Chapter 11. Configuring and Tuning the Server

The MySQL installation process (see Chapter 1) provides everything necessary to install the MySQL process and start using it. However, it is required for production systems to do some fine-tuning, adjusting MySQL parameters and the operating system to optimize MySQL Server's performance. This chapter will cover the recommended best practices for different installations and show you the parameters that need to be adjusted based on the expected or current workload. As you'll see, it is not necessary to memorize all the MySQL parameters. Based on the *Pareto principle*, which states that, for many events, roughly 80% of the effects come from 20% of the causes, we will concentrate on the MySQL and operating system parameters that are responsible for most of the performance issues. There are some advanced topics in this chapter related to computer architecture (such as NUMA); the intent here is to introduce you to a few components that can affect MySQL performance that you will need to interact with sooner or later in your career.

The MySQL Server Daemon

Since 2015, the majority of Linux distributions have adopted `systemd`. Because of that, Linux operating systems do not use the `mysqld_safe` process to start MySQL anymore. `mysqld_safe` is called an *angel process*, because it adds some safety features, such as restarting the server when an error occurs and logging runtime information to the MySQL error log. For operating systems that use `systemd` (controlled and configured with the `systemctl` command), these functionalities have been incorporated into `systemd` and the `mysqld` process.

`mysqld` is the core process of MySQL Server. It is a single multithreaded program that does most of the work in the server. It does not spawn additional processes—we're talking about a single process with multiple threads, making MySQL a *multithreaded process*.

Let’s take a closer look at some of those terms. A *program* is code that is designed to accomplish a specific objective. There are many types of programs, including ones to assist parts of the operating system and others that are designed for user needs, such as web browsing.

A *process* is what we call a program that has been loaded into memory along with all the resources it needs to operate. The operating system allocates memory and other resources for it.

A *thread* is the unit of execution within a process. A process can have just one thread or many threads. In single-threaded processes, the process contains one thread, so only one command is executed at a time.

Because modern CPUs have multiple cores, they can execute multiple threads at the same time, so multithreaded processes are widespread nowadays. It’s important to be aware of this concept to understand some of the proposed settings in the following sections.

To conclude, MySQL is single-process software that spawns multiple threads for various purposes, such as serving user activities and executing background tasks.

MySQL Server Variables

MySQL Server has many variables that allow tuning its operation. For example, MySQL Server 8.0.25 has an impressive *588* server variables!

Each system variable has a default value. Also, we can adjust most system variables dynamically (or “on the fly”); however, a few of them are static, which means that we need to change the *my.cnf* file and restart the MySQL process so they can take effect (as discussed in Chapter 9).

The system variables can have two different scopes: `SESSION` and `GLOBAL`. That is, a system variable can have a global value that affects server operation as a whole, like the `innodb_log_file_size`, or a session value that affects only a specific session, like the `sql_mode`.

Checking Server Settings

Databases are not static entities; on the contrary, their workload is dynamic and changes over time, with a tendency to growth. This organic behavior requires constant monitoring, analysis, and adjustment. The command to show the MySQL settings is:

```
SHOW [GLOBAL|SESSION] VARIABLES;
```

When you use the `GLOBAL` modifier, the statement displays global system variable values. When you use `SESSION`, it displays the system variable values that affect the current connection. Observe that different connections can have different values.

If no modifier is present, the default is `SESSION`.

Best Practices

There are many aspects to optimize in a database. If the database runs on *bare metal* (a physical host), we can control hardware and operating system resources. When we move to virtualized machines, we have reduced control over these resources because we can't control what happens with the underlying host. The last option is managed databases in the cloud, like those provided by Amazon Relational Database Service (RDS), where only a few database settings are available. There's a trade-off between being able to perform fine-grained tuning to extract the most performance and the comfort of having most of the tasks automated (at the cost of a few extra dollars).

Let's start by reviewing some settings at the operating system level. After that, we will check out the MySQL parameters.

Operating system best practices

There are several operating system settings that can affect the performance of MySQL. We'll run through some of the most important ones here.

The swappiness setting and swap usage

The `swappiness` parameter controls the behavior of the Linux operating system in the swap area. Swapping is the process of transferring data between memory and the disk. This can have a significant effect on performance, because disk access (even with NVMe disks) is at least an order of magnitude slower than memory access.

The default setting (`60`) encourages the server to swap. You will want your MySQL server to keep swapping to a minimum for performance reasons. The recommended value is `1` , which means do not swap until it is absolutely necessary for the OS to be functional. To adjust this parameter, execute the following command as root:

```
# echo 1 > /proc/sys/vm/swappiness
```

Note that this is a nonpersistent change; the setting will revert to its original value when you reboot the OS. To make this change persistent after an operating system reboot, adjust the setting in *sysctl.conf*:

```
# sudo sysctl -w vm.swappiness=1
```

You can get information on swap space usage using the following command:

```
# free -m
```

Or, if you want more detailed information, you can run the following snippet in the shell:

```
#!/bin/bash
SUM=0
OVERALL=0
for DIR in `find /proc/ -maxdepth 1 -type d | egrep "^/
        PID=`echo $DIR | cut -d / -f 3`
        PROGNAME=`ps -p $PID -o comm --no-headers`
        for SWAP in `grep Swap $DIR/smaps 2>/dev/null|
        do
                let SUM=$SUM+$SWAP
        done
        echo "PID=$PID - Swap used: $SUM - ($PROGNAME )
        let OVERALL=$OVERALL+$SUM
        SUM=0
```

```
done
echo "Overall swap used: $OVERALL"
```

NOTE

The difference between setting `vm.swappiness` to `1` and `0` is negligible. We chose the value of `1` because in some kernels there is a bug that can lead the Out of Memory (OOM) Killer to terminate MySQL when it's set to `0`.

I/O scheduler

The I/O scheduler is an algorithm the kernel uses to commit reads and writes to disk. By default, most Linux installs use the Completely Fair Queuing (`cfq`) scheduler. This works well for many general use cases, but offers few latency guarantees. Two other schedulers are `deadline` and `noop`. The `deadline` scheduler excels at latency-sensitive use cases (like databases), and `noop` is closer to no scheduling at all. For bare-metal installations, either `deadline` or `noop` (the performance difference between them is imperceptible) will be better than `cfq`.

If you are running MySQL in a VM (which has its own I/O scheduler), it is best to use `noop` and let the virtualization layer take care of the I/O scheduling itself.

First, verify which algorithm is currently in use by Linux:

```
#  cat /sys/block/xvda/queue/scheduler
```

```
noop [deadline] cfq
```

To change it dynamically, run this command as root:

```
# echo "noop" > /sys/block/xvda/queue/scheduler
```

In order to make this change persistent, you need to edit the GRUB configuration file (usually */etc/sysconfig/grub*) and add the `elevator` option to `GRUB_CMDLINE_LINUX_DEFAULT` . For example, you would replace this line:

```
GRUB_CMDLINE_LINUX="console=tty0 crashkernel=auto consc
```

with this line:

```
GRUB_CMDLINE_LINUX="console=tty0 crashkernel=auto consc
    elevator=noop"
```

It is essential to take extra care when editing the GRUB config. Errors or incorrect settings can make the server unusable and require installing the operating system again.

NOTE

There are cases where the I/O scheduler has a value of `none` —most notably in AWS VM instance types where EBS volumes are exposed as NVMe block devices. This is because the setting has no use in modern PCIe/NVMe devices, which have a substantial internal queue and bypass the I/O scheduler altogether. The `none` setting is optimal in such disks.

Filesystems and mount options

Choosing the filesystem appropriate for your database is an important decision due to the many options available and the trade-offs involved. It is worth mentioning two important ones that are frequently used: *XFS* and *ext4*.

XFS is a high-performance journaling filesystem designed for high scalability. It provides near-native I/O performance even when the filesystem spans multiple storage devices. XFS has features that make it suitable for very large filesystems, supporting files up to 8 EiB in size. Other features include fast recovery, fast transactions, delayed allocation for reduced fragmentation, and near-raw I/O performance with direct I/O.

The make filesystem XFS command (`mkfs.xfs`) has several options to configure the filesystem. However, the default options for `mkfs.xfs` are good for optimal speed, so the default command to create the filesystem will provide good performance while ensuring data integrity:

```
# mkfs.xfs /dev/target_volume
```

Regarding the filesystem mount options, the defaults again should fit most cases. You may see a performance increase on some filesystems by adding the `noatime` mount option to the */etc/fstab* file. For XFS filesystems, the default `atime` behavior is `relatime`, which has almost no overhead compared to `noatime` and still maintains sane `atime` values. If you create an XFS filesystem on a logical unit number (LUN) that has a battery-backed, nonvolatile cache, you can further increase the filesystem's performance by disabling the write barrier with the mount option `nobarrier`. These settings help you avoid flushing data more often than necessary. If a backup battery unit (BBU) is not present, however, or you are unsure about it, leave barriers on; otherwise, you may jeopardize data consistency. The example below shows two imaginary mountpoints with these options:

```
/dev/sda2              /datastore              xfs
/dev/sdb2              /binlog                 xfs
```

The other popular option is ext4, developed as the successor to ext3 with added performance improvements. It is a solid option that will fit most workloads. We should note here that it supports files up to 16 TB in size, a smaller limit than XFS. This is something you should consider if excessive tablespace size/growth is a requirement. Regarding mount options, the same considerations apply. We recommend the defaults for a robust filesystem without risks to data consistency. However, if an enterprise storage controller with a BBU cache is present, the following mount options will provide the best performance:

```
/dev/sda2              /datastore              ext4
noatime,data=writeback,barrier=0,nobh,errors=remount-ro
/dev/sdb2              /binlog                 ext4
noatime,data=writeback,barrier=0,nobh,errors=remount-ro
```

Transparent Huge Pages

The operating system manages memory in blocks known as *pages*. A page has a size of 4,096 bytes (or 4 KB); 1 MB of memory is equal to 256 pages, 1 GB of memory is equivalent to 256,000 pages, etc. CPUs have a built-in memory management unit that contains a list of these pages, with each page referenced through a *page table entry*. It is common to see servers nowadays with hundreds or terabytes of memory. There are two ways to enable the system to manage large amounts of memory:

- Increase the number of page table entries in the hardware memory management unit.
- Increase the page size.

The first method is expensive since the hardware memory management unit in a modern processor only supports hundreds or thousands of page table entries. Besides, hardware and memory management algorithms that work well with thousands of pages (megabytes of memory) may have problems performing well with millions (or even billions) of pages. To address the scalability issue, operating systems started using huge pages. Simply put, huge pages are blocks of memory that can come in sizes of 2 MB, 4 MB, 1 GB size, etc. Using huge page memory increases the CPU cache hits against the transaction lookaside buffer (TLB).

You can run `cpuid` to verify the processor cache and TLB:

```
# cpuid | grep "cache and TLB information" -A 5
   cache and TLB information (2):
      0x5a: data TLB: 2M/4M pages, 4-way, 32 entries
      0x03: data TLB: 4K pages, 4-way, 64 entries
      0x76: instruction TLB: 2M/4M pages, fully, 8 entr
      0xff: cache data is in CPUID 4
      0xb2: instruction TLB: 4K, 4-way, 64 entries
```

Transparent Huge Pages (THP), as the name suggests, is intended to bring huge page support automatically to applications without requiring custom configuration.

For MySQL in particular, using THP is not recommended, for a couple of reasons. First, MySQL databases use small memory pages (16 KB), and using

THP can cause excessive I/O because MySQL believes it is accessing 16 KB while THP is scanning a page larger than that. Also, the huge pages tend to become fragmented and impact performance. There have also been some cases reported over the years where using THP can result in memory leaking, eventually crashing MySQL.

To disable THP for RHEL/CentOS 6 and RHEL/CentOS 7, execute the following commands:

```
# echo "never" > /sys/kernel/mm/transparent_hugepage/er
# echo "never" > /sys/kernel/mm/transparent_hugepage/de
```

To ensure that this change will survive a server restart, you'll have to add the flag `transparent_hugepage=never` to your kernel options (*/etc/sysconfig/grub*):

```
GRUB_CMDLINE_LINUX="console=tty0 crashkernel=auto consc
elevator=noop transparent_hugepage=never"
```

Back up the existing GRUB2 configuration file (*/boot/grub2/grub.cfg*), and then rebuild it. On BIOS-based machines, you can do this with the following command:

```
# grub2-mkconfig -o /boot/grub2/grub.cfg
```

If THP is still not disabled, it may be necessary to disable the `tuned` services:

```
# systemctl stop tuned
# systemctl disable tuned
```

To disable THP for Ubuntu 20.04 (Focal Fossa), we recommend you use the `sysfsutils` package. To install it, execute the following command:

```
# apt install sysfsutils
```

Then append the following lines to the */etc/sysfs.conf* file:

```
kernel/mm/transparent_hugepage/enabled = never
kernel/mm/transparent_hugepage/defrag = never
```

Reboot the server and check if the settings are in place:

```
# cat /sys/kernel/mm/transparent_hugepage/enabled
always madvise [never]
# cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise madvise [never]
```

jemalloc

MySQL Server uses dynamic memory allocation, so a good memory allocator is important for proper CPU and RAM resource utilization. An efficient memory allocator should improve scalability, increase throughput, and keep the memory footprint under control.

It is important to mention a characteristic of InnoDB here. InnoDB creates a read view for every transaction and allocates memory for this structure from the `heap` area. The problem is that MySQL deallocates the heap on each commit, and thus the read view memory is reallocated on the next transaction, leading to memory fragmentation.

`jemalloc` is a memory allocator that emphasizes fragmentation avoidance and scalable concurrency support.

Using `jemalloc` (with THP disabled), you have less memory fragmentation and more efficient resource management of the available server memory. You can install the `jemalloc` package from the `jemalloc` repository or the Percona *yum* or *apt* repository. We prefer to use the Percona repository because we consider it simpler to install and manage. We describe the steps to install the *yum* repository in "Installing Percona Server 8.0" and the *apt* repository in "Installing Percona Server 8".

Once you have the repo, you run the install command for your operating system.

TIP

In CentOS, if the server has the Extra Packages for Enterprise Linux (EPEL) repository installed, it is possible to install `jemalloc` from this repo with `yum`. To install the EPEL package, use:

```
# yum install epel-release -y
```

If you are using Ubuntu 20.04, then you need to execute the following steps to enable `jemalloc`:

1. Install `jemalloc`:

   ```
   # apt-get install libjemalloc2
   # dpkg -L libjemalloc2
   ```

2. The `dpkg` command will show the location of the `jemalloc` library:

   ```
   # dpkg -L libjemalloc2
   /.
   /usr
   /usr/lib
   /usr/lib/x86_64-linux-gnu
   /usr/lib/x86_64-linux-gnu/libjemalloc.so.2
   /usr/share
   /usr/share/doc
   /usr/share/doc/libjemalloc2
   /usr/share/doc/libjemalloc2/README
   /usr/share/doc/libjemalloc2/changelog.Debian.gz
   /usr/share/doc/libjemalloc2/copyright
   ```

3. Override the default configuration of the service with the command:

   ```
   # systemctl edit mysql
   ```

   which will create the */etc/systemd/system/mysql.service.d/override.conf* file.
4. Add the following configuration to the file:

```
[Service]
Environment="LD_PRELOAD=/usr/lib/x86_64-linux-gnu/li
```

5. Restart the MySQL service to enable the `jemalloc` library:

   ```
   # systemctl restart mysql
   ```

6. To verify if it worked, with the `mysqld` process running, execute the following command:

   ```
   # lsof -Pn -p $(pidof mysqld) |  grep "jemalloc"
   ```

   You should see similar output to the following:

   ```
   mysqld  3844 mysql  mem      REG              253,0
   /usr/lib/x86_64-linux-gnu/libjemalloc.so.2
   ```

If you are using CentOS/RHEL, you need to execute the following steps:

1. Install the `jemalloc` package:

   ```
   # yum install jemalloc
   # rpm -ql jemalloc
   ```

2. The `rpm -ql` command will show the library location:

   ```
   /usr/bin/jemalloc.sh
   /usr/lib64/libjemalloc.so.1
   /usr/share/doc/jemalloc-3.6.0
   /usr/share/doc/jemalloc-3.6.0/COPYING
   /usr/share/doc/jemalloc-3.6.0/README
   /usr/share/doc/jemalloc-3.6.0/VERSION
   /usr/share/doc/jemalloc-3.6.0/jemalloc.html
   ```

3. Override the default configuration of the service with the command:

   ```
   # systemctl edit mysqld
   ```

which will create the */etc/systemd/system/mysqld.service.d/override.conf* file.

4. Add the following configuration to the file:

```
[Service]
Environment="LD_PRELOAD=/usr/lib64/libjemalloc.so.1"
```

5. Restart the MySQL service to enable the `jemalloc` library:

```
# systemctl restart mysqld
```

6. To verify if it worked, with the `mysqld` process running, execute the following command:

```
# lsof -Pn -p $(pidof mysqld) |  grep "jemalloc"
```

You should see similar output to the following:

```
mysqld  4784 mysql  mem       REG               253,0
/usr/lib64/libjemalloc.so.1
```

CPU governor

One of the most effective ways to reduce the power consumption and heat output on your system is to use CPUfreq. CPUfreq, also referred to as CPU frequency scaling or CPU speed scaling, allows the processor's clock speed to be adjusted on the fly. This feature enables the system to run at a reduced clock speed to save power. The rules for shifting frequencies—whether and when to shift to a faster or slower clock speed—are defined by the CPUfreq *governor*. The governor defines the power characteristics of the system CPU, which in turn affects CPU performance. Each governor has its own unique behavior, purpose, and suitability in terms of workload. However, for MySQL databases, we recommend using the maximum performance setting to achieve the best throughput.

For CentOS, you can view which CPU governor is currently being used by executing:

```
# cat /sys/devices/system/cpu/cpu/cpufreq/scaling_gover
```

You can enable performance mode by running:

```
# cpupower frequency-set --governor performance
```

For Ubuntu, we recommend installing the `linux-tools-common` package so you have access to the `cpupower` utility:

```
# apt install linux-tools-common
```

Once you've installed it, you can change the governor to performance mode with the following command:

```
# cpupower frequency-set --governor performance
```

MySQL best practices

Now let's look at MySQL Server settings. This section proposes recommended values for the main MySQL parameters that have a direct impact on performance. You'll see that it's not necessary to change the default values for the majority of the parameters.

Buffer pool size

The `innodb_buffer_pool_size` parameter controls the size in bytes of the InnoDB buffer pool, the memory area where InnoDB caches table and index data. There's no question that for tuning InnoDB, this is the most important parameter. The typical rule of thumb is to set it to around 70% of the total available RAM for a MySQL dedicated server.

However, the larger the server is, the more likely it is that this will end up wasting RAM. For a server with 512 GB of RAM, for example, this would leave 153 GB of RAM for the operating system, which is more than it needs.

So what's a better rule of thumb? Set the `innodb_buffer_pool_size` as large as possible, without causing swapping when the system is running the production workload. This will require some tuning.

In MySQL 5.7 and later this is a dynamic parameter, so you can change it on the fly without the need to restart the database. For example, to set it to 1 GB, use this command:

```
mysql> SET global innodb_buffer_pool_size = 1024*1024*1
```

```
Query OK, 0 rows affected (0.00 sec)
```

To make the change persistent across restarts, you'll need to add this parameter to *my.cnf*, under the `[mysqld]` section:

```
[mysqld]
innodb_buffer_pool_size = 1G
```

The innodb_buffer_pool_instances parameter

One of the more obscure MySQL parameters is `innodb_buffer_pool_instances`. This parameter defines the number of instances that InnoDB will split the buffer pool into. For systems with buffer pools in the multigigabyte range, dividing the buffer pool into separate instances can improve concurrency by reducing contention as different threads read and write to cached pages.

However, in our experience, setting a high value for this parameter may also introduce additional overhead. The reason is that each buffer pool instance manages its own free list, flush list, LRU list, and all other data structures connected to a buffer pool, and is protected by its own buffer pool mutex.

Unless you run benchmarks that prove performance gains, we suggest using the default value (`8`).

NOTE

The `innodb_buffer_pool_instances` parameter was deprecated in MariaDB 10.5.1 and removed in MariaDB 10.6. According to MariaDB architect Marko Makela, this is because the original reasons for splitting the buffer pool have mostly gone away nowadays. You can find more details in the MariaDB Jira ticket.

Redo log size

The redo log is a structure used during crash recovery to correct data written by incomplete transactions. The main goal is to guarantee the durability (D) property of ACID transactions by providing redo recovery for committed transactions. Because the redo file logs all data written to MySQL even before the commit, having the right redo log size is fundamental for MySQL to run smoothly without struggling. An undersized redo log can even lead to errors in operations!

Here's an example of the kind of error you might see if using a small redo log file:

```
[ERROR] InnoDB: The total blob data length (12299456) i
of the total redo log size (100663296). Please increase
```

In this case MySQL was using the default value for the `innodb_log_file_size` parameter, which is 48 MB. To estimate the optimal redo log size, there is a formula that we can use in the majority of cases. Take a look at the following commands:

```
mysql> pager grep sequence
PAGER set to 'grep sequence'
mysql> show engine innodb status\G select sleep(60);
    -> show engine innodb status\G
```

```
Log sequence number 3836410803
1 row in set (0.06 sec)

1 row in set (1 min 0.00 sec)
```

```
Log sequence number 3838334638
1 row in set (0.05 sec)
```

The log sequence number is the total number of bytes written to the redo log. Using the `SLEEP()` command, we can calculate the delta for that period. Then, using the following formula, we can reach an estimated value for the amount of space needed to hold an hour or so of logs (a good rule of thumb):

```
mysql> SELECT (((3838334638 - 3836410803)/1024/1024)*6
    -> AS Estimated_innodb_log_file_size;
```

```
+-------------------------------+
| Estimated_innodb_log_file_size |
+-------------------------------+
|               55.041360855088 |
+-------------------------------+
1 row in set (0.00 sec)
```

We usually round up, so the final number will be 56 MB. This is the value that needs to be added to *my.cnf* under the `[mysqld]` section:

```
[mysqld]
innodb_log_file_size=56M
```

The sync_binlog parameter

The binary log is a set of log files that contain information about data modifications made to a MySQL server instance. They are different from the redo files and have other uses. For example, they are used to create replicas and InnoDB Clusters, and are helpful for performing PITR.

By default, the MySQL server synchronizes its binary log to disk (using `fdatasync()`) before transactions are committed. The advantage is that in the event of a power failure or operating system crash, transactions that are missing from the binary log are only in a prepared state; this allows the automatic recovery routine to roll back the transactions, guaranteeing that no transaction is lost from the binary log. However, the default value (`sync_binlog = 1`) brings a penalty in performance. As this is a dynamic option, you can change it while the server is running with the following command:

```
mysql> SET GLOBAL sync_binlog = 0;
```

For the change to persist after a restart, add the parameter to your *my.cnf* file under the `[mysqld]` section:

```
[mysqld]
sync_binlog=0
```

NOTE

Most of the time, using `sync_binlog=0` will provide good performance (when binary logs are enabled). However, the performance variance can be significant because MySQL will rely on the OS flushing to flush the binary logs. Depending on the workload, using `sync_binlog=1000` or higher will provide better performance than `sync_binlog=1` and less variance than `sync_binlog=0` .

The binlog_expire_logs_seconds and expire_logs_days parameters

To avoid MySQL filling the entire disk with binary logs, you can adjust the settings of the parameters `binlog_expire_logs_seconds` and `expire_logs_days` . `expire_logs_days` specifies the number of days before automatic removal of binary log files. However, this parameter is deprecated in MySQL 8.0, and you should expect it to be removed in a future release.

Consequently, a better option is to use `binlog_expire_logs_seconds` , which sets the binary log expiration period in seconds. The default value for this parameter is `2592000` (30 days). MySQL can automatically remove the binary log files after this expiration period ends, either at startup or the next time the binary log is flushed.

NOTE

If you want to flush the binary log manually, you can execute the following command:

```
mysql> FLUSH BINARY LOGS;
```

The innodb_flush_log_at_trx_commit parameter

`innodb_flush_log_at_trx_commit` controls the balance between strict ACID compliance for commit operations and the higher performance possible when commit-related I/O operations are rearranged and done in batches. It is a delicate option, and many prefer to use the default value (`innodb_flush_log_at_trx_commit=1`) in the source servers, while for replicas they use a value of `0` or `2`. The value `2` instructs InnoDB to write to the log files after each transaction commit, but to flush them to disk only once per second. This means you can lose up to a second of updates if the OS crashes, which, with modern hardware that supports up to one million inserts per second, is not negligible. The value `0` is even worse: logs are written and flushed to disk just once per second, so you may lose up to a second's worth of transactions even if the `mysqld` process crashes.

WARNING

Many operating systems, and some disk hardware, "fool" the flush-to-disk operation. They may tell `mysqld` that the flush has taken place, even though it has not. In this case, the durability of transactions is not guaranteed even with the recommended settings, and in the worst case, a power outage can corrupt InnoDB data. Using a battery-backed disk cache in the SCSI disk controller or in the disk itself speeds up file flushes and makes the operation safer. You can also disable the caching of disk writes in hardware caches if the battery is not working correctly.

The innodb_thread_concurrency parameter

`innodb_thread_concurrency` is set to `0` by default, which means that an infinite number (up to the hardware limit) of threads can be opened and executed inside MySQL. The usual recommendation is to leave this parameter with its default value and only change it to solve contention problems.

If your workload is consistently heavy or has occasional spikes, you can set the value of `innodb_thread_concurrency` using the following formula:

```
innodb_thread_concurrency = Number of Cores * 2
```

Because MySQL does not use multiple cores to execute a single query (it is a 1:1 relation), each core will run one query per single unit of time. Based on our experience, because modern CPUs are fast in general, setting the maximum number of executing threads to double the CPUs available is a good start.

Once the number of executing threads reaches this limit, additional threads sleep for a number of microseconds, set by the configuration parameter `innodb_thread_sleep_delay`, before being placed into the queue.

`innodb_thread_concurrency` is a dynamic variable, and we can change it at runtime:

```
mysql> SET GLOBAL innodb_thread_concurrency = 0;
```

To make the change persistent, you'll also need to add this to *my.cnf*, under the `[mysqld]` section:

```
[mysqld]
innodb-thread-concurrency=0
```

You can validate that MySQL applied the setting with this command:

```
mysql> SHOW GLOBAL VARIABLES LIKE '%innodb_thread_conc
```

NOTE

The MySQL 8.0.14 release notes state: "As of MySQL 8.0.14, InnoDB supports parallel clustered index reads, which can improve `CHECK TABLE` performance." Parallel clustered index reads also work for a simple `COUNT(*)` (without a `WHERE` condition). You can control the number of parallel threads with the `innodb_parallel_read_threads` parameter.

This feature is currently limited and available only for queries without a `WHERE` condition (full scans). However, it is a great start for MySQL and opens the road to real parallel query execution.

NUMA architecture

Non-uniform memory access (NUMA) is a shared memory architecture that describes the placement of main memory modules relative to processors in a multiprocessor system. In the NUMA shared memory architecture, each processor has its own local memory module, leading to a distinct performance advantage because the memory and the processor are physically closer. At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect), as shown in Figure 11-1.

NOTE

A nice alternative that can show memory usage across NUMA nodes is the `numastat` command. You can get a more detailed memory usage per node by executing:

```
# numastat -m
```

Another way to visualize this is by specific process. For example, to check the memory usage in NUMA nodes of the mysqld process:

```
# numastat -p $(pidof mysqld)
```

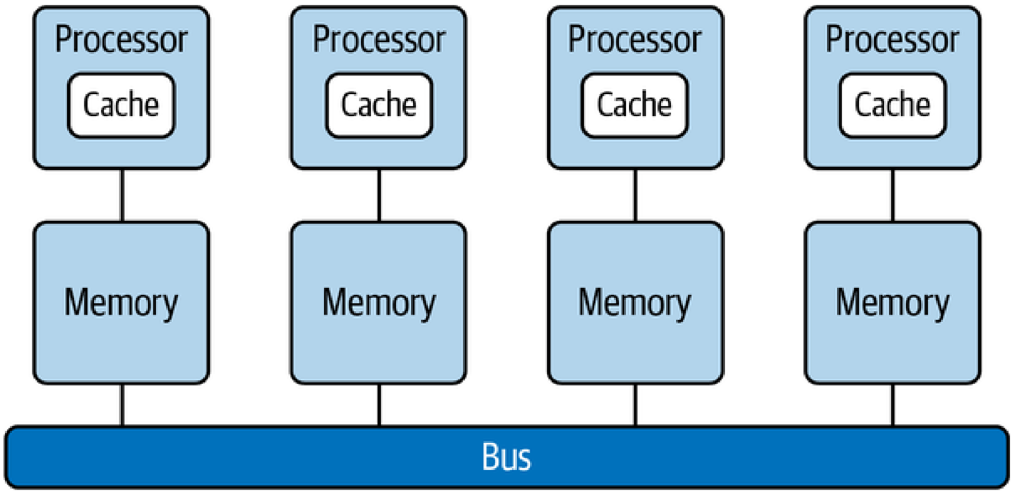

Figure 11-1. NUMA architecture overview

The following command shows an example of the available nodes on a server that has NUMA enabled:

```
shell> numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 24 25 26 27 28 2
node 0 size: 130669 MB
node 0 free: 828 MB
node 1 cpus: 12 13 14 15 16 17 18 19 20 21 22 23 36 37
node 1 size: 131072 MB
node 1 free: 60 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

As we can see, node 0 has more free memory than node 1. There is an issue with this that causes the OS to swap even with memory available, as explained in the excellent article "The MySQL *Swap Insanity* Problem and the Effects of the NUMA Architecture" by Jeremy Cole.

In MySQL 5.7 the `innodb_buffer_pool_populate` and `numa_interleave` parameters were removed, and their functions are now controlled by the `innodb_numa_interleave` parameter. When we enable it, we balance memory allocation across nodes in a NUMA system, avoiding the swap insanity problem.

This parameter is not dynamic, so to enable it we need to add it to the *my.cnf* file, under the `[mysqld]` section, and restart MySQL:

```
[mysqld]
innodb_numa_interleave = 1
```