Chapter 16. Miscellaneous Topics

The idea of this chapter is to go beyond troubleshooting a query, or an overloaded system, or setting up different MySQL topologies. We want to show you the arsenal of tools available for you to make daily tasks easier or investigate complex issues. Let's start with MySQL Shell.

MySQL Shell

MySQL Shell is an advanced client and code editor for MySQL. It expands the functionality of the traditional MySQL client that most DBAs worked with in MySQL 5.6 and 5.7. MySQL Shell supports programming languages such as Python, JavaScript, and SQL. It also extends functionalities using an API command syntax. For example, it is possible to customize scripts to administer an InnoDB Cluster. From MySQL Shell, you can also start and configure MySQL sandbox instances.

Installing MySQL Shell

For supported Linux distributions, the easiest way to install MySQL Shell is to use the MySQL *yum* or *apt* repository. Let's see how to install it on Ubuntu and CentOS.

Installing MySQL Shell on Ubuntu 20.04 Focal Fossa

Installing MySQL Shell in Ubuntu is relatively easy since it is part of the regular repositories.

First, we need to configure the MySQL repository. We can use these commands to download the *apt* repository to our server and install it:

```
# wget  https://dev.mysql.com/get/mysql-apt-config_0.8.
# dpkg -i mysql-apt-config_0.8.16-1_all.deb
```

Once installed, update our package information:

```
# apt-get update
```

Then execute the `install` command to install MySQL Shell:

```
# apt-get install mysql-shell
```

We can now start MySQL Shell using the command line:

```
# mysqlsh


MySQL Shell 8.0.23

Copyright (c) 2016, 2021, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation
Other names may be trademarks of their respective owner

Type '\help' or '\?' for help; '\quit' to exit.
 MySQL  JS >
```

Installing MySQL Shell on CentOS 8

To install MySQL Shell in CentOS 8, we follow the same steps as described for Ubuntu—but first we need to make sure the default MySQL package present in CentOS 8 is disabled:

```
# yum remove mysql-community-release -y


No match for argument: mysql-community-release
No packages marked for removal.
Dependencies resolved.
Nothing to do.
Complete!


# dnf erase mysql-community-release
```

```
No match for argument: mysql-community-release
No packages marked for removal.
Dependencies resolved.
Nothing to do.
Complete!
```

Next, we are going to configure our *yum* repository. We need to get the correct OS version from the download page:

```
# yum install \
    https://dev.mysql.com/get/mysql80-community-release
```

With the repository installed, we will install the MySQL Shell binary:

```
# yum install mysql-shell -y
```

And we can validate that the installation worked by running it:

```
# mysqlsh


MySQL Shell 8.0.23

Copyright (c) 2016, 2021, Oracle and/or its affiliates.
Oracle is a registered trademark of Oracle Corporation
Other names may be trademarks of their respective owner

Type '\help' or '\?' for help; '\quit' to exit.
 MySQL  JS >
```

Deploying a Sandbox InnoDB Cluster with MySQL Shell

MySQL Shell automates the deployment of sandbox instances with AdminAPI, which provides the `dba.deploySandboxInstance( port_number )` command.

By default, the sandbox instances are placed in a directory named *$HOME/mysql-sandboxes/port*. Let's see how to change the directory:

```
# mkdir /var/lib/sandboxes
# mysqlsh
```

```
MySQL  JS > shell.options.sandboxDir='/var/lib/sandbox
```

```
/var/lib/sandboxes
```

A prerequiste to deploy a sandbox instance is to install the MySQL binaries. If necessary, review Chapter 1 for details. You'll need to enter a password for the `root` user in order to complete the deployment:

```
MySQL  JS > dba.deploySandboxInstance(3310)
```

```
A new MySQL sandbox instance will be created on this hc
/var/lib/sandboxes/3310

Warning: Sandbox instances are only suitable for deploy
running on your local machine for testing purposes and
accessible from external networks.

Please enter a MySQL root password for the new instance

Deploying new MySQL instance...

Instance localhost:3310 successfully deployed and start
Use shell.connect('root@localhost:3310') to connect to
```

We are going to deploy two more instances:

```
MySQL  JS > dba.deploySandboxInstance(3320)
MySQL  JS > dba.deploySandboxInstance(3330)
```

The next step is to create the InnoDB Cluster while connected to the seed MySQL Server instance. The *seed* instance is the instance we are connected to

via MySQL Shell and that we want to replicate to the other instances. In this example the sandbox instances are all blank instances, so we can choose any instance. In a production setup the seed instance would be the one that contains the existing dataset to be replicated to the other instances in the cluster.

We use this command to connect MySQL Shell to the seed instance, in this case the one at port 3310:

```
MySQL  JS > \connect root@localhost:3310


Creating a session to root@localhost:3310
Please provide the password for root@localhost:3310:
Save password for root@localhost:3310? [Y]es/[N]o/Ne[v]
Fetching schema names for autocompletion... Press ^C tc
Your MySQL connection id is 12
Server version: 8.0.21 Source distribution
No default schema selected; type \use <schema> to set c
```

Subsequently, we will use the `createCluster()` method to create the InnoDB Cluster with the currently connected instance as the seed:

```
MySQL localhost:3310 ssl  JS > var cluster = dba.create


A new InnoDB cluster will be created on instance 'local

Validating instance configuration at localhost:3310...
NOTE: Instance detected as a sandbox.
Please note that sandbox instances are only suitable fc
for use within the same host.

This instance reports its own address as 127.0.0.1:3310

Instance configuration is suitable.
NOTE: Group Replication will communicate with other mem
'127.0.0.1:33101'. Use the localAddress option to overr

Creating InnoDB cluster 'learning_mysql' on '127.0.0.1:

Adding Seed Instance...
```

```
Cluster successfully created. Use Cluster.addInstance()
At least 3 instances are needed for the cluster to be a
one server failure.
```

As we can see in the output, three instances are capable of keeping the database online with one server failure, which is why we deployed three sandbox instances.

The next step is to add secondary instances to our `learning_mysql` InnoDB Cluster. Any transactions that were executed by the seed instance are reexecuted by each secondary instance as it is added.

The seed instance in this example was recently created, so it is nearly empty. Therefore, there is little data that needs to be replicated from the seed instance to the secondary instances. If it's necessary to replicate data, MySQL will use the clone plugin (discussed in "Creating a Replica Using the Clone Plugin") to configure the instances automatically.

Let's add one secondary to see the process in action. To add the second instance to the InnoDB Cluster:

```
MySQL  localhost:3310 ssl  JS >  cluster.addInstance('r
```

```
...

* Waiting for clone to finish...
NOTE: 127.0.0.1:3320 is being cloned from 127.0.0.1:331
** Stage DROP DATA: Completed
** Clone Transfer
    FILE COPY  ##########################################
    100%  Completed
    PAGE COPY  ##########################################
    100%  Completed
    REDO COPY  ##########################################
    100%  Completed

NOTE: 127.0.0.1:3320 is shutting down...

* Waiting for server restart... ready
* 127.0.0.1:3320 has restarted, waiting for clone to fi
** Stage RESTART: Completed
```

```
* Clone process has finished: 59.62 MB transferred in a
(~59.62 MB/s)

State recovery already finished for '127.0.0.1:3320'

The instance '127.0.0.1:3320' was successfully added tc
```

Then add the third instance:

```
MySQL  localhost:3310 ssl  JS >  cluster.addInstance('
```

At this point we have created a cluster with three instances: a primary and two secondaries. We can see the status by running the following command:

```
MySQL  localhost:3310 ssl  JS > cluster.status()
```


```
{
    "clusterName": "learning_mysql",
    "defaultReplicaSet": {
        "name": "default",
        "primary": "127.0.0.1:3310",
        "ssl": "REQUIRED",
        "status": "OK",
        "statusText": "Cluster is ONLINE and can tolera
        "topology": {
            "127.0.0.1:3310": {
                "address": "127.0.0.1:3310",
                "mode": "R/W",
                "readReplicas": {},
                "replicationLag": null,
                "role": "HA",
                "status": "ONLINE",
                "version": "8.0.21"
            },
            "127.0.0.1:3320": {
                "address": "127.0.0.1:3320",
                "mode": "R/O",
                "readReplicas": {},
                "replicationLag": null,
                "role": "HA",
                "status": "ONLINE",
                "version": "8.0.21"
```

```
            },
            "127.0.0.1:3330": {
                "address": "127.0.0.1:3330",
                "mode": "R/O",
                "readReplicas": {},
                "replicationLag": null,
                "role": "HA",
                "status": "ONLINE",
                "version": "8.0.21"
            }
        },
        "topologyMode": "Single-Primary"
    },
    "groupInformationSourceMember": "127.0.0.1:3310"
```

Assuming MySQL Router is already installed (see "MySQL Router"), the only required step is to bootstrap it with the location of the InnoDB Cluster metadata server.

We observe the router being bootstrapped:

```
# mysqlrouter --bootstrap root@localhost:3310 --user=my
```

```
Please enter MySQL password for root:
# Bootstrapping system MySQL Router instance...

- Creating account(s) (only those that are needed, if a
...

## MySQL Classic protocol

- Read/Write Connections: localhost:6446
- Read/Only Connections:  localhost:6447

...
```

MySQL Shell Utilities

As we've said, MySQL Shell is a powerful, advanced client and code editor for MySQL. Among its many functionalities are utilities to create a logical dump and do a logical restore for the entire database instance, including users.

The advantage, compared to `mysqldump`, for example, is that the utility has parallelization capacity, greatly improving the dump and restore speed.

Here are the utilities to execute the dump and restore process:

`util.dumpInstance()`

Dump an entire database instance, including users

`util.dumpSchemas()`

Dump a set of schemas

`util.loadDump()`

Load a dump into a target database

`util.dumpTables()`

Load specific tables and views

Let's take a closer look at each of these in turn.

util.dumpInstance()

The `dumpInstance()` utility will dump all the databases that are present in the MySQL data directory (see "The Contents of the MySQL Directory"). It will exclude the `information_schema`, `mysql_`, `ndbinfo`, `performance_schema`, and `sys` schemas while taking the dump.

There's also a dry-run option that allows you to inspect the schemas and view the compatibility issues and then run the dump with the appropriate compatibility options applied to remove the issues. Let's try this now—we'll examine the possible errors and see the options for the dump utility.

To start the dump, run the following command:

```
MySQL  JS > shell.connect('root@localhost:48008');
MySQL  localhost:48008 ssl  JS > util.dumpInstance("/t
                            > {ocimds: true, compat
                              dryRun: true})
```

```
Acquiring global read lock
Global read lock acquired
Gathering information - done
All transactions have been started
Locking instance for backup
...
NOTE: Database test had unsupported ENCRYPTION option c
ERROR: Table 'test'.'sbtest1' uses unsupported storage
(fix this with 'force_innodb' compatibility option)
Compatibility issues with MySQL Database Service 8.0.23
Please use the 'compatibility' option to apply compatib
to the dumped DDL.
Util.dumpInstance: Compatibility issues were found (Rur
```

With the `ocimds` option set to `true`, the dump utility will check the data dictionary and index dictionary. Encryption options in `CREATE TABLE` statements are commented out in the DDL files, to ensure that all tables are located in the MySQL data directory and use the default schema encryption. `strip_restricted_grants` removes specific privileges that are restricted by MySQL Database Service that would cause an error during the user creation process. `dryRun` is self-explanatory: it will perform validation only, and no data will be actually dumped.

So, we have a MyISAM table in the `test` database. The dry-run option clearly throws the error.

To fix this error, we are going to use the `force_innodb` option, which will convert all unsupported engines to InnoDB in the `CREATE TABLE` statement:

```
MySQL  localhost:48008 ssl  JS > util.dumpInstance("ba
                            > {ocimds: true, compat
                            > ["strip_restricted_gr
                            dryRun: true})
```

Now the dry run does not throw any errors, and there are no exceptions. Let's run the `dumpInstance()` command to take an instance backup. The target directory must be empty before the export takes place. If the directory does not yet exist in its parent directory, the utility creates it.

We are going to process the dump in parallel. For this, we will use the option `threads` and set a value of 10 threads:

```
 MySQL  localhost:48008 ssl  JS > util.dumpInstance("/t
                              > {ocimds: true, compat
                              > ["strip_restricted_gr
                              > threads : 10 })
```

If we observe the last part of the output, we'll see:

```
1 thds dumping - 100% (10.00K rows / ~10.00K rows), 0.0
uncompressed, 0.00 B/s

uncompressed
Duration: 00:00:00s
Schemas dumped: 1
Tables dumped: 10
Uncompressed data size: 1.88 MB
Compressed data size: 598.99 KB
Compression ratio: 3.1
Rows written: 10000
Bytes written: 598.99 KB
Average uncompressed throughput: 1.88 MB/s
Average compressed throughput: 598.99 KB/s
```

If we were using `mysqldump`, we would have a single file. As we can see here, there are multiple files in the backup directory:

```
@.done.json
@.json
@.post.sql
@.sql
test.json
test@sbtest10@@0.tsv.zst
test@sbtest10@@0.tsv.zst.idx
test@sbtest10.json
test@sbtest10.sql
...
test@sbtest1@@0.tsv.zst
test@sbtest1@@0.tsv.zst.idx
test@sbtest1.json
```

```
test@sbtest1.sql
test@sbtest9@@0.tsv.zst
test@sbtest9@@0.tsv.zst.idx
test@sbtest9.json
test@sbtest9.sql
test.sql
```

Let's take a look at these:

- The *@.json* file contains server details and lists of users, database names, and their character sets.
- The *@.post.sql* and *@.sql* files contain MySQL Server version details.
- The *test.json* file contains view, stored procedure, and function names along with a list of tables.
- The *@.users.sql* file (not shown) contains a list of database users.
- The *test@sbtest10.json* file contains column names and character sets. There will be a similiarly named file for each dumped table.
- The *test@sbtest1.sql* file contains a table structure. There will be one for each dumped table.
- The *test@sbtest10@@0.tsv.zst* file is a binary file. It stores data. There will be a similarly named file for each dumped table.
- The *test@sbtest10@@0.tsv.zst.idx* file is a binary file. It stores table index stats. There will be a similarly named file for each dumped table.
- The *@.done.json* file contains the backup end time and data file sizes in KB.
- The *test.sql* file contains a database statement.

util.dumpSchemas()

This utility is similar to `dumpInstance()`, but it allows us to specify schemas to dump. It supports the same options:

```
MySQL  localhost:48008 ssl  JS > util.dumpSchemas(["te
                               > {ocimds: true, compat
                               > ["strip_restricted_gr
                               > threads : 10 , dryRur
```

If we want to specify multiple schemas, we can do that by running:

```
MySQL  localhost:48008 ssl  JS > util.dumpSchemas(["te
                               "learning_mysql"],"/t
                             > {ocimds: true, compat
                             > ["strip_restricted_gr
                             > threads : 10 , dryRur
```

util.dumpTables()

If we want to extract more granular data, like specific tables, we can use the `dumpTables()` utility. Again, the big advantage compared to `mysqldump` is the potential to extract data from MySQL in parallel:

```
MySQL  localhost:48008 ssl  JS > util.dumpTables("test
                             > "sbtest2" ],"/backup'
                             > {ocimds: true, compat
                             > ["strip_restricted_gr
                             > threads : 2 , dryRun:
```

util.loadDump(url[, options])

We've seen all the utilities to extract data, but there is one remaining: the one to load data into MySQL.

The `loadDump()` enables provides data streaming to remote storage, parallel loading of tables or table chunks, and progress state tracking. It also provides resume and reset capabilities and the option of concurrent loading while the dump is still taking place.

Note that this utility uses the `LOAD DATA LOCAL INFILE` statement, so we need to enable the `local_infile` parameter globally while importing.

The `loadDump()` utility checks whether the `sql_require_primary_key` system variable is set to `ON`, and if it is, returns an error if there is a table in the dump files with no primary key:

```
MySQL  localhost:48008 ssl  JS > util.loadDump("/backu
                             > {progressFile :"/back
                             > restore.json",threads
```

The last part of the output will be similar to this:

```
[Worker006] percona@sbtest7@@0.tsv.zst: Records: 400000
Warnings: 0
[Worker007] percona@sbtest4@@0.tsv.zst: Records: 400000
Warnings: 0
[Worker002] percona@sbtest13@@0.tsv.zst: Records: 22074
Warnings: 0
Executing common postamble SQL

23 chunks (5.03M rows, 973.06 MB) for 23 tables in 3 sc
1 min 24 sec (avg throughput 11.58 MB/s)
0 warnings were reported during the load.
```

Be sure to check the warnings reported at the end in case any show up.

Flame Graphs

Quoting Brendan Gregg, determining why CPUs are busy is a routine task for performance analysis, which often involves profiling *stack traces*. Profiling by sampling at a fixed rate is a coarse but effective way to see which code paths are *hot* (busy on the CPU). It usually works by creating a timed interrupt that collects the current program counter, function address, or entire stack trace, and translates these to something human-readable when printing a summary report. *Flame graphs* are a type of visualization for sampled stack traces that allow hot code paths to be identified quickly.

A *stack trace* (also called *stack backtrace* or *stack traceback*) is a report of the active stack frames at a certain point in time during the execution of a program. There are many tools available to collect stack traces. These tools are also known as *CPU profilers*. The CPU profiler we are going to use is `perf` .

`perf` is a profiler tool for Linux 2.6+–based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface. `perf` is based on the `perf_events` interface exported by recent versions of the Linux kernel.

`perf_events` is an event-oriented observability tool that can help solve advanced performance and troubleshooting tasks. Questions that can be answered include:

- Why is the kernel on-CPU so much? What code paths are hot?
- Which code paths are causing CPU level 2 cache misses?
- Are the CPUs stalled on memory I/O?
- Which code paths are allocating memory, and how much?
- What is triggering TCP retransmits?

- Is a certain kernel function being called, and how often?
- Why are threads leaving the CPU?

Note that in this book, we are only scratching the surface of `perf` 's capabilities. We highly recommend checking out Brendan Gregg's website, which contains much more detailed information about `perf` and other CPU profilers.

To produce flame graphs, we need to start collecting the stack trace report with `perf` in the MySQL server. This operation needs to be done on the MySQL host. We will collect data for 60 seconds:

```
# perf record -a -g -F99 -p $(pgrep -x mysqld) -- sleep
# perf report > /tmp/perf.report;
# perf script > /tmp/perf.script;
```

And if we check the */tmp* directory, we will see `perf` files:

```
# ls -l /tmp/perf*
-rw-r--r-- 1 root root  502100 Feb 13 22:01 /tmp/perf.r
-rw-r--r-- 1 root root 7303290 Feb 13 22:01 /tmp/perf.s
```

The next step doesn't need to be executed on the MySQL host; we can copy the files to another Linux host or even macOS.

To produce the flame graphs we can use Brendan's GitHub repository. For this example, we will clone the Flame Graph repository in the directory where our `perf` report is located:

```
# git clone https://github.com/brendangregg/FlameGraph
# ./FlameGraph/stackcollapse-perf.pl ./perf.script > pe
# ./FlameGraph/flamegraph.pl ./perf.report.out.folded >
```

We've produced a file named *perf.report.out.svg*. This file can be opened in any browser to be visualized. Figure 16-1 is an example of a flame graph.

Flame graphs show the sample population across the x-axis, and stack depth on the y-axis. Each function (stack frame) is drawn as a rectangle, with the width relative to the number of samples; so the bigger the bar, the more CPU time was spent on that function. The x-axis spans the stack trace collection but does not show the passage of time, so the left-to-right ordering has no special meaning. The ordering is done alphabetically based on the function names, from the root to the leaf of each stack.

The file that's created is interactive, so we can explore where kernel CPU time is spent. In the previous example an `INSERT` operation is consuming 44% of the CPU time, as you can see in Figure 16-2.

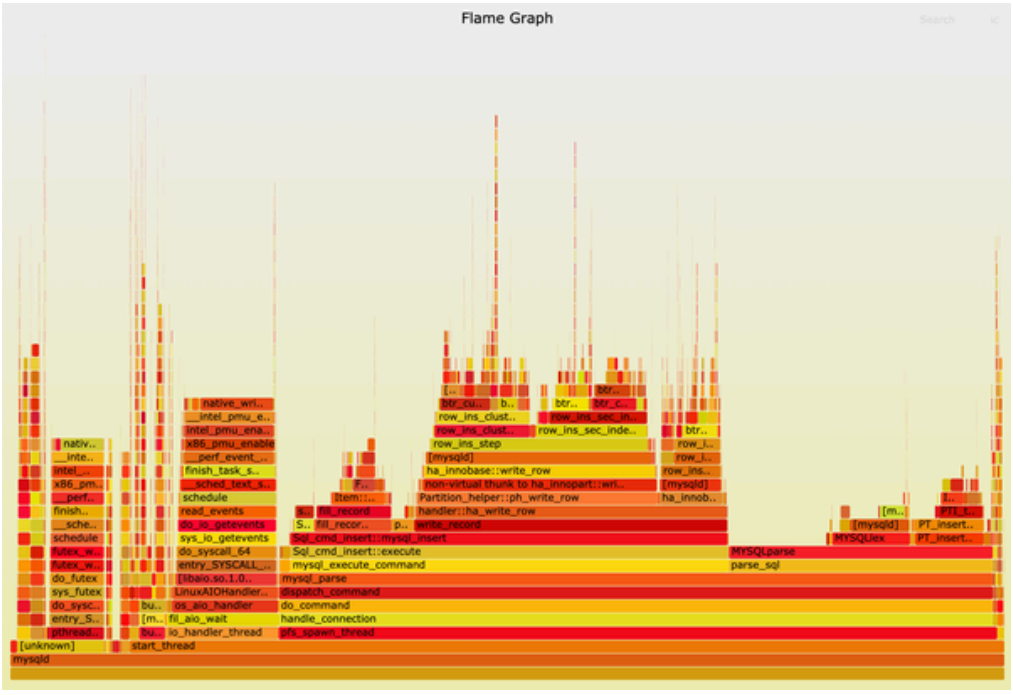

Figure 16-1. An example of a flame graph

Figure 16-2. 44% of CPU time is used for an `INSERT` operation

Building MySQL from Source

As Chapter 1 explained, MySQL has a distribution available for most common operating systems. Some companies have also compiled their own MySQL versions, such as Facebook, which worked on the RocksDB engine and integrated it into MySQL. RocksDB is an embeddable, persistent key/value store for fast storage that has several advantages compared with InnoDB with regard to space efficiency.

Despite its advantages, RocksDB does not support replication or a SQL layer. This led the Facebook team to build MyRocks, an open source project that integrates RocksDB as a MySQL storage engine. With MyRocks, it is possible to use RocksDB as backend storage and still benefit from all the features of MySQL. Facebook's project is open source and available on GitHub.

Another motivation to compile MySQL is the ability to customize its build. For example, for a very specific problem, we can always try to debug MySQL to gather extra information. To do this, we need to configure MySQL with the `-DWITH_DEBUG=1` option.

Building MySQL for Ubuntu Focal Fossa and ARM Processors

Because ARM processors are currently gaining traction (particularly thanks to Apple's M1 chip), we will show you how to compile MySQL for Ubuntu Focal Fossa running on ARM.

First, we are going to create our directories. We will create one directory that will be for the source code, another one for the compiled binaries, and a third for the `boost` library:

```
# cd /
# mkdir compile
# cd compile/
# mkdir build
# mkdir source
# mkdir boost
# mkdir basedir
# mkdir /var/lib/mysql
```

Next, we need to install the additional Linux packages required to compile MySQL:

```
# apt-get -y install dirmngr
# apt-get update -y
# apt-get -y install cmake
# apt-get -y install lsb-release wget
# apt-get -y purge eatmydata || true
# apt-get -y install psmisc pkg-config
# apt-get -y install libsasl2-dev libsasl2-modules libs
    apt-get -y install libsasl2-modules libsasl2-module
# apt-get -y install dh-systemd || true
# apt-get -y install curl bison cmake perl libssl-dev g
    libldap2-dev libwrap0-dev gdb unzip gawk
# apt-get -y install lsb-release libmecab-dev libncurse
    libpam-dev zlib1g-dev
# apt-get -y install libldap2-dev libnuma-dev libjemall
    libc6-dbg valgrind libjson-perl  libsasl2-dev
# apt-get -y install libmecab2 mecab mecab-ipadic
# apt-get -y install build-essential devscripts libnuma
# apt-get -y install cmake autotools-dev autoconf autom
    devscripts debconf debhelper fakeroot
# apt-get -y install libcurl4-openssl-dev patchelf
# apt-get -y install libeatmydata1
# apt-get install libmysqlclient-dev -y
# apt-get install valgrind -y
```

These packages are related to the CMake flags that we will run. If we remove or add certain flags, some packages are not necessary to install (for example, if we don't want to compile with Valgrind, we don't need this package).

Next, we will download the source code. For this, we will use MySQL repository on GitHub:

```
# cd source
# git clone https://github.com/mysql/mysql-server.git
```

The output will be similar to this:

```
Cloning into 'mysql-server'...
remote: Enumerating objects: 1639611, done.
```

```
remote: Total 1639611 (delta 0), reused 0 (delta 0), pa
Receiving objects: 100% (1639611/1639611), 3.19 GiB | 4
Resolving deltas: 100% (1346714/1346714), done.
Updating files: 100% (32681/32681), done.
```

To check which version we will compile, we can run the following:

```
# cd mysql-server/
# git branch
```

Next, we will go to our *build* directory and run `CMake` with our chosen flags:

```
# cd /compile/build
# cmake ../source/mysql-server/  -DBUILD_CONFIG=mysql_r
    -DCMake_BUILD_TYPE=${CMake_BUILD_TYPE:-RelWithDebIr
    -DWITH_DEBUG=1 \
    -DFEATURE_SET=community \
    -DENABLE_DTRACE=OFF \
    -DWITH_SSL=system \
    -DWITH_ZLIB=system \
    -DCMake_INSTALL_PREFIX="/compile/basedir/" \
    -DINSTALL_LIBDIR="lib/" \
    -DINSTALL_SBINDIR="bin/" \
    -DWITH_INNODB_MEMCACHED=ON \
    -DDOWNLOAD_BOOST=1 \
    -DWITH_VALGRIND=1 \
    -DINSTALL_PLUGINDIR="plugin/" \
    -DMYSQL_DATADIR="/var/lib/mysql/" \
    -DWITH_BOOST="/compile/boost/"
```

Here's what each of these does:

- `DBUILD_CONFIG` configures a source distribution with the same build options as for MySQL releases (we are going to override some of them).
- `DCMake_BUILD_TYPE` with `RelWithDebInfo` enables optimizations and generates debugging information.
- `DWITH_DEBUG` enables the use of the `--debug="d,parser_debug"` option when MySQL is started. This causes the Bison parser used to process SQL statements to dump a parser trace to the server's standard error output. Typically, this output is written to the error log.
- `DFEATURE_SET` indicates we are going to install community features.

- `DENABLE_DTRACE` includes support for DTrace probes. The DTrace probes in the MySQL server are designed to provide information about the execution of queries within MySQL and the different areas of the system being utilized during that process.
- The `DWITH_SSL` option adds support for encrypted connections, entropy for random number generation, and other encryption-related operations.
- `DWITH_ZLIB` enables compression library support for the `COMPRESS()` and `UNCOMPRESS()` functions, and compression of the client/server protocol.
- `DCMake_INSTALL_PREFIX` sets the location of our installation base directory.
- `DINSTALL_LIBDIR` indicates where to install the library files.
- `DINSTALL_SBINDIR` specifies where to install `mysqld` .
- `DWITH_INNODB_MEMCACHED` generates memcached shared libraries (*libmemcached.so* and *innodb_engine.so*).
- `DDOWNLOAD_BOOST` makes CMake download the `boost` library and place it in the location specified with `DWITH_BOOST` .
- `DWITH_VALGRIND` enables Valgrind, exposing the Valgrind API to MySQL code. This is useful for analyzing memory leaks.
- `DINSTALL_PLUGINDIR` defines where the compiler will place the plugin libraries.
- `DMYSQL_DATADIR` defines the location of the MySQL data directory.
- `DWITH_BOOST` defines the directory where CMake will download the `boost` library.

NOTE

If by mistake you miss a step and the CMake process fails, to prevent old object files or configuration information from being used in the next attempt you'll need to clean up the build directory and the previous configuration. That is, you'll need to run the following commands in the build directory on Unix before rerunning CMake:

```
# cd /compile/build
# make clean
# rm CMakeCache.txt
```

After we run CMake, we are going to compile MySQL using the `make` command. To optimize the compiling process we will use the `-j` option, which specifies how many threads we are going to use to compile MySQL.

Since in our instance we have 16 ARM cores, we are going to use 15 threads (leaving one for OS activities):

```
# make  -j 15
# make install
```

This process may take a while, and it is very verbose. After it's finished, we can see the binaries in the *basedir* directory:

```
# ls -l /compile/basedir/bin
```

Note that we are not going to find a *mysqld* binary in the */compile/build/bin/* directory, but instead we will see *mysqld-debug*. This is because of the `DWITH_DEBUG` option we set previously:

```
# /compile/build/bin/mysqld-debug --version


/compile/build/bin/mysqld-debug  Ver 8.0.23-debug-valgr
(Source distribution)
```

Now, we can test our binary. For this we are going to manually create the directories and configure the permissions:

```
# mkdir /var/log/mysql/
# mkdir /var/run/mysqld/
# chown ubuntu: /var/log/mysql/
# chown ubuntu: /var/run/mysqld/
```

Then add these settings to */etc/my.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
```

Next, we are going to initialize the MySQL data dictionary:

```
# /compile/basedir/bin/mysqld-debug --defaults-file=/et
--user ubuntu
```

Now, MySQL is ready to be started:

```
# /compile/basedir/bin/mysqld-debug --defaults-file=/et
```

A temporary password will be created, and we can extract it from the error log:

```
# grep "A temporary password" /var/log/mysql/error.log
```

```
2021-02-14T16:55:25.754028Z 6 [Note] [MY-010454] [Serve
password is generated for root@localhost: yGldRKoRf0%T
```

Now we can connect using the MySQL client of our preference:

```
# mysql -uroot -p'yGldRKoRf0%T'
```

```
mysql: [Warning] Using a password on the command line i
insecure. Welcome to the MySQL monitor. Commands end wi
Your MySQL connection id is 8
Server version: 8.0.23-debug-valgrind

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation
affiliates. Other names may be trademarks of their resp
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the c

mysql>
```

Analyzing a MySQL Crash

We say that MySQL *crashes* when the `mysqld` process dies without the proper shutdown command. MySQL can crash for a variety of reasons, including these:

- Hardware failure (memory, disk, processor)
- Segmentation faults (invalid memory access)
- Bugs
- Being killed by the `OOM` process
- Various other causes, such as cosmic rays.

The MySQL process can receive a number of signals from Linux. The following are among the most common:

Signal 15 (`SIGTERM`)

Causes the server to shut down. This is like executing a `SHUTDOWN` statement without having to connect to the server (which for shutdown requires an account that has the `SHUTDOWN` privilege). For example, the following two commands result in a regular shutdown:

```
# systemctl stop mysql
# kill -15 -p $(pgrep -x mysqld)
```

Signal 1 (`SIGHUP`)

Causes the server to reload the grant tables and to flush tables, logs, the thread cache, and the host cache. These actions are like various forms of the `FLUSH` statement:

```
mysql> FLUSH LOGS;
```

or:

```
# kill -1 -p $(pgrep -x mysqld)
```

Signal 6 (`SIGABRT`)

Happens because something went wrong. It is commonly used by `libc` and other libraries to abort the program in case of critical errors. For example, `glibc` sends a `SIGABRT` if it detects a double free or other heap corruption. `SIGABRT` will write the crash details in the MySQL error log, like this:

```
18:03:28 UTC - mysqld got signal 6 ;
Most likely, you have hit a bug, but this error c
Thread pointer: 0x7fe6b4000910
Attempting backtrace. You can use the following i
where mysqld died. If you see no messages after t
terribly wrong...
stack_bottom = 7fe71845fbc8 thread_stack 0x46000
/opt/mysql/8.0.23/bin/mysqld(my_print_stacktrace(
/opt/mysql/8.0.23/bin/mysqld(handle_fatal_signal+
/lib64/libpthread.so.0(+0xf630) [0x7fe7244e5630]
/lib64/libc.so.6(gsignal+0x37) [0x7fe7224fa387]
/lib64/libc.so.6(abort+0x148) [0x7fe7224fba78]
/opt/mysql/8.0.23/bin/mysqld() [0xd52c3d]
/opt/mysql/8.0.23/bin/mysqld(MYSQL_BIN_LOG::new_f
/opt/mysql/8.0.23/bin/mysqld(MYSQL_BIN_LOG::rotat
/opt/mysql/8.0.23/bin/mysqld(MYSQL_BIN_LOG::rotat
/opt/mysql/8.0.23/bin/mysqld(handle_reload_reques
/opt/mysql/8.0.23/bin/mysqld(signal_hand+0x2ea) [
/opt/mysql/8.0.23/bin/mysqld() [0x25973dc]
/lib64/libpthread.so.0(+0x7ea5) [0x7fe7244ddea5]
/lib64/libc.so.6(clone+0x6d) [0x7fe7225c298d]

Trying to get some variables.
Some pointers may be invalid and cause the dump t
Query (0): Connection ID (thread ID): 0
Status: NOT_KILLED

The manual page at http://dev.mysql.com/doc/mysql
contains information that should help you find ou
the crash.
2021-02-14T18:03:29.120726Z mysqld_safe mysqld fr
```

Signal 11 (`SIGSEGV`)

Indicates a segmentation fault, bus error, or access violation issue. This is generally an attempt to access memory that the CPU cannot physically address, or an access violation. When MySQL receives a

`SIGSEGV`, a core dump will be created if the `core-file` parameter is configured.

Signal 9 (`SIGKILL`)

Causes a process to terminate immediately (kills it). This is probably the most famous signal. In contrast to `SIGTERM` and `SIGINT`, this signal cannot be caught or ignored, and the receiving process cannot perform any cleanup upon receiving this signal. Besides the chance of corrupting MySQL data, `SIGKILL` will also force MySQL to perform a recovery process when restarted to bring it to an operational state. The following example shows how to send a `SIGKILL` manually to the MySQL process:

```
# kill -9 -p $(pgrep -x mysqld)
```

Also, the Linux `OOM` process executes a `SIGKILL` to terminate with the MySQL process.

Let's try to analyze a crash where MySQL got a signal 11:

```
11:47:47 UTC - mysqld got signal 11 ;
Most likely, you have hit a bug, but this error can als
Build ID: Not Available
Server Version: 8.0.22-13 Percona Server (GPL), Release
Thread pointer: 0x7f0e46c73000
Attempting backtrace. You can use the following informa
where mysqld died. If you see no messages after this, s
terribly wrong...
stack_bottom = 7f0e664ecd10 thread_stack 0x46000
/usr/sbin/mysqld(my_print_stacktrace(unsigned char cons
/usr/sbin/mysqld(handle_fatal_signal+0x3c3) [0x1260d33]
/lib/x86_64-linux-gnu/libpthread.so.0(+0x128a0) [0x7f0e
/usr/sbin/mysqld(Item_splocal::this_item()+0x14) [0xe36
/usr/sbin/mysqld(Item_sp_variable::val_str(String*)+0x2
/usr/sbin/mysqld(Arg_comparator::compare_string()+0x27)
/usr/sbin/mysqld(Item_func_ne::val_int()+0x30) [0xe580e
/usr/sbin/mysqld(Item::val_bool()+0xcc) [0xe3ddbc]
/usr/sbin/mysqld(sp_instr_jump_if_not::exec_core(THD*,
/usr/sbin/mysqld(sp_lex_instr::reset_lex_and_exec_core(
/usr/sbin/mysqld(sp_lex_instr::validate_lex_and_execute
/usr/sbin/mysqld(sp_head::execute(THD*, bool)+0x5c7) [0
/usr/sbin/mysqld(sp_head::execute_trigger(THD*, MYSQL_L
```

```
/usr/sbin/mysqld(Trigger::execute(THD*)+0x10b) [0x12288
/usr/sbin/mysqld(Trigger_chain::execute_triggers(THD*)+
/usr/sbin/mysqld(Table_trigger_dispatcher::process_trig
/usr/sbin/mysqld(fill_record_n_invoke_before_triggers(T
/usr/sbin/mysqld(Sql_cmd_update::update_single_table(TH
/usr/sbin/mysqld(Sql_cmd_update::execute_inner(THD*)+0x
/usr/sbin/mysqld(Sql_cmd_dml::execute(THD*)+0x6c0) [0x1
/usr/sbin/mysqld(mysql_execute_command(THD*, bool)+0xaf
/usr/sbin/mysqld(mysql_parse(THD*, Parser_state*, bool)
/usr/sbin/mysqld(dispatch_command(THD*, COM_DATA const*
/usr/sbin/mysqld(do_command(THD*)+0x204) [0x1116554]
/usr/sbin/mysqld() [0x1251c20]
/usr/sbin/mysqld() [0x2620e84]
/lib/x86_64-linux-gnu/libpthread.so.0(+0x76db) [0x7f0e7
/lib/x86_64-linux-gnu/libc.so.6(clone+0x3f) [0x7f0e78c9
Trying to get some variables.
Some pointers may be invalid and cause the dump to abor
Query (7f0e46cb4dc8): update table1 set c2_id='R', c3_c
Connection ID (thread ID): 111
Status: NOT_KILLED
Please help us make Percona Server better by reporting
bugs at https://bugs.percona.com/
```

NOTE

Sometimes stack may not contain fully resolved symbols or may only have addresses. That depends on whether the mysqld binary is stripped and whether the debug symbols are available. As a rule of thumb, we recommend installing debug symbols, as that has no disadvantage other than using up some disk space. Having debug symbols installed doesn't make your MySQL server run in some slow debug mode. Official MySQL 8.0 builds are always symbolized, however, so you don't need to worry.

The stack trace is analyzed from top to bottom. We can see from the crash that this is a Percona Server v8.0.22. Next, we see the thread being created at the OS level at this point:

```
/lib/x86_64-linux-gnu/libpthread.so.0(+0x76db) [0x7f0e7
```

Continuing up through the stack, the code path enters MySQL and starts executing a command:

```
/usr/sbin/mysqld(do_command(THD*)+0x204)...
```

And the code path that crashes is the `Item_splocal` function:

```
/usr/sbin/mysqld(Item_splocal::this_item()+0x...
```

With a bit of investigation in the MySQL code, we discover that `Item_splocal` is part of the stored procedure code. If we look at the end of the stack trace, we will see a query:

```
Query (7f0e46cb4dc8): update table1 set c2_id='R', c3_c
```

Triggers can also use the stored procedure path when they contain variables. If we check whether this table has triggers, we see this:

```
CREATE DEFINER=`root`@`localhost` TRIGGER `table1_updat
BEFORE UPDATE ON `table1` FOR EACH ROW BEGIN
DECLARE vc1_id VARCHAR(2);
SELECT c2_id FROM table1 WHERE c1_id = new.c1_id INTO v
IF vc1_id <> P THEN
INSERT INTO table1_hist(
c1_id,
c2_id,
c3_description)
VALUES(
old.c1_id,
old.c2_id,
new.c3_description);
END IF;
END
;;
```

With all this information, we can create a test case and report the bug:

```
USE test;

CREATE TABLE `table1` (
  `c1_id` int primary key auto_increment,
  `c2_id` char(1) NOT NULL,
```

```
    `c3_description` varchar(255));

CREATE TABLE `table1_hist` (
  `c1_id` int,
  `c2_id` char(1) NOT NULL,
  `c3_description` varchar(255));
  insert into table1 values (1, T,  test crash);

delimiter ;;

CREATE DEFINER=`root`@`localhost` TRIGGER `table1_updat
BEFORE UPDATE ON `table1` FOR EACH ROW BEGIN
DECLARE vc1_id VARCHAR(2);
SELECT c2_id FROM table1 WHERE c1_id = new.c1_id INTO v
IF vc1_id <> P THEN
INSERT INTO table1_hist(
c1_id,
c2_id,
c3_description)
VALUES(
old.c1_id,
old.c2_id,
new.c3_description);
END IF;
END
;;
```

To reproduce it, we run multiple commands simultaneously in the same table until the error happens:

```
$ mysqlslap --user=msandbox --password=msandbox \
    --socket=/tmp/mysql_sandbox37515.sock \
    --create-schema=test --port=37515 \
    --query="update table1 set c2_id='R',
    *c3_description='testing crash' where c1_id=1" \
    --concurrency=50 --iterations=200
```

This bug is relatively easy to reproduce, and we recommend you test it. You can find more details about this bug in Percona's Jira system.

Also, we can see that Oracle fixed the bug at version 8.0.23 thanks to the release notes:

Prepared statements involving stored programs could cause heap-use-after-free memory problems (Bug #32131022, Bug #32045681, Bug #32051928).

Sometimes bugs are not easy to reproduce and can be really frustrating to investigate. Even experienced engineers have problems with this, especially when investigating memory leaks. We hope we have sparked your curiosity to investigate crashes.