default options" -A2
Default options are read from the following files in th
```

```
/etc/my.cnf /etc/mysql/my.cnf /usr/etc/my.cnf ~/.my.cnf  
The following groups are read: mysqld server mysqld-8.0
```

On Windows you need to run `mysqld.exe` instead of `mysqld`, but the output will stay the same. That output includes the list of configuration files read, and their order. You can also see the list of option groups read by `mysqld`: [mysqld], [server], and [mysqld-8.0]. Note that you can alter the list of option groups that any program reads by adding the `--defaults-group-suffix` option:

```
$ mysqld --defaults-group-suffix=-test --verbose --help  
The following groups are read: mysqld server mysqld-8.0  
... mysqld-test server-test mysqld-8.0-test
```

You know at this point what option files and option groups are read. However, it's also important to know the order of precedence for those option files. Nothing prevents you from setting one or more options in multiple configuration files, after all. In the case of MySQL programs, the order of precedence for config files is simple: options from files read later take precedence over options in previously read files. Options passed to commands directly as command-line arguments take precedence over any configuration options in any config files.

In Tables [9-1](#) and [9-2](#), the files are read in order from top to bottom. The lower the config file in the list, the higher the “weight” of options there. For example, for any programs that are not `mysqld`, values in `.mylogin.cnf` take precedence over those in any other config files, and only have lower precedence than values set through command-line arguments. For `mysqld`, the same is true for persisted variables set in `DATADIR /mysqld-auto.cnf`.

The ability to include configuration files in other files through inclusion directives makes things slightly more complicated, but you always include extras within one or more of the option files listed in Tables [9-1](#) and [9-2](#). You can think of this as MySQL appending the included files to the parent config file just before reading it, inserting each one into the file just after its inclusion directive. Thus, the precedence of the options globally is that of the parent configuration file. Within the resulting file itself (with all the included files added in order), options defined later take precedence over ones defined earlier.

# Special Option Files

There are two special configuration files used by MySQL, which are exceptions to the structure outlined in “[Structure of the Option File](#)”.

## Login Path Configuration File

First, there’s a `.mylogin.cnf` file, which is used as part of the *login path* system. Even though you can think of its structure as similar to that of a regular option file, this particular file is not a regular text file. In fact, it’s an encrypted text file. This file is intended to be created and modified through use of the special `mysql_config_editor` program, which is supplied with MySQL, usually in the client package. It is encrypted because the purpose of `.mylogin.cnf` (and the whole login path system) is to store MySQL connection options, including passwords, in a convenient and secure manner.

By default, `mysql_config_editor` and other MySQL programs will look for `.mylogin.cnf` in the `$HOME` of the current user on Linux and various Unix flavors, and in `%APPDATA%\MySQL` on Windows. It is possible to change the location and name of the file by setting the `MYSQL_TEST_LOGIN_FILE` environment variable.

You can create this file, if it doesn’t already exist, by storing a password for the `root` user in it:

```
$ mysql_config_editor set --user=root --password  
Enter password:
```

After entering the password and confirming the input, we can take a look at the file’s contents:

```
$ ls -la ~/.mylogin.cnf  
-rw----- 1 skuzmichev skuzmichev 100 Jan 18 18:03 .n  
$ cat ~/.mylogin.cnf
```

>pZ  
prI

R86w"># &.h.m:4+|DDKn1\_K3>73x\$

```
$ file ~/.mylogin.cnf
.mylogin.cnf: data
$ file ~/.my.cnf
.my.cnf: ASCII text
```

As you can see, on the surface at least, *.mylogin.cnf* is for sure not a regular configuration file. As such, it requires special treatment. In addition to creating the file, you can view and modify *.mylogin.cnf* with the `mysql_config_editor`. Let's start with how to actually see what's inside. The option for that is `print`:

```
$ mysql_config_editor print
[client]
user = "root"
password = *****
```

`client` is a default login path. All operations done with `mysql_config_editor` without an explicit login path specification affect the `client` login path. We didn't specify any login path when running `set` earlier, so `root`'s credentials were written under the `client` path. It's possible to specify a specific login path for any operation, though. Let's put `root`'s credentials under a login path named `root`:

```
$ mysql_config_editor set --login-path=root --user=root
Enter password:
```



To specify the login path, use the `--login-path` or `-G` option, and to view all the paths when using `print`, add the `--all` option:

```
$ mysql_config_editor print --login-path=root
[root]
user = root
password = *****
$ mysql_config_editor print --all
[client]
user = root
password = *****
[root]
user = root
password = *****
```

You can see that the output resembles an option file, so you can think of `.mylogin.cnf` as an option file with some special treatment. Just don't edit it manually. Speaking of editing, let's add a few more options to the `set` command as `mysql_config_editor` calls it. We'll create a new login path in the process.

`mysql_config_editor` supports the `--help` (or `-?`) argument, which can be combined with other options to get help specifically on `print` or `set`, for example. Let's start by looking at a slightly truncated help output for `set`:

```
$ mysql_config_editor set --help
...
MySQL Configuration Utility.

Description: Write a login path to the login file.
Usage: mysql_config_editor [program options] [set [comm
    -?, --help           Display this help and exit.
    -h, --host=name      Host name to be entered into the
    -G, --login-path=name
                           Name of the login path to use in
                           : client)
    -p, --password       Prompt for password to be entered
    -u, --user=name      User name to be entered into the
    -S, --socket=name    Socket path to be entered into the
    -P, --port=name      Port number to be entered into the
    -w, --warn            Warn and ask for confirmation if
                           overwrite an existing login path
                           (Defaults to on; use --skip-warn
...

```



You can see here another interesting property of `.mylogin.cnf`: you can't put arbitrary parameters into it. Now we know that we can basically set only a few options related to logging into a MySQL instance or instances—which is, of course, to be expected of the “login path” file. Now, let's get back to editing the file:

```
$ mysql_config_editor set --login-path=scott --user=scc
$ mysql_config_editor set --login-path=scott --user=scc
WARNING : scott path already exists and will be overwri
```

```
Continue? (Press y|Y for Yes, any other key for No) :  
$ mysql_config_editor set --login-path=scott --user=scott
```

Here we've shown all the behaviors that `mysql_config_editor` can exhibit when modifying or creating a login path. If the login path doesn't yet exist, no warning is produced. If there's already such a path, a warning and confirmation will be printed, but only if `--skip-warn` is not specified. Note that we're talking here in terms of the whole login path! It is not possible to modify a single property of the path: the whole login path is written out every time. If you want to change a single property, you'll need to specify all the other properties that you need too.

Let's add some more details and view the result:

```
$ mysql_config_editor set --login-path=scott \  
--user=scott --port=3306 --host=192.168.122.1 \  
--password --skip-warn  
Enter password:  
$ mysql_config_editor print --login-path=scott  
[scott]  
user = scott  
password = *****  
host = 192.168.122.1  
port = 3306
```

## Persistent System Variables Configuration File

The second special file is `mysqld-auto.cnf`, which has resided in the data directory since MySQL 8.0. It is a part of the new persisted system variables feature, which allows you to update MySQL options on disk using regular `SET` statements. Before, you could not change MySQL's configuration from within a database connection. The usual flow was to change the option files on disk and then run a `SET GLOBAL` statement to change the configuration variables online. As you can imagine, this can lead to mistakes and to changes only being made online, for example. The new `SET PERSIST` statement takes care of both tasks: variables updated online are also updated on disk. It's also possible to update a variable on disk only.

The file itself is, surprisingly, not like any other configuration file in MySQL at all. Whereas `.mylogin.cnf` was an encrypted but still regular option file,

*mysqld-auto.cnf* uses a common but completely different format: JSON.

Before you persist anything, *mysqld-auto.cnf* doesn't exist. So, we'll start by changing a system variable:

```
mysql> SELECT @@GLOBAL.max_connections;
```

```
+-----+  
| @@GLOBAL.max_connections |  
+-----+  
| 100 |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> SET PERSIST max_connections = 256;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> SELECT @@GLOBAL.max_connections;
```

```
+-----+  
| @@GLOBAL.max_connections |  
+-----+  
| 256 |  
+-----+  
1 row in set (0.00 sec)
```

As expected, the variable was updated on a global scope online. Let's now explore the resulting config file. Since we know that the contents are in JSON format, we'll use the `jq` utility to format it nicely. That's not necessary, but makes the file easier to read:

```
$ cat /var/lib/mysql/mysqld-auto.cnf | jq .  
{  
  "Version": 1,  
  "mysql_server": {  
    "max_connections": {  
      "Value": "256",
```

```
        "Metadata": {  
            "Timestamp": 1611728445802834,  
            "User": "root",  
            "Host": "localhost"  
        }  
    }  
}  
}
```

Just by looking at this file containing a single variable value, you can see why plain `.ini` is used for config files that are intended to be edited by humans. This is verbose! However, JSON is excellent for reading by computers, so it's a good match for a configuration written and read by MySQL itself. As an added benefit, you get auditing of the changes: as you can see, the `max_connection` property has metadata containing the time when the change occurred and the author of the change.

Since this is a text file, unlike the login path config file, which is binary, it's possible to edit `mysqld-auto.cnf` manually. However, it's unlikely that there will be many cases where that's needed.

## Determining the Options in Effect

The last routine task that pretty much anyone working with MySQL will face is finding out values for the variables, and in what option files they are set (and why, but no amount of technology can help with human reasoning sometimes!).

At this point, we know what files MySQL programs read, in what order, and their precedence. We also know that command-line arguments override any other settings. Still, understanding where exactly some variable is set can be a daunting task. Multiple files scanned, potentially with nested inclusions, can make for a long investigation.

Let's start by looking at how to determine the options currently used by a program. For some, like MySQL Server (`mysqld`), that is easy. You can get the list of current values used by `mysqld` by running `SHOW GLOBAL VARIABLES`. It's impossible to change an option value that `mysqld` uses and not see the effect reflected in the global variables' state. For other programs, things get more complicated. To understand what options are used by `mysql`, you'd have to run it and then check the outputs of `SHOW`

`VARIABLES` and `SHOW GLOBAL VARIABLES` to see which options are overridden on the session level. But even before a successful connection to the server is established, `mysql` must read or receive connection information.

There are two easy ways to determine the list of options in effect when the program starts: by passing the `--print-defaults` argument to that program or by using the special `my_print_defaults` program. Let's take a look at the former option as executed on Linux. You can ignore the `sed` part, but it makes the output slightly nicer for human eyes:

```
$ mysql --print-defaults
mysql would have been started with the following arguments:
--user=root --password=*****
$ mysqld --print-defaults | sed 's/--/\n--/g'
/usr/sbin/mysqld would have been started with the following arguments:
--datadir=/var/lib/mysql
--socket=/var/lib/mysql/mysql.sock
--log-error=/var/log/mysqld.log
--pid-file=/run/mysqld/mysqld.pid
--max_connections=100000
--core-file
--innodb_buffer_pool_in_core_file=OFF
--innodb_buffer_pool_size=256MiB
```

The variables picked up here come from all the option files we discussed before. If a variable value was set multiple times, the last occurrence will take precedence. However, `--print-defaults` will actually output every option set. For example, the output could look like this—even though `innodb_buffer_pool_size` is set five times, the value in effect will be 384 M:

```
$ mysqld --print-defaults | sed 's/--/\n--/g'
/usr/sbin/mysqld would have been started with the following arguments:
--datadir=/var/lib/mysql
--socket=/var/lib/mysql/mysql.sock
--log-error=/var/log/mysqld.log
--pid-file=/run/mysqld/mysqld.pid
--max_connections=100000
```

```
--core-file
--innodb_buffer_pool_in_core_file=OFF
--innodb_buffer_pool_size=268435456
--innodb_buffer_pool_size=256M
--innodb_buffer_pool_size=256MB
--innodb_buffer_pool_size=256MiB
--large-pages
--innodb_buffer_pool_size=384M
```

You can also combine `--print-defaults` with other command-line arguments. For example, if you intend to run a program with command-line arguments, you can see whether they will override or repeat already set values for configuration options for a particular session:

```
$ mysql --print-defaults --host=192.168.4.23 --user=bot
mysql would have been started with the following arguments

--user=root
--password=*****
--host=192.168.4.23
--user=bob
```

The other way to print variables is using the `my_print_defaults` program. It takes one or more section headers as arguments and will print all options it finds in scanned files that fall into the requested groups. That can be better than using `--print-defaults` when you just need to review one option group. In MySQL 8.0, the `[mysqld]` program reads the following groups: `[mysqld]`, `[server]`, and `[mysqld-8.0]`. The combined output of options may be lengthy, but what if we only need to view options specifically set for 8.0? For this example, we've added the `[mysqld-8.0]` option group to the option file and put a couple of configuration parameter values there:

```
$ my_print_defaults mysqld-8.0
--character_set_server=latin1
--collation_server=latin1_swedish_ci
```

That can also help with other software, like PXC, or with the MariaDB flavor of MySQL, both of which include multiple configuration groups. In particular, you would likely want to review the `[wsrep]` section without any other

options. `my_print_defaults` can, of course, be used to output a complete set of options too; it just needs to be passed all the section headers a program reads. For example, the `[mysql]` program reads the `[mysql]` and `[client]` option groups, so we could use:

```
$ my_print_defaults mysql client
--user=root
--password=*****
--default-character-set=latin1
```

The user and password definitions come from the client group in the login path config we set before, and the charset from the `[mysql]` option group in the regular `.my.cnf`. Note that we added that group and charset config manually; by default that option is not set.

You can see that while both ways to read options talk of *defaults*, they actually output the options that we have explicitly set, making them nondefault. This is an interesting tidbit, but it doesn't change anything in the grand scheme of things.

Unfortunately, neither of these ways of reviewing options is perfect at determining the complete set of options in effect. The problem is they only read the configuration files listed in Tables [9-1](#) and [9-2](#), but it's possible for MySQL programs to read other config files or to be started with command-line arguments. Additionally, the variables persisted in `DATADIR /mysqld-auto.cnf` through `SET PERSIST` are not provided by defaults-printing routines.

We mentioned that MySQL programs do not read options from any other files than the ones that were listed in Tables [9-1](#) and [9-2](#). However, those lists do include the “extra config file,” which can be in an arbitrary location. Unless you specify the same extra file when invoking `my_print_defaults` or another program with `--print-defaults`, options from that extra file won't be read. The extra file is specified with the command-line argument, `--defaults-extra-file`, and can be specified for most if not all MySQL programs. The two defaults-printing routines only read predefined config files and will miss that file. You can, however, specify `--defaults-extra-file` both for `my_print_defaults` and for the program invoked with `--print-defaults`, and both will read the extra file then. The same applies to the `--defaults-file` option we mentioned earlier, which basically forces

the MySQL program to only read the single file passed as a value for this option.

Both `--defaults-extra-file` and `--defaults-file` share a thing in common: they are command-line arguments. Command-line arguments passed to a MySQL program override any options read from configuration files, but at the same time you can miss them when you do `--print-defaults` or `my_print_defaults`, as they are coming from outside of any config files. To put it more concisely: a particular MySQL program, such as `mysqld`, can be started by someone with unknown and arbitrary command-line arguments. Thus, when we're talking about options, in effect we must also consider the presence of such arguments.

On Linux and Unix-like systems, you can use the `ps` utility (or an equivalent) to view information on currently running processes, including their full command lines. Let's see an example on Linux where `mysqld` was started with `--no-defaults` and with all config options passed as arguments:

```
$ ps auxf | grep mysqld | grep -v grep
root      397830  ... \_ sudo -u mysql bash -c mysqld .
mysql     397832  ... \_ mysqld --datadir=/var/lib/my
```

Or, if we print just the command line for the `mysqld` process and make it cleaner with `sed`:

```
$ ps -p 397832 -o command ww | sed 's/--/\n--/g'
COMMAND
mysqld
--datadir=/var/lib/mysql
--socket=/var/lib/mysql/mysql.sock
--log-error=/var/log/mysqld.log
--pid-file=/run/mysqld/mysqld.pid
...
--character_set_server=latin1
--collation_server=latin1_swedish_ci
```

Note that for this example we started `mysqld` without using any of the provided scripts. You won't often see this way of starting the MySQL server, but it's possible.

You can pass any configuration option as an argument, so the output can be quite lengthy. However, when you're not sure how exactly `mysqld` or another program was executed, that's an important thing to check. On Windows, you can view the command-line arguments of a running program either by opening the Task Manager and adding a Command Line column to the Processes tab (through the View menu) or by using the Process Explorer tool from the `sysinternals` package.

If your MySQL program is started from within a script, you should inspect that script to find all the arguments used. While this is probably going to be a rare occasion for `mysqld`, it's a common practice to run `mysql`, `mysqldump`, and `xtrabackup` from custom scripts.

Understanding the currently used options can be a daunting task, but it's extremely important at times. Hopefully, these guidelines and hints will help you.