Chapter 12. Monitoring MySQL Servers

Monitoring can be defined as observing or checking the quality or progress of something over a period of time. Applying that definition to MySQL, what we observe and check are the server's "health" and performance. Quality, then, would be maintaining uptime and having performance meet desired levels. So really, monitoring is a continuous effort to keep things under observation and control. Usually it's thought of as something optional, which may not be needed unless there's a particularly high load or high stakes. However, just like backups, monitoring benefits almost every installation of any database.

We think that having monitoring in place and understanding the metrics you're getting from it is one of the most important tasks for anyone operating a database system—probably just after setting up proper verified backups. Like operating a database without backups, failing to monitor your database is dangerous: what use is a system that provides unpredictable performance and may be "down" randomly? The data may be safe, but it might not be usable.

In this chapter, we'll try to give you a foundation for understanding how to monitor MySQL efficiently. This book is not called *High Performance MySQL*, and we won't be going into depth on the specifics of exactly what different metrics mean or how to perform a complex analysis of a system. But we will talk about few basic metrics that should be checked regularly in every MySQL installation, and we'll discuss important OS-level metrics and tools. We'll then briefly touch on a few widely used methodologies for assessing system performance. After that, we will review a few popular open source monitoring solutions, and finally we'll show you how to gather data for investigation and monitoring purposes manually.

After completing this chapter, you should feel comfortable picking a monitoring tool and be able to understand some of the most important metrics it shows.

Operating System Metrics

An operating system is a complex computer program: an interface between applications, mainly MySQL in our case, and hardware. In the early days, OSs were simple; now they are arguably quite complex, but the idea behind them never really changed. An OS tries to hide, or abstract away, the complexity of dealing with the underlying hardware. It's possible to imagine some special-purpose RDBMS running directly on hardware, being its own operating system, but realistically you'll likely never see that. Apart from providing a convenient and powerful interface, operating systems also expose a lot of performance metrics. You don't need to know each and every one of them, but it's important to have a basic understanding of how to assess the performance of the layer underneath your database.

Usually, when we talk about operating system performance and metrics, what's really being discussed is hardware performance assessed at the operating system level. There's nothing wrong in saying "OS metrics," but remember that at the end of the day they are mostly showing hardware performance.

Let's take a look at the most important OS metrics that you'll want to monitor and in general have a feel for. We will be covering two major OSs in this section: Linux and Windows. Unix-like systems, like macOS and others, will have either the same tools as Linux or at least tools showing the same or similar outputs.

CPU

The *central processing unit* (CPU) is the heart of any computer. Nowadays, CPUs are so complex they can be considered separate computers within computers. Luckily, the basic metrics we think you should understand are universal. In this section, we'll take a look at CPU utilization as reported by Linux and Windows and see what contributes to the overall load.

Before we get into measuring CPU utilization, let's do a quick recap of what a CPU is and what characteristics of it are most important for database operators. We called it the "heart of a computer," but that's oversimplified. In fact, a CPU is a device that can do a few basic (and not so basic) operations, on top of which we build layers and layers of complexity from machine code

up to the high-level programming languages running operating systems and ultimately (for us) database systems.

Every operation a computer does is done by a CPU. As Kevin Closson has said, “Everything is a CPU problem.” When a program is actively being executed—for example, MySQL parsing a query—the CPU is doing all the work. When a program is waiting for a resource—MySQL waiting for a data read from disk—the CPU is involved in “telling” the program when the data is available. Such a list could go on forever.

Here are a few of the most important metrics of a CPU for a server (or any computer in general):

CPU frequency

CPU frequency is the number of times per second a CPU core can “wake up” to execute a piece of work. This is basically the “speed” of the CPU. The more the merrier, but surprisingly often frequency is not the most important metric.

Cache memory

The size of the cache defines the amount of memory located directly within the CPU, making it extremely fast. Again, the more the better, and there are no downsides to having more.

Number of cores

This is the number of execution units within a single CPU “package” (a physical item), and the sum of those units across all CPUs we can fit into a server. Nowadays, it’s increasingly difficult to find a CPU that has a single core: most CPUs are multicore systems. Some even have “virtual” cores, making the difference between the “actual” number of CPUs and the total number of cores even higher.

Usually, having more cores is a good thing, but there are caveats to that. In general, the more cores are available, the more processes can be scheduled by the OS to be executed simultaneously. For MySQL, that means more queries executed in parallel, and less impact from background operations.

But if half of the available cores are "virtual," you don't get the 2x performance increase you might expect. Rather, you *may* get a 2x increase, or you may get anywhere between a 1x and a 2x increase: not every workload (even within MySQL) benefits from virtual cores.

Also, having multiple CPUs in different sockets makes interfacing with memory (RAM) and other onboard devices (like network cards) more complicated. Usually, regular servers are physically laid out in such a way that some CPUs (and their cores) will access parts of RAM quicker than other parts—this is the NUMA architecture we talked about in the previous chapter. For MySQL, this means that memory allocation and memory-related issues can become a pain point. We covered the necessary configuration on NUMA systems in "NUMA architecture".

The basic measurement of CPU is its load in percent. When someone tells you "CPU 20," you can be *pretty* sure that they mean "the CPU is 20% busy at the moment." You can never be totally sure, though, so you'd better double-check. For example, 20% of one core on a multicore system may be just 1% of the overall load. Let's try to visualize this load.

On Linux, the basic command to get the CPU load is `vmstat`. If run without arguments, it will output current average values and then exit. If we run it with a digit argument (we'll call it `X` here), it'll print values every `X` seconds. We recommend that you run `vmstat` with a digit argument—for example, `vmstat 1`—for a few seconds. If you run just `vmstat`, you get averages since boot, which are usually misleading. `vmstat 1` will execute forever until interrupted (pressing Ctrl+C is the easiest way out).

The `vmstat` program prints information not only on CPU load, but also memory and disk-related metrics, as well as advanced system metrics. We will be exploring some other sections of the `vmstat` output soon, but here we'll concentrate on CPU and process metrics.

To start, let's see the `vmstat` output on an idle system. The CPU section is truncated; we'll review it in detail later:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo
```

```
 2  0      0 1229924 1856608 6968268    0    0    39
 1  0      0 1228028 1856608 6969384    0    0     0
 0  0      0 1220972 1856620 6977688    0    0     0
 0  0      0 1217420 1856644 6976796    0    0     0
 0  0      0 1223768 1856648 6968352    0    0     0
```

The first line of the output after the header is an average since boot, and later lines represent current values when printed. The output can be hard to read at first, but you get used to it rather quickly. For clarity, in the rest of this section we will provide a truncated output with only the information we want, in the `procs` and `cpu` sections:

```
procs ------cpu-----
 r  b us sy id wa st
 2  0 18  7 75  0  0
 1  0  2  1 97  0  0
 0  0  3  1 96  0  0
 0  0  2  2 96  0  0
 0  0  2  1 97  0  0
```

`r` and `b` are process metrics: the number of processes actively running and the number of processes blocked (usually waiting for I/O). The other columns represent a breakdown of CPU utilization in percentage points (from 0% to 100%, even on a multicore system). Together, the values in these columns will always add up to 100. Here's what the `cpu` columns indicate:

`us` *(user)*

Time spent running user programs (or, the load put on a system by these programs). MySQL Server is a user program, as is every piece of code that exists outside of the kernel. Importantly, this metric shows time spent purely inside the program itself. For example, when MySQL is doing some computation, or parsing a complex query, this value will go up. When MySQL wants to perform a disk or network operation, this value will also go up, but so will two other values, as you'll soon see.

`sy` *(system)*

Time spent running kernel code. Due to the way Linux and other Unix-like systems are organized, user programs increase this counter. For example, whenever MySQL needs to do a disk read, some work will

have to be done by the OS kernel. Time spent doing that work will be included in the `sy` value.

`id` (idle)

Time spent doing nothing; idle time. On a perfectly idle server, this metric will be 100.

`wa` (I/O wait)

Time spent waiting for I/O. This is an important metric for MySQL, as reading and writing to various files are a relatively large part of MySQL operation. When MySQL does a disk read, some time will be spent in MySQL's internal functions and reflected in `us`. Then some time will be spent inside the kernel and reflected in `sy`. Finally, once the kernel has sent a read request to the underlying storage device (which could be a local or network device) and is waiting for the response and data, all the time spent is accumulated in `wa`. If our program and kernel are very slow and all we do is I/O, in theory this metric could be close to 100. In reality, double-digit values are rare and usually indicate some I/O issues. We'll talk about I/O in depth in "Disk".

`st` (steal)

This is a difficult metric to explain without getting deep into the weeds. It's defined by the MySQL Reference Manual as "time stolen from a virtual machine." You can think of this as the time during which the VM wanted to execute its instructions but had to wait for the host server to allocate CPU time. There are multiple reasons for this behavior, a couple of which are notable. The first is host overprovisioning: running too many large VMs, resulting in a situation where the sum of resources the VMs require is more than the host's capacity. The second is the "noisy neighbor" situation, where one or more VMs suffer from a particularly loaded VM.

Other commands, like `top` (which we'll show in a bit), will have finer CPU load breakdowns. However, the columns just listed are a good starting point and cover most of what you need to know about a running system.

Now let's get back to our `vmstat 1` output on an idle system:

```
procs ------cpu-----
 r  b us sy id wa st
 2  0 18  7 75  0  0
 1  0  2  1 97  0  0
 0  0  3  1 96  0  0
 0  0  2  2 96  0  0
 0  0  2  1 97  0  0
```

What can we tell from this output? As mentioned previously, the first line is an average since boot. On average, there are two processes running (`r`) on this system, with 0 blocked (`b`); user CPU utilization is 18% (`us`), system CPU utilization is 7% (`sy`), and overall the CPU is 75% idle (`id`). I/O wait (`wa`) and steal time (`st`) are 0.

After the first one, each line of the output printed is an average over a sampling interval, which is 1 second in our example. This is pretty close to what we could call “current” values. As this is an idle machine, we see that overall the values are below average. Only one or no processes are running or blocked, user CPU time is 2–3%, system CPU time is 1–2%, and the system is idle 96–97% of the time.

For good measure, let’s look at the `vmstat 1` output on the same system doing a CPU-intensive computation in a single process:

```
procs ------cpu-----
 r  b us sy id wa st
 2  0 18  7 75  0  0
 1  0 13  0 87  0  0
 1  0 13  0 86  0  0
 1  0 14  0 86  0  0
 1  0 15  0 84  0  0
```

The averages since boot are the same, but we have a single process running in every sample, and it drives the user CPU time to 13–15%. The problem with `vmstat` is that we can’t learn from its output which process specifically is burning the CPU. Of course, if this is a dedicated database server, you can suppose that most if not all user CPU time is going to be accounted for by MySQL and its threads, but things happen. The other problem is that on machines with a high CPU core count, you can mistakenly take low readings in `vmstat` output for a fact—but `vmstat` gives a reading from 0% to

100% even on a 256-core machine. If 8 cores of such a machine are 100% loaded, the user time shown by `vmstat` will be 3%, but in reality some workload may be throttled.

Before we talk about a solution to those problems, let's talk Windows a little bit. A lot of what we've said in general about CPU utilization and especially about CPUs will translate to Windows, with some notable differences:

- There's no I/O wait accounting in Windows, as the I/O subsystem is fundamentally different. Time spent by threads waiting for I/O is going into the idle counter.
- The system CPU time counterpart is, roughly, the privileged CPU time.
- Steal information is not available.

The user and idle counters remain unchanged, so you can base your CPU monitoring on user, privileged, and idle CPU time as it is exposed by Windows. There are other counters and metrics available, but this should have you covered quite well. Getting the current CPU utilization on Windows can be done using many different tools. The simplest one, and probably the closest one to `vmstat` in spirit, is the good old Task Manager, a staple of looking at Windows performance metrics. It's readily available, it's simple, and you've probably used it before. The Task Manager can show you CPU utilization in percentage points broken down by CPU cores and also split between user and kernel time.

Figure 12-1 shows the Task Manager running on an idle system.

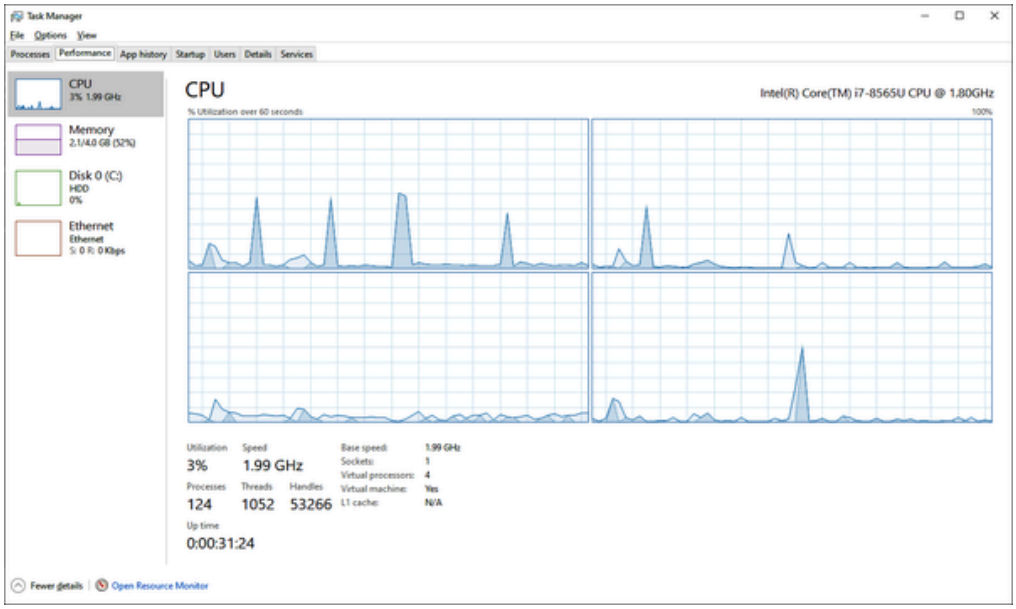

Figure 12-1. Task Manager showing an idle CPU

Figure 12-2 shows the Task Manager running on a busy system.

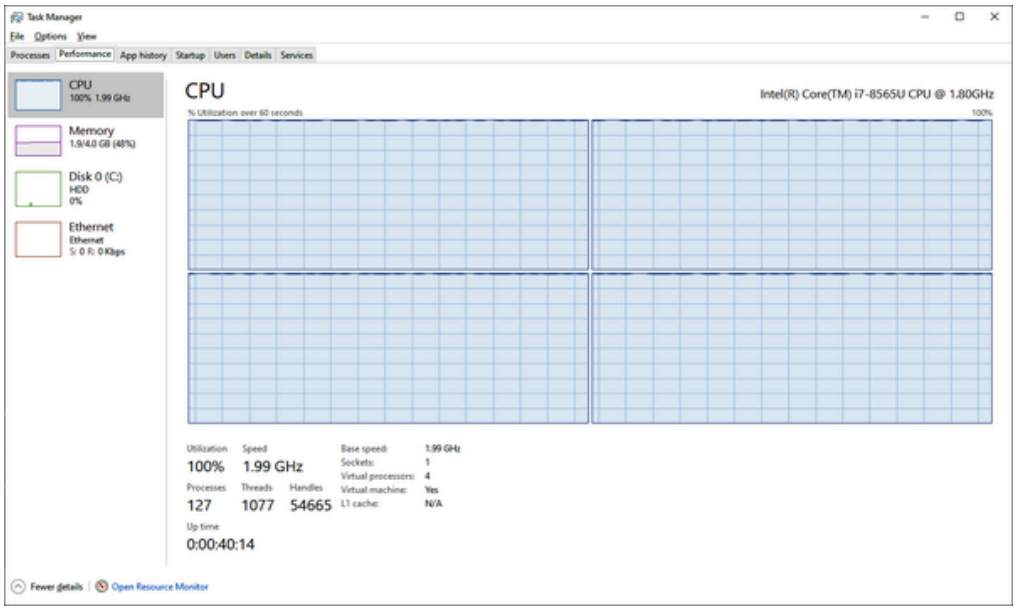

Figure 12-2. Task Manager showing a busy CPU

As we said earlier, `vmstat` has a couple of problems: it doesn't break down load per process or per CPU core. Solving both problems requires running other tools. Why not just run them right away? `vmstat` is universal, gives more than just CPU readings, and is very concise. It's a good way to quickly see if there's anything very wrong with a given system. The same goes for the Task Manager, although it is actually more capable than `vmstat`.

On Linux, the next-simplest tool to use after `vmstat` is `top`, another basic element in the toolbox of anyone dealing with a Linux server. It expands on the basic CPU metrics we have discussed, and adds both per-core load breakdown and per-process load accounting. When you execute `top` without any arguments, it starts in a terminal UI, or TUI, mode. Press `?` to see the help menu. To display the per-core load breakdown, press `1`. Figure 12-3 shows what the output of `top` looks like.

You can see here that each process gets its own overall CPU utilization shown under the `%CPU` column. For example, `mysqld` is using 104.7% of the overall CPU time. Now we can also see how that load is distributed among the many cores the server has. In this particular case, one core (`Cpu0`) is slightly more loaded than the other one. There are cases when MySQL hits a limit of single-CPU throughput, and thus having per-core load breakdown is important. Having a view on how load is distributed between processes is important if you suspect some rogue process is eating into the server's capacity.

```
top - 06:27:27 up 2 min,  2 users,  load average: 0.90, 0.22, 0.07
Tasks: 120 total,   1 running, 119 sleeping,   0 stopped,   0 zombie
%Cpu0  : 27.0 us, 39.8 sy,  0.0 ni, 29.0 id,  1.2 wa,  0.0 hi,  2.9 si,  0.0 st
%Cpu1  : 21.2 us, 32.4 sy,  0.0 ni, 42.0 id,  3.8 wa,  0.0 hi,  0.7 si,  0.0 st
KiB Mem :  1014596 total,    67232 free,   397304 used,   550060 buff/cache
KiB Swap:  1572860 total,  1572596 free,      264 used.   447692 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6186 mysql     20   0  942688 331964  51760 S 104.7 32.7   0:21.13 mysqld
   30 root      20   0       0      0      0 S   7.6  0.0   0:01.50 kworker/0:1
  334 root       0 -20       0      0      0 S   5.0  0.0   0:00.98 kworker/0:1H
 6232 root      20   0  275916   3468   2556 S   4.7  0.3   0:00.97 mysqlslap
    6 root      20   0       0      0      0 S   2.7  0.0   0:00.50 ksoftirqd/0
   38 root      20   0       0      0      0 S   0.3  0.0   0:00.02 kswapd0
  233 root      20   0       0      0      0 S   0.3  0.0   0:00.07 kworker/1:2
    1 root      20   0  128016   5212   2684 S   0.0  0.5   0:00.95 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0
```

Figure 12-3. `top` in TUI mode

There are many more tools that can show you even more data. We can't talkin detail about all of them, but we'll name a few. `mpstat` can give very in-depth CPU statistics. `pidstat` is a universal tool that provides stats on CPU, memory, disk, and network utilization for each individual process running. `atop` is an advanced version of `top`. There are more, and everyone has their favorite set of tools. We firmly believe that what really matters is not the tools themselves, though they help, but an understanding of the core metrics and stats that they provide.

On Windows, the Task Manager program is actually much closer to `top` than it is to `vmstat`, although we've done just that comparison. The Task Manager's ability to show per-core load and per-process load makes it quite a useful first step in any investigation. We recommend diving into the Resource Monitor right away, as it provides more details. The easiest way to access it is to click the Open Resource Monitor link in the Task Manager.

Figure 12-4 shows a Resource Monitor window with CPU load details.

The Task Manager and Resource Monitor are not the only tools on Windows capable of showing performance metrics. Here are a couple other tools that you may want to get comfortable using. They are more advanced, so we won't go into detail here:

Performance Monitor

This built-in tool is a GUI for the performance counter subsystem in Windows. In short, you can view and plot any (or all) of the various performance metrics Windows measures, not only those related to the CPU.

Process Explorer

This tool is a part of a suite of advanced system utilities called Windows Sysinternals. It's more powerful and more advanced than the other tools listed here and can be useful to learn. Unlike the other tools, you'll have to install Process Explorer separately from its home page on the Sysinternals site.

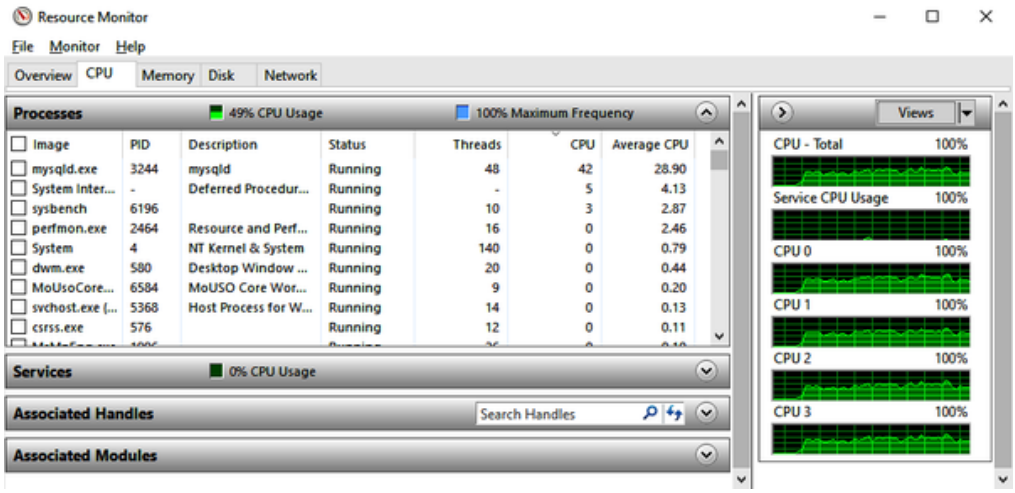

Figure 12-4. Resource Monitor showing CPU load details

Disk

The disk or I/O subsystem is crucial for database performance. Although the CPU underpins every operation done on any given system, for databases in particular, the disk is likely to be the most problematic bottleneck. That's only logical—after all, databases store data on disk and then serve that data from disk. There are many layers of caches on top of slow and durable long-term storage, but they cannot always be utilized and are not infinitely large. Thus, understanding basic disk performance is extremely important when dealing with database systems. The other important and frequently underestimated property of any storage system is not related to performance at all—it's the capacity. We'll start with that.

Disk capacity and utilization refer to the total amount of data that can be stored on a given disk (or a lot of disks together in a storage system) and how much of that data is already stored. These are boring but important metrics. While it's not really necessary to monitor the disk capacity, as it's unlikely to change without you noticing, you absolutely must keep an eye on the disk utilization and available space.

Most databases only grow in size over time. MySQL in particular requires a healthy amount of available disk space headroom to accommodate table changes, long-running transactions, and spikes in write load. When there's no

more disk space available for use by a MySQL database instance, it may crash or stop working and is unlikely to start working again until some space is freed up or more disk capacity is added. Depending on your circumstances, adding more capacity may take from minutes to days. That's something you probably want to plan for ahead of time.

Luckily, monitoring disk space usage is very easy. On Linux, it can be done using the simple `df` command. Without arguments, it will show capacity and usage in 1 KB blocks for every filesystem. You can add the `-h` argument to get human-readable measurements, and specify a mountpoint (or just a path) to limit the check. Here's an example:

```
$ df -h /var/lib/mysql
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        40G   18G   23G  45% /
```

`df` 's output is self-explanatory, and it's one of the easiest tools to work with. We recommend that you try to keep your database mountpoints at 90% capacity unless you run multiterabyte systems. In that case, go higher. A trick that you can use is to put some large dummy files on the same filesystem as your database. Should you start running out of space, you can remove one or more of these files to give yourself some more time to react. Instead of relying on this trick, though, we recommend that you have some disk space monitoring in place.

On Windows, the trusty File Explorer can provide disk space utilization and capacity information, as shown in Figure 12-5.

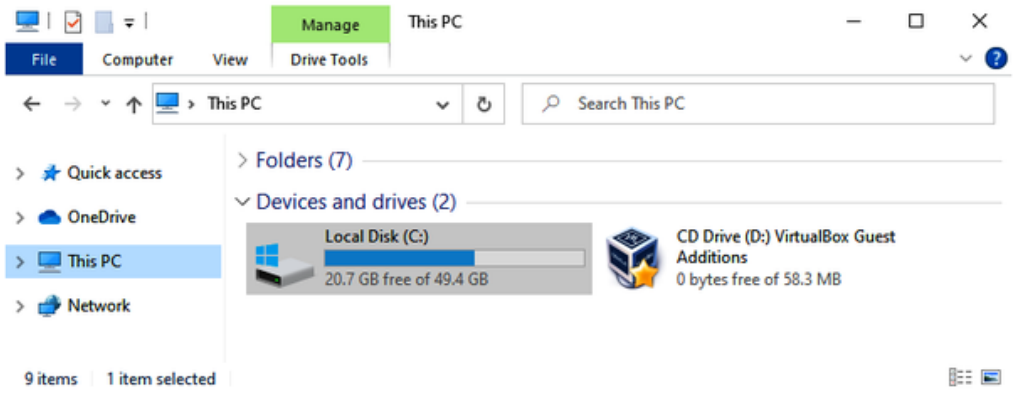

Figure 12-5. File Explorer showing available disk space

With disk space covered, we will now explore the key performance properties of any I/O subsystem:

Bandwidth

How many bytes of data can be pushed to (or pulled from) storage per unit of time

I/O operations per second (IOPS)

The number of operations a disk (or other storage system) is capable of serving per unit of time

Latency

How long it takes for a read or a write to be served by the storage system

These three properties are enough to describe any storage system and start forming an understanding of whether it's good, bad, or ugly. As we did in the CPU section, we'll show you a couple of tools to inspect disk performance and use their output to explain the specific metrics. We're again focusing on Linux and Windows; other systems will have something similar, so the knowledge is portable.

The I/O load analog to `vmstat` on Linux is the `iostat` program. The pattern of interaction should be familiar: invoke the command without arguments, and you get average values since boot; pass a number as an argument, and you get averages for the sampling period. We also prefer running the tool with the `-x` argument, which adds a lot of useful details. Unlike `vmstat`, `iostat` gives metrics broken down by a block device, similar to the `mpstat` command we mentioned earlier.

TIP

`iostat` is usually installed as part of the `sysstat` package. Use `apt` on Ubuntu/Debian or `yum` on RHEL-based operating systems to install the package. You should be comfortable using these tools after following the instructions in Chapter 1.

Let's take a look at an example output. We'll use a `iostat -dxyt 5` command, which translates to: print the device utilization report, display extended statistics, omit the first report with averages since boot, add the timestamp for each report, and report the average values over every 5-second period some sample output on a loaded system:

```
05/09/2021 04:45:09 PM
Device:         rrqm/s   wrqm/s   r/s       w/s    rkB/s
sda               0.00     0.00  0.00   1599.00     0.00
...avgrq-sz avgqu-sz   await r_await  w_await  svctm
...  256.00   141.67   88.63    0.00    88.63   0.63  1
```

There's quite a lot to unpack here. We won't cover every column, but we'll highlight the ones that correspond to properties we mentioned before:

Bandwidth

In `iostat` output, the columns `rkB/s` and `wkB/s` correspond to bandwidth utilization (read and write, respectively). If you know the characteristics of the underlying storage (for example, you may know it promises 200 MiB/s of combined read and write bandwidth), you can tell if you're pushing the limits. Here you can see a respectable figure of just over 200,000 KB per second being written to the `/dev/sda` device and no reads taking place.

IOPS

This metric is represented by the `r/s` and `w/s` columns, giving the number of read and write operations per second, respectively. Our example shows 1,599 write operation per second happening. As expected, no read operations are registered.

Latency

Shown in a slightly more complex manner, latency is broken down into four columns (or more, in newer `iostat` versions): `await`, `r_await`, `w_await`, and `svctm`. For a basic analysis you should be looking at the `await` value, which is the average latency for serving any request. `r_await` and `w_await` break `await` down by reads and writes. `svctime` is a deprecated metric, which attempts to show the pure device latency without any queueing.

Having these basic metric readings and knowing some basic facts about the storage used, it is possible to tell what's going on. Our example is running on a modern consumer-grade NVMe SSD in one of the author's laptops. While the bandwidth is pretty good, each request averages 88 ms, which is a lot. You can also do some simple math to get an I/O load pattern from these metrics.

For example, if we divide bandwidth by IOPS, we get a figure of 128 KB per request. `iostat` does, actually, include that metric in the `avgrq-sz` column, which shows the average request size in a historical unit of *sectors* (512 bytes). You can go forward and measure that 1,599 writes per second can be served only at ~40 ms/request, meaning that there's parallel write load (and also that our device is capable of serving parallel requests).

I/O patterns—size of requests, degree of parallelism, random versus sequential—can shift the upper limits of the underlying storage. Most devices will advertise maximum bandwidth, maximum IOPS, and minimal latency at specific conditions, but these conditions may vary for maximum IOPS and maximum bandwidth measurements, as well as for optimal latency. It is rather difficult to definitely answer the question of whether metric readings are good or bad. Without knowing anything about the underlying storage, one way to look at utilization is to try to assess saturation. Saturation, which we'll touch on in "The USE Method", is a measure of how overloaded a resource is. This becomes increasingly complicated with modern storage, capable of servicing long queues efficiently in parallel, but in general, queueing on a storage device is a sign of saturation. In the `iostat` output, that is the `avgqu-sz` column (or `aqu-sz` in newer versions of `iostat`), and values larger than 1 usually mean that a device is saturated. Our example shows a queue of 146 requests, which is a lot, likely telling us that I/O is highly utilized and may be a bottleneck.

Unfortunately, as you might've noticed, there's no simple straight measure of I/O utilization: there seems to be a caveat for every metric. Measuring storage performance is a difficult task!

The same metrics define storage devices on Linux, Windows, and any other OS.

Let's now take a look at basic Windows tools for assessing I/O performance. Their readings should be familiar by now. We recommend using the Resource Monitor, which we showed in the CPU section, but this time navigate to the Disk tab. Figure 12-6 shows that view with MySQL under heavy write load.

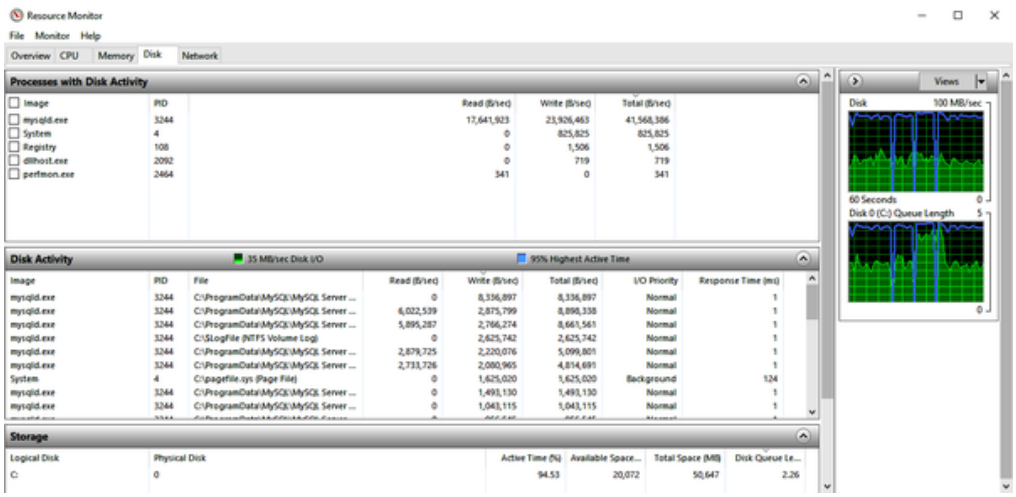

Figure 12-6. Resource Monitor showing IO load details

The metrics presented by the Resource Monitor are similar to those of `iostat`. You can see bandwidth, latency, and length of the request queue. One metric that's missing is IOPS. To get that information, you'll need to use the Performance Monitor (`perfmon`), but we'll leave that as an exercise.

The Resource Monitor actually shows a slightly more detailed view than `iostat`. There's a breakdown of I/O load per process, and a further breakdown of that load per file. We don't know of a single tool on Linux that is capable of showing load broken down like that simultaneously. To get the load breakdown per program, you can use the `pidstat` tool on Linux, which we mentioned before. Here's some example output:

```
# pidstat -d 5
...
10:50:01 AM   UID       PID   kB_rd/s   kB_wr/s kB_ccwr
10:50:06 AM    27      4725      0.00  30235.06      0.

10:50:06 AM   UID       PID   kB_rd/s   kB_wr/s kB_ccwr
10:50:11 AM    27      4725      0.00  23379.20      0.
...
```

Getting a breakdown per file on Linux is quite easily achieved using the BCC Toolkit, specifically the `filetop` tool. There are many more tools to explore in the toolkit, but most are quite advanced. The tools we've shown here should be enough to cover basic investigation and monitoring needs.

Memory

Memory, or RAM, is another important resource for any database. Memory offers vastly superior performance to disk for reading and writing data, and thus databases strive to operate "in memory" as much as possible. Unfortunately, memory is not persistent, so eventually every operation must be reflected on the disk. (For more on disk performance, refer to the previous section.)

In contrast to the CPU and disk sections, we won't actually be talking about memory performance. Even though it's important, it's also a very advanced and deep topic. Instead, we will be focusing on memory utilization. That can also get quite complex quite quickly, so we'll try to stay focused.

Let's start with some basic premises. Every program needs some memory to operate. Database systems, including MySQL, usually need a *lot* of memory. When you run out of it, applications start having performance issues and may even fail, as you'll see at the end of the section. Monitoring memory utilization therefore is crucial to any system's stability, but especially to a database system's.

In this case, we'll actually start with Windows, since on the surface it has slightly less convoluted memory accounting mechanism than Linux. To get the overall OS memory utilization on Windows, all you need to do is start the Task Manager, as described in "CPU", navigate to the Performance tab, and pick Memory. You can see the Task Manager's memory usage display in Figure 12-7.

This machine has 4 GB of memory in total, with 2.4 GB being currently used and 1.6 GB available, making overall utilization 60%. This is a safe amount, and we may even want to allocate more memory to MySQL to minimize "wasted" free memory. Some ideas on MySQL's InnoDB buffer pool sizing can be found in "Buffer pool size".

On Linux, the simplest tool to get memory utilization details is the `free` command. We recommend using it with the `-h` argument, which converts all fields to a human-readable format. Here's some sample output on a machine running CentOS 7:

```
$ free -h
               total        used        free      sharec
Mem:           3.7G        2.7G        155M        8.5M
Swap:          2.0G         13M        2.0G
```

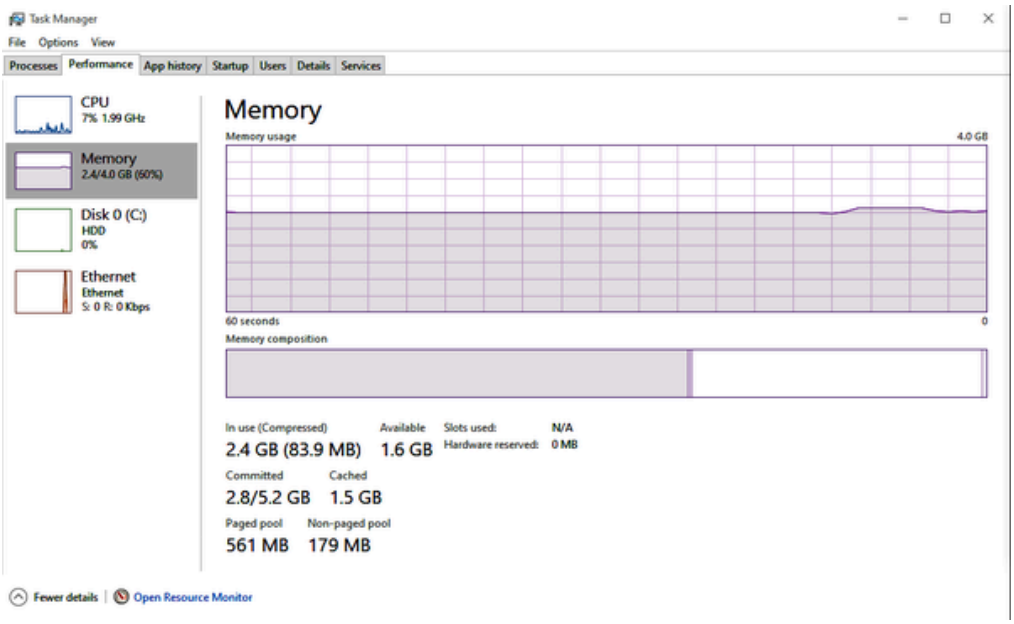

Figure 12-7. Task Manager showing memory utilization details

Now, that's more data than we saw on Windows. In reality, Windows has most of these counters; they are just not as visible.

Let's go through the output. For now, we'll be covering the Mem row, and we'll talk about Swap later.

The two key metrics here are `used` and `available`, which translate to the Task Manager's *In use* and *Available*. A frequent mistake, and one your authors used to make, is to look at the `free` metric instead of `available`. That's not correct! Linux (and, in fact, Windows) doesn't like to keep memory free. After all, free memory is a wasted resource. When there's memory available that's not needed by applications directly, Linux will use that memory to keep a cache of data being read and written from and to the disk. We'll show later that Windows does the same, but you cannot see that from the Task Manager. For more on why it's a mistake to focus on the `free` metric, see the site "Linux ate my ram".

Let's further break down the output of this command. The columns are:

`total`

Total amount of memory available on the machine

used

Amount of memory currently used by applications

free

Actual free memory not used by the OS at all

shared

A special type of memory that needs to be specifically requested and allocated and that multiple processes can access together; because it's not used by MySQL, we're skipping the details here

buff/cache

Amount of memory the OS is currently using as a cache to improve I/O

available

Amount of memory that applications could use if they needed it; usually the sum of `free` and `buff/cache`

In general, for basic but robust monitoring, you only need to look at the `total`, `used`, and `available` amounts. Linux should be capable of handling the cached memory on its own. We're deliberately not covering the page cache here, as that's an advanced topic. By default, MySQL on Linux will utilize the page cache, so you should size your instance to accommodate for that. An often recommended change, however, is to tell MySQL to avoid the page cache (look for the documentation on `innodb_flush_method`), which will allow more memory to be used by MySQL itself.

We've mentioned that Windows has mostly the same metrics; they're just hidden. To see that, open the Resource Monitor and navigate to the Memory tab. Figure 12-8 shows the contents of this tab.

You'll immediately notice that the amount of free memory is just 52 MB, and there's a hefty chunk of standby memory, with a little bit of modified memory. The Cached value in the list below is the sum of the modified and standby amounts. When the screenshot was taken, 1,593 MB of memory was being used by cache, with 33 MB of that being dirty (or modified). Windows, like Linux, caches filesystem pages in an attempt to minimize and smooth I/O and utilize the memory to its fullest capacity.

Another thing you can see is a breakdown of memory utilization per process, with `mysqld.exe` holding just under 500 MB of memory. On Linux, a similar output can be obtained with the `top` command, which we first used in "CPU". Once `top` is running, press Shift+M to sort the output by memory usage and get human-readable figures. `top` output showing memory usage details is in Figure 12-9.

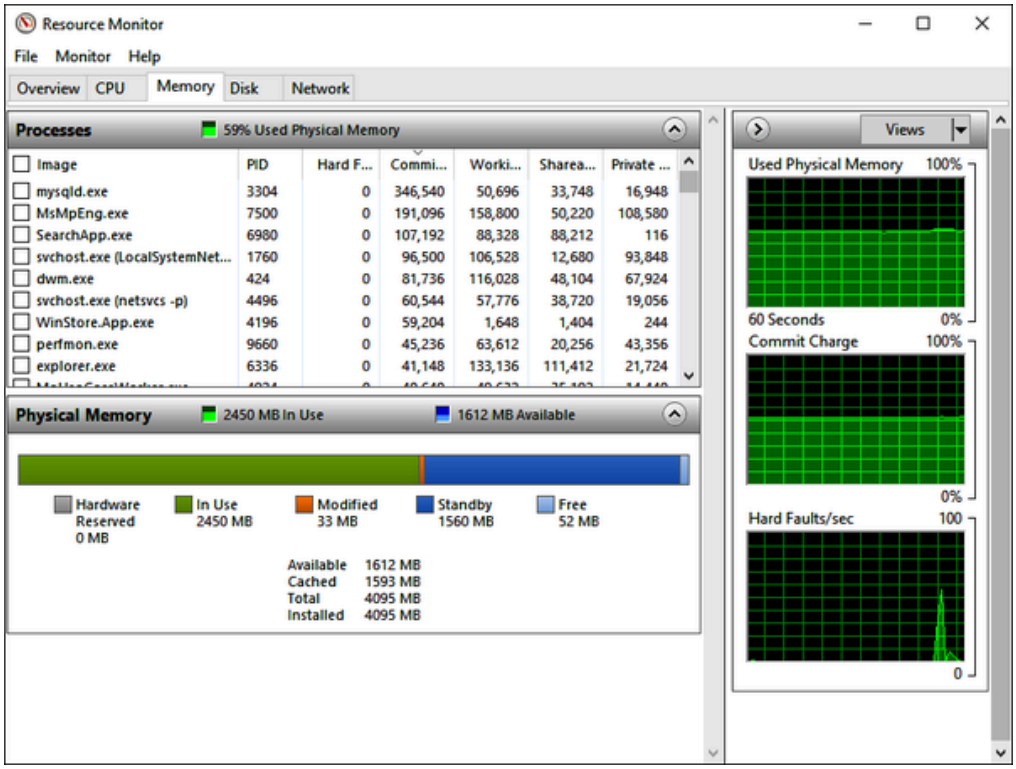

Figure 12-8. Resource Monitor showing memory utilization details

```
top - 19:59:54 up  1:37,  2 users,  load average: 0.02, 0.02, 0.05
Tasks: 113 total,   1 running, 112 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  3880160 total,   144280 free,  2805208 used,   930672 buff/cache
KiB Swap:  2097148 total,  2083060 free,    14088 used.   794528 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5576 mysql     20   0 4330756   2.5g  16468 S   0.3 66.7   0:27.77 mysqld
 5309 root      20   0  741500  80444  29052 S   0.0  2.1   0:02.14 dockerd
 5296 root      20   0  536684  45276  15272 S   0.0  1.2   0:01.60 containerd
 2722 root      20   0  574032  17244   6104 S   0.0  0.4   0:01.38 tuned
 2340 polkitd   20   0  613016  13300   5072 S   0.0  0.3   0:00.21 polkitd
 3832 root      20   0  625648   9000   6696 S   0.0  0.2   0:00.36 NetworkManager
    1 root      20   0  193696   6744   4192 S   0.0  0.2   0:02.19 systemd
 2724 root      20   0  216416   5772   2740 S   0.0  0.1   0:00.70 rsyslogd
 4552 root      20   0  154664   5528   4212 S   0.0  0.1   0:00.35 sshd
 6294 root      20   0  154664   5524   4212 S   0.0  0.1   0:00.17 sshd
 3857 root      20   0  107472   5436   3444 S   0.0  0.1   0:00.01 dhclient
 1215 root      20   0   47580   5056   2884 S   0.0  0.1   0:00.24 systemd-udevd
 6250 root      20   0  241280   4604   3440 S   0.0  0.1   0:00.03 sudo
 2721 root      20   0  112872   4340   3312 S   0.0  0.1   0:00.02 sshd
 2954 postfix   20   0   89832   4072   3068 S   0.0  0.1   0:00.02 qmgr
```

Figure 12-9. `top` showing memory utilization details

On this system the output is not very interesting, but you can quickly see that it's MySQL that consumes the most memory with its `mysqld` process.

Before finishing this section, we want to talk about what happens when you run out of memory. Before that, though, let's discuss swapping, or paging. We should mention here that most modern OSs implement memory management

in such a way that individual applications each have their own view of the system's memory (hence you may see application memory being called *virtual* memory) and that the sum total of the virtual memory that applications can use exceeds the total actual memory capacity of the system. Discussion of the former point is better suited for a university course on operating system design, but the latter point is very important when running database systems.

The implications of this design are important, because an OS can't just magically extend the capacity of the system's memory. In fact, what happens is the OS uses disk storage to extend the amount of memory, and as we've mentioned, RAM is usually far more performant than even the fastest disk. Thus, as you can imagine, there's a price to pay for this memory extension. *Paging* can occur in a few different ways and for different reasons. Most important for MySQL is the type of paging called *swapping*—writing out parts of memory into a dedicated place on disk. On Linux, that place can be a separate partition, or a file. On Windows, there's a special file called *pagefile.sys* that has mostly the same use.

Swapping is not bad per se, but it's problematic for MySQL. The problem is that our database thinks it's reading something from memory, whereas in reality the OS has paged out some of that data into the swap file and will actually read it from disk. MySQL cannot predict when this situation will happen and can do nothing to prevent it or optimize the access. For the end user, this can mean a sudden unexplained drop in query response times. Having *some* swap space, though, is an important protective measure, as we'll show.

Let's move on to answer the question of what really happens when you run out of memory. In short: nothing good. There are only a couple of general outcomes for MySQL when the system is running out of memory, so let's talk through them:

- MySQL requests more memory from the OS, but there's none available—everything that could be paged out is not in memory, and the swap file is absent or already full. Usually, this situation results in a crash. That's a really bad outcome.
- On Linux, a variation of the preceding point is that the OS detects a situation where the system is close to running out of memory and forcefully terminates—in other words, kills one or more processes. Usually, the processes terminated will be the ones holding the most

memory, and usually on a database server, MySQL will be the top memory consumer. This usually happens before the situation explained in the previous point.

- MySQL, or some other program, fills up memory to a point where the OS has to start swapping. This assumes the swap space (or pagefile in Windows) is set up. As explained a few paragraphs back, MySQL's performance will degrade unexpectedly and unpredictably when its memory is swapped out. This arguably is a better outcome than just a crash or MySQL being terminated, but nevertheless it's something to avoid.

So, MySQL will either get slower, crash, or get killed, as simple as that. You now should see clearly why monitoring available and used memory is *very* important. We also recommend leaving some memory headroom on your servers and having a swap/pagefile set up. For some advice on Linux swap setup, see "Operating system best practices".

Network

Of all the OS resources, the network is probably the one most frequently blamed for random unexplained issues. There's a good reason for that: monitoring the network is difficult. Understanding issues with the network sometimes requires a detailed analysis of the whole network stream. It's a peculiar resource because, unlike CPU, disk, and memory, it is not contained within a single server. At the very least, you need two machines communicating with each other for "network" to even be a factor. Of course, there are local connections, but they are usually stable. And granted, disk storage may be shared, and CPU and memory in case of virtual machines can be shared, too, but networking is *always* about multiple machines.

Since this chapter is about monitoring, we're not going to cover connectivity issues here—yet a surprising number of issues with networking boil down to the simple problem of one computer not being able to talk to another. Do not take connectivity for granted. Network topologies are usually complex, with each packet following a complicated route through multiple machines. In cloud environments, the routes can be even more complex and less obvious. If you think you have some network issues, it's wise to check that connections can be established at all.

We'll be touching on the following properties of any network:

Bandwidth and its utilization (throughput)

This is similar to the same concept defined in "Disk". Every network connection has a maximum bandwidth capacity, usually expressed as some unit of volume of data per second. Internet connections usually use Mbps or megabits per second, but MBps, or megabytes per second, can also be used. Network links and equipment put a hard cap on maximum bandwidth. For example, currently, common household network equipment rarely exceeds 1 Gbps bandwidth. More advanced data center equipment regularly supports 10 Gbps. Special equipment exists that can drive bandwidth to hundreds of Gbps, but such connections are usually unrouted direct connections between two servers.

Errors—their number and sources

Network errors are unavoidable. In fact, the Transmission Control Protocol (TCP), a backbone of the internet and a protocol used by MySQL, is built around the premise that packets will be lost. You'll undoubtedly see errors from time to time, but having a high rate of errors will cause connections to be slow, as communicating parties will need to resend packets over and over.

Continuing the analogy with the disk, we could also include latency and number of packets sent and received (loosely resembling IOPS). However, packet transmission latency can be measured only by the application that's doing the actual transmission. The OS can't measure and show some average latency for a network. And the number of packets is usually redundant, as it follows the bandwidth and throughput figures.

One particular metric that is useful to add when looking at networks is the number of *retransmitted* packets. Retransmission happens when a packet is lost or damaged. It is not an error, but is usually a result of some issues with the connection. Just like running out of bandwidth, an increased number of retransmissions will lead to choppy network performance.

On Linux, we can start by looking at the network interface statistics. The easiest way to do this is to run the `ifconfig` command. Its output by default will include every network interface on a particular host. Since we know in this case all load comes through `eth1`, we can show only stats for that:

```
$ ifconfig eth1
...
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1
        inet 192.168.10.11  netmask 255.255.255.0  broa
        inet6 fe80::a00:27ff:fef6:b4f  prefixlen 64  sc
        ether 08:00:27:f6:0b:4f  txqueuelen 1000  (Ethe
        RX packets 6217203  bytes 735108061 (701.0 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11381894  bytes 18025086781 (16.7 Gi
        TX errors 0  dropped 0 overruns 0  carrier 0  c
...
```

We can immediately see that the network is pretty healthy just by the fact that there are no errors receiving (RX) or sending (TX) packets. The RX and TX total data stats (701.0 MiB and 16.7 GiB, respectively) will grow each time you run `ifconfig`, so you can easily measure bandwidth utilization by running it over time. That's not terribly convenient, and there are programs that show transmission rates in real time, but none of those ships by default in common Linux distributions. To see a history of the transmission rate and errors, you can use the `sar -n DEV` or `sar -n EDEV` command, respectively (`sar` is a part of the `sysstat` package we mentioned when talking about `iostat`):

```
$ sar -n DEV
                IFACE   rxpck/s   txpck/s    rxkB/s
06:30:01 PM      eth0      0.16      0.08      0.01
06:30:01 PM      eth1   7269.55  13473.28    843.84  21
06:30:01 PM        lo      0.00      0.00      0.00
06:40:01 PM      eth0      0.48      0.28      0.03
06:40:01 PM      eth1   7844.90  13941.09    893.95  19
06:40:01 PM        lo      0.00      0.00      0.00
...rxcmp/s   txcmp/s  rxmcst/s
...   0.00      0.00      0.00
...   0.00      0.00      0.00
...   0.00      0.00      0.00
...   0.00      0.00      0.00
...   0.00      0.00      0.00
...   0.00      0.00      0.00
```

```
$ sar -n EDEV
04:30:01 PM     IFACE   rxerr/s   txerr/s    coll/s  rx
06:40:01 PM      eth0      0.00      0.00      0.00
06:40:01 PM      eth1      0.00      0.00      0.00
06:40:01 PM        lo      0.00      0.00      0.00
...txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
...    0.00      0.00      0.00      0.00      0.00
...    0.00      0.00      0.00      0.00      0.00
...    0.00      0.00      0.00      0.00      0.00
```

Again, we see that in our example interface `eth1` is quite loaded, but there are no errors being reported. If we stay within bandwidth limits, network performance should be normal.

To get a full detailed view of the various errors and issues that have happened within the network, you can use the `netstat` command. With the `-s` flag, it will report a lot of counters. To keep things basic, we will show just the `Tcp` section of the output, with a number of retransmits. For a more detailed overview, check the `TcpExt` section of the output:

```
$ netstat -s
...
Tcp:
    55 active connections openings
    39 passive connection openings
    0 failed connection attempts
    3 connection resets received
    9 connections established
    14449654 segments received
    25994151 segments send out
    54 segments retransmitted
    0 bad segments received.
    19 resets sent
...
```

Considering the sheer number of segments sent out, the retransmission rate is excellent. This network seems to be fine.

On Windows, we again resort to checking the Resource Monitor, which provides most of the metrics we want, and more. Figure 12-10 shows

network-related views the Resource Monitor has to offer on a host running a synthetic load against MySQL.

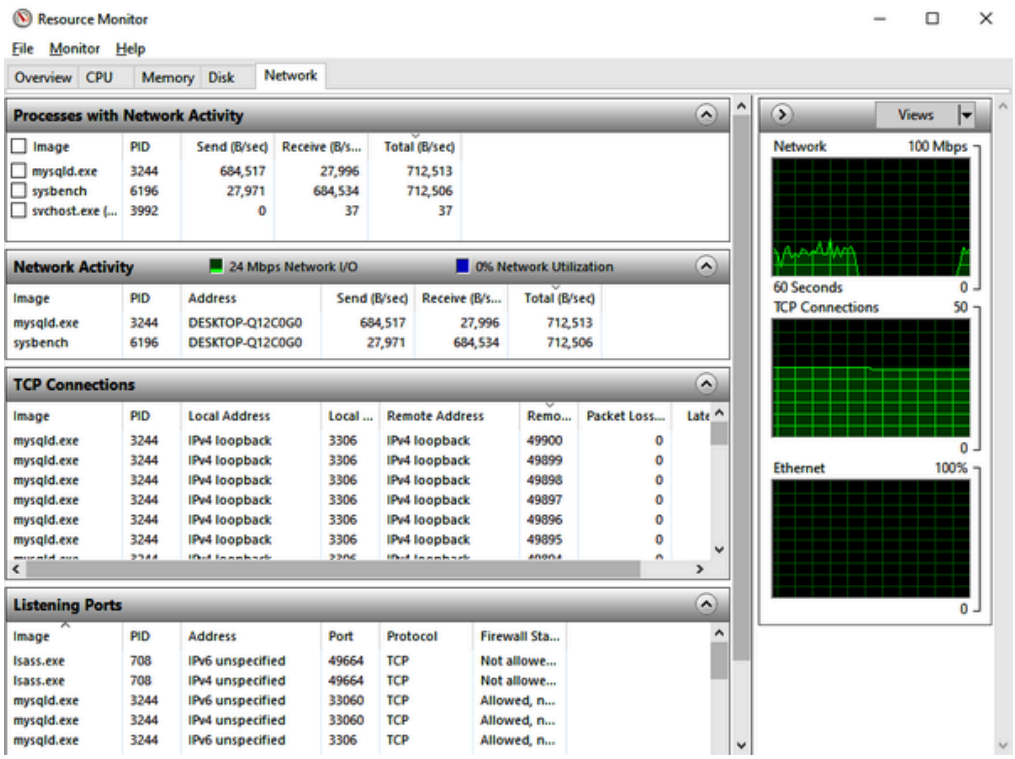

Figure 12-10. Resource Monitor showing network utilization details

To get a reading on the number of errors on Windows, you can use the `netstat` command. Note that even though it has the same name as the Linux tool we used previously, they are slightly different. In this case, we have no errors:

```
C:\Users\someuser> netstat -e
Interface Statistics

                           Received            Sent

Bytes                    58544920         7904968
Unicast packets             62504           32308
Non-unicast packets             0             364
Discards                        0               0
Errors                          0               0
Unknown protocols               0
```

The `-s` modifier for `netstat` exists on Windows, too. Again, we're only showing a part of the output here:

```
C:\Users\someuser> netstat -s
...
```

```
TCP Statistics for IPv4

  Active Opens                        = 457
  Passive Opens                       = 30
  Failed Connection Attempts          = 3
  Reset Connections                   = 121
  Current Connections                 = 11
  Segments Received                   = 61237201
  Segments Sent                       = 30866526
  Segments Retransmitted              = 0
...
```

Judging by the metrics we highlighted for monitoring—bandwidth utilization and errors—this system's network is operating perfectly fine. We understand this is barely scratching the surface when it comes to the complexity of networking. However, this minimal set of tools can help you out immensely in understanding whether you should be blaming your network at all.

That finishes a pretty lengthy overview of OS monitoring basics. We could probably have kept it shorter, and you may ask why we put all this in a book about MySQL. The answer is pretty simple: because it is important. Any program interacts with an OS and requires some of the system's resources. MySQL, by nature, is usually going to be a very demanding program, which you expect to be performing well. For that, however, you need to make sure that you have the necessary resources and you're not running out of a performance capacity for disk, CPU, or network, or just out of capacity for disk and memory. Sometimes, an issue with system resources caused by MySQL can also lead you to uncover issues within MySQL itself. For example, a badly written query may put a lot of load on the CPU and disk while causing a spike in memory usage. The next section shows some basic ways to monitor and diagnose a running MySQL server.

MySQL Server Observability

Monitoring MySQL is simultaneously easy and difficult. It's easy, because MySQL exposes almost 500 status variables, which allow you to see almost exactly what is going on inside your database. In addition to that, InnoDB has its own diagnostic output. Monitoring is hard, though, because it may be tricky to make sense of the data you have.

In this section, we're going to explain the basics of MySQL monitoring, starting with going over what the status variables are and how to get them, and moving on to InnoDB's diagnostics. Once that's covered, we'll show a few basic recipes we believe should be a part of every MySQL database monitoring suite. With these recipes and what you learned about OS monitoring in the previous section, you should be able to understand what's going on with your system.

Status Variables

We'll start with MySQL's *server status variables*. These variables, unlike configuration options, are read-only, and they show you information about the current state of the MySQL server. They vary in nature: most of them are either ever-increasing counters, or gauges with values moving up and down. Some, though, are static text fields, which are helpful to understand the current server configuration. All status variables can be accessed at the global server level and at the current session level. But not every variable makes sense on a session level, and some will show the same values on both levels.

`SHOW STATUS` is used to get the current status variable values. It has two optional modifiers, `GLOBAL` and `SESSION`, and defaults to `SESSION`. You can also specify the name of a variable, or a pattern, but that's not mandatory. The command in the following example shows all the status variable values for the current session:

```
mysql> SHOW STATUS;
```

```
+-----------------------------------------------+------
| Variable_name                                 | Value
+-----------------------------------------------+------
| Aborted_clients                               | 0
| Aborted_connects                              | 0
| Acl_cache_items_count                         | 0
| ...
| Threads_connected                             | 2
| Threads_created                               | 2
| Threads_running                               | 2
| Uptime                                        | 30566
| Uptime_since_flush_status                     | 30566
| validate_password.dictionary_file_last_parsed | 2021-
| validate_password.dictionary_file_words_count | 0
```

```
+-------------------------------------------------+------
482 rows in set (0.01 sec)
```

Scrolling through hundreds of rows of output is suboptimal, so let's instead use a wildcard to limit the number of variables we request. `LIKE` in `SHOW STATUS` works the same as it does for regular `SELECT` statements, as explained in Chapter 3:

```
mysql> SHOW STATUS LIKE 'Created%';

+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 7     |
| Created_tmp_tables      | 0     |
+-------------------------+-------+
3 rows in set (0.01 sec)
```

Now the output is much easier to read. To read the value of a single variable, just specify its full name within quotes without the wildcard, like so:

```
mysql> SHOW STATUS LIKE 'Com_show_status';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Com_show_status | 11    |
+-----------------+-------+
1 row in set (0.00 sec)
```

You might notice in the output for the `Created%` status variables that MySQL showed a value of `7` for `Created_tmp_files`. Does that mean this session created seven temporary files, while creating zero temporary tables? No—in fact, the `Created_tmp_files` status variable has only a global scope. This is a known issue with MySQL at the moment: you always see all status variables, regardless of the requested scope, but their values will be properly scoped. The MySQL documentation includes a helpful "Server

Status Variable Reference" that can help you understand the scope of the different variables.

Unlike `Created_tmp_files`, the `Com_show_status` variable has scope "both," meaning that you can get a global counter as well as a per-session value. Let's see that in practice:

```
mysql> SHOW STATUS LIKE 'Com_show_status';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Com_show_status | 13    |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL STATUS LIKE 'Com_show_status';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Com_show_status | 45    |
+-----------------+-------+
1 row in set (0.00 sec)
```

Another important thing to note when looking at the status variables is that it's possible to reset most of them back to zero on a session level. That is achieved by running the `FLUSH STATUS` command. This command resets status variables within all connected sessions to zero, after adding their current values to the global counters. Thus, `FLUSH STATUS` operates on a session level, but for all sessions. To illustrate this, we'll reset the status variable values in the session we used before:

```
mysql> FLUSH STATUS;

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

```
mysql> SHOW STATUS LIKE 'Com_show_status';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Com_show_status | 1     |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL STATUS LIKE 'Com_show_status';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| Com_show_status | 49    |
+-----------------+-------+
1 row in set (0.00 sec)
```

Even though the global counter keeps on increasing, the session counter was reset to `0` and incremented to only `1` when we ran the `SHOW STATUS` command. This can be useful to see in isolation how, for example, running a single query changes the status variable values (in particular, the `Handler_*` family of status variables).

Note that it's impossible to reset global counters without a restart.

Basic Monitoring Recipes

You could monitor numerous metrics, in different combinations. We believe, however, that there are a few that must be in the toolbox of every database operator. While you are learning MySQL, these should be enough to give you a reasonable sense of how your database is doing. Most of the existing monitoring systems should have these covered and usually include many more metrics. You may never set up collection yourself, but our explanation should also allow you to get a better understanding of just what your monitoring system tells you.

We're going to give few broad categories of metrics in the following subsections, within which we'll detail what we think are some of the more

important counters.

MySQL server availability

This is the most important thing you should monitor. If MySQL Server is not accepting connections or is not running, all the other metrics don't matter at all.

MySQL Server is a robust piece of software that's capable of running with an uptime of months or years. Yet, there are situations that can lead to a premature unplanned shutdown (or, more plainly, a crash). For example, in "Memory" we discussed that out-of-memory conditions may lead to MySQL crashing or being killed. Other incidents happen, too. There are crashing bugs in MySQL, however rare they are nowadays. There are also operational mistakes: who hasn't forgotten to bring up a database after planned maintenance? Hardware fails, servers restart—many things may compromise the availability of MySQL.

There are a few approaches to monitoring MySQL availability, and there's no single best one; it's better to combine a few. A very simple basic approach is to check that the `mysqld` (or `mysqld.exe`) process is actually running and visible from the OS level. On Linux and Unix-like systems you can use the `ps` command for this, and on Windows, you can check the Task Manager or run the `Get-Service` PowerShell command. This is not a useless check, but it has its issues. For one, the fact that MySQL is running does not guarantee that it's actually doing what it should be—that is, processing clients' queries. MySQL could be swamped by load or suffering from a disk failure and running unbearably slowly. From the OS's perspective, the process is running, but from the client's perspective it's as good as shut down anyway.

The second approach is to check MySQL's availability from the application's point of view. Usually, that's achieved by running the MySQL monitor and executing some simple, short query. Unlike the previous check, this one makes sure that a new connection to MySQL can be established and that the database is processing queries. You can locate these checks on the application side to make them even closer to how apps see the database. Instead of setting up such checks as independent entities, applications can be adjusted to probe MySQL and report clear errors either to operators or to the monitoring system.

The third approach lies between the previous two and is concentrated on the DB side. While monitoring MySQL, you will at least need to execute simple queries against the database to check status variables. If those queries fail, your monitoring system should alert, as potentially MySQL is starting to have issues. A variation would be to check whether any data from the target instance has been received by your monitoring system in the last few minutes.

What ideally should come out of these checks is not only the alert "MySQL is down" at the appropriate time but also some clue as to why it is down. For example, if the second type of check cannot initiate a new connection because MySQL has run out of connections, then that should be a part of the alert. If the third type of check is failing, but the first type is okay, then it's a different situation than a crash.

Client connections

MySQL Server is a multithreaded program, as was laid out in depth in "The MySQL Server Daemon". Every client connecting to a database causes a new thread to be spawned within the process of the MySQL server (`mysqld` or `mysqld.exe`). That thread will be responsible for executing statements sent by a client, and in theory there can be as many concurrent queries executing as there are client threads.

Each connection and its thread put some low overhead on the MySQL server, even when they are idle. Apart from that, from the database's point of view, each connection is a liability: the database cannot know when a connection will send a statement. *Concurrency*, or the number of simultaneously running transactions and queries, usually increases with an increase in the number of established connections. Concurrency is not bad in itself, but each system will have a limit of scalability. As you'll remember from "Operating System Metrics", CPU and disk resources have performance limits, and it's impossible to push past them. And even with an infinite amount of OS resources, MySQL itself has internal scalability limits.

To put it simply: the number of connections, especially active connections, should ideally be kept minimal. From the MySQL side, no connections is the perfect situation, but from the application side that's unacceptable. Some applications, though, make no attempt at limiting the number of connections they make and queries they send, assuming that the database will take care of the load. This can create a dangerous situation known as a *thundering herd*:

for some reason, queries run longer, and the app reacts by sending more and more queries, overloading the database.

Finally, MySQL has an upper limit on the number of client connections, controlled by the system variable `max_connections`. Once the number of existing connections hits the value of that variable, MySQL will refuse to create new connections. That's a bad thing. `max_connections` should be used as a protection from complete server meltdown if clients establish thousands of connections. But ideally, you should monitor the number of connections and work with the app teams to keep that number low.

Let's review the specific connection and thread counters that MySQL exposes:

`Threads_connected`

: Number of currently connected client threads, or in other words number of established client connections. We've been explaining the importance of this one for the last few paragraphs, so you should know why you have to check it.

`Threads_running`

: Number of client threads that are currently executing a statement. Whereas `Threads_connected` indicates a potential for high concurrency, `Threads_running` actually shows the current measure of that concurrency. Spikes in this counter indicate either an increased load from the application or a slowness in the database leading to queries stacking up.

`Max_used_connections`

: Maximum number of connections established that was recorded since the last MySQL server restart. If you suspect that a connection flood happened, but don't have a recorded history of changes in `Threads_connected`, you can check this status variable to see the highest peak recorded.

`Max_used_connections_time`

: Date and time when MySQL Server saw the maximum number of connections since the last restart to date.

Another important metric to monitor about connections is their rate of failure. An increased rate of errors may indicate that your applications are having trouble communicating with your database. MySQL distinguishes between connections that clients failed to establish and existing connections that failed due to a timeout, for example:

`Aborted_clients`

Number of already established connections that were aborted. The MySQL documentation mentions "the client died without closing the connection properly," but this can also happen if there's a network issue between the server and client. Frequent sources of increases in this counter are `max_allowed_packet` violations (see "Scope of Options") and session timeouts (see the `wait_timeout` and `interactive_timeout` system variables). Some errors are to be expected, but sharp spikes should be checked.

`Aborted_connects`

Number of new connections that failed to be established. Causes include incorrect passwords, connecting to a database for which the user has no permission, protocol mismatches, `connect_timeout` violations, and reaching `max_connections`, among other things. They also include various network-related issues. There's a family of status variables under the `Connection_errors_%` wildcard that look in more depth into some of the specific issues. An increase in `Aborted_connects` should be checked, as it can indicate an application configuration issue (wrong user/password) or a database issue (running out of connections).

NOTE

MySQL Enterprise Edition, Percona Server, and MariaDB offer a thread pool functionality. This changes the connection and thread accounting. With a thread pool, the number of connections stays the same, but the number of threads running within MySQL is limited by the size of the pool. When a connection needs to execute a statement, it will have to get an available thread from the pool and wait if such a thread is not available. Using a thread pool improves MySQL performance with hundreds or thousands of connections. Since this feature is not available in regular MySQL and we believe it's an advanced one, we're not covering it in this book.

Query counts

The next broad category of metrics is query-related metrics. Where `Threads_running` shows how many sessions are active at once, metrics in this category will show the quality of the load those sessions produce. Here, we'll start by looking at the overall amount of queries, then move on to breaking down queries by type, and last but not least we'll look into how the queries execute.

It's important to monitor query counts. Thirty running threads may each be executing a single hour-long query, or a few dozen queries per second. The conclusions you make will be completely different in each case, and the load profile will likely change, too. These are the important metrics showing the number of queries executed:

`Queries`

This global status variable, simply put, gives the number of statements executed by the server (excluding `COM_PING` and `COM_STATISTICS`). If you run `SHOW GLOBAL STATUS LIKE 'Queries'` on an idle server, you will see the counter value increasing with each execution of the `SHOW STATUS` command.

`Questions`

Almost the same as `Queries`, but excludes statements executed within stored procedures, and also the following types of queries: `COM_PING`, `COM_STATISTICS`, `COM_STMT_PREPARE`, `COM_STMT_CLOSE`, or `COM_STMT_RESET`. Unless your database clients use stored procedures extensively, the `Questions` metric is closer to the amount of actual queries being executed by the server, compared to the amount of statements in `Queries`. In addition to that, `Questions` is both a session-level and a global status variable.

TIP

Both `Queries` and `Questions` are incremented when the query starts executing, so it's necessary to also look at the `Threads_running` value to see how many queries are actually being executed right now.

QPS

Queries per second. This is a synthetic metric that you can arrive at by looking at how the `Queries` variable changes over time. QPS based on `Queries` will include almost any statement that the server executes.

The QPS metric does not tell us about the quality of queries executed—that is, the extent of their impact on the server—but it's a useful gauge nevertheless. Usually, the load on a database from applications is regular. It may move in waves (more during the day, fewer at night), but over a week or a month a pattern of number of queries over time will show. When you get a report about a database being slow, looking at QPS may give you a quick indication of whether there's been a sudden unexpected growth in application load. A drop in QPS, on the other hand, may indicate that issues are on the database side, as it cannot process as many queries as usual in the same time.

Query types and quality

The next logical step from knowing the QPS is understanding what types of queries are being executed by the clients, and the impact of those queries on the server. All queries are not equal, and some, you may say, are *bad*, or produce unnecessary load on the system. Looking for and catching such queries is an important part of monitoring. In this section we're trying to answer the question "are there a lot of bad queries?" and in "The Slow Query Log" we'll show you how to catch the specific offenders.

Types of queries

Each query MySQL executes has a type. What's more, any *command* that you can execute has a type. MySQL keeps track of the different types of commands and queries executed with the `Com_%` family of status variables. There are 172 of these variables in MySQL 8.0.25, accounting for almost a third of all status variables. As you can guess from this number, MySQL counts a lot of commands that you perhaps wouldn't even think of: for example, `Com_uninstall_plugin` counts the number of times `UNINSTALL PLUGIN` was called, and `Com_help` counts uses of the `HELP` statment.

Every `Com_%` status variable is available on both the global and session levels, as was shown with `Com_show_status` in "Status Variables".

However, MySQL doesn't expose other threads' counters for `Com_%` variables, so for monitoring purposes, global status variables are assumed here. It's possible to get other sessions' statement counters, but that's achieved through the Performance Schema family of events, called `statement/sql/%`. That can be useful to attempt to find a thread that's sending a disproportional amount of some type of statement, but it's a bit advanced and falls under investigation rather than monitoring. You can find more details in the "Performance Schema Status Variable Tables" section of the MySQL documentation.

Since there are so many `Com_%` status variables, monitoring every type of command would be both too noisy and unnecessary. You should, however, try to store the values of all of them. You can go two ways with looking at these counters.

The first option is to pick the command types that are relevant for your database load profile, and monitor those. For example, if your database clients do not use stored procedures, then looking at `Com_call_procedure` is a waste of time. A good starting selection is to cover `SELECT` and basic DML statements, which usually comprise the bulk of any database system's load—for example, `Com_select`, `Com_insert`, `Com_update`, and `Com_delete` (the status variables' names are self-explanatory here). One interesting thing MySQL does is account for multitable updates and deletes (see "Performing Updates and Deletes with Multiple Tables") separately, under `Com_update_multi` and `Com_delete_multi`; these should also be monitored, unless you're sure such statements are never run in your system.

You can then look at all of the `Com_%` status variables, see which ones are growing, and add those to your selection of monitored variables. Unfortunately, the flaw with this approach is that you can miss some unexpected spikes.

Another way of looking at these counters could be to look at the top 5 or 10 of them over time. This way, a sudden change of load pattern can be more difficult to miss.

Knowing what types of queries are running is important in shaping an overall understanding of the load on a given database. Moreover, it changes how you approach tuning the database, because, for example, an insert-heavy workload

may require a different setup compared to a read-only or mostly-read workload. Changes in query load profile, like a sudden appearance of thousands of `UPDATE` statements executed per second, can indicate changes on the application side.

Query quality

The next step after knowing what queries are running is to understand their quality, or their impact on the system. We mentioned this, but it's worth reiterating: all queries are not equal. Some will put more burden on the system than others. Looking at the overall query-related metrics may give you advance warning of problems growing in the database. You will learn that it's possible to notice problematic behaviors by monitoring just a few counters.

`Select_scan` counts the number of queries that caused a full table scan, or in other words forced MySQL to read the whole table to form the result. Now, we should immediately acknowledge that full table scans are not always a problem. After all, sometimes just reading all of the data in a table is a viable query execution strategy, especially when the number of rows is low. You can also expect to always see some amount of full table scans happening, as a lot of MySQL catalog tables are read that way. For example, just running a `SHOW GLOBAL STATUS` query will cause `Select_scan` to increase by two. Often, however, full table scans imply that there are queries hitting the database that are performing suboptimally: either they are improperly written and don't filter out data efficiently, or there are simply no indexes that the queries can use. We give more information about query execution details and plans in "The EXPLAIN Statement".

`Select_full_join` is similar to `Select_scan`, but counts the number of queries that caused a full table scan on a referenced table within a `JOIN` query. The referenced table is the rightmost table in the `JOIN` condition—see "Joining Two Tables" for more information. Again, as with `Select_scan`, it's hard to say that a high `Select_full_join` count is always bad. It's common in large data warehouse systems, for example, to have compact dictionary tables, and reading those fully may not be a problem. Still, usually a high value for this status variable indicates the presence of badly behaving queries.

`Select_range` counts the number of queries that scanned data with some range condition (covered in "Selecting Rows with the WHERE Clause").

Usually this is not a problem at all. If the range condition is not possible to satisfy using an index, then the value of `Select_scan` or `Select_full_join` will grow alongside this status variable. Probably the only time when this counter's value may indicate an issue is when you see it growing even though you know that most of the queries running in the database in fact do not utilize ranges. As long as the associated table scan counters aren't growing as well, the issue is likely still benign.

`Select_full_range_join` combines `Select_range` and `Select_full_join` . This variable holds a counter for queries that caused a range scan on referenced tables in `JOIN` queries.

So far, we've been counting individual queries, but MySQL also does a similar accounting for *every row* it reads from the storage engines! The family of status variables showing those counters are the `Handler_%` variables. Simply put, every row MySQL reads increments some `Handler_%` variable. Combining this information with the query type and query quality counters you've seen so far can tell you, for example, if full table scans that run in your database are a problem at all.

The first handler we'll look at is `Handler_read_rnd_next` , which counts the number of rows read when a full or partial table scan is performed. Unlike the `Select_%` status variables, the `Handler_%` variables do not have nice, easy-to-remember names, so some memorization is necessary. High values in the `Handler_read_rnd_next` status variable in general indicate that either a lot of tables are not indexed properly or many queries do not utilize the existing indexes. Remember we mentioned when explaining `Select_scan` that some full table scans are not problematic. To see whether that's true or not in your case, look at the ratio of `Handler_read_rnd_next` to other handlers. You want to see a low value for that counter. If your database returns on average a million rows per minute, then you probably want the number of rows returned by full scans to be in the thousands, not tens or hundreds of thousands.

`Handler_read_rnd` counts the number of rows usually read when the sorting of a result set is performed. High values may indicate the presence of many full table scans and joins not using indexes. However, unlike with `Handler_read_rnd_next` , this is not a sure sign of problems.

`Handler_read_first` counts how many times the first index entry was read. A high value for this counter indicates that a lot of full index scans are occurring. This is better than full table scans, but still a problematic behavior. Likely, some of the queries are missing filters in their `WHERE` clauses. The value of this status variable should again be viewed in relation to the other handlers, as some full index scans are unavoidable.

`Handler_read_key` counts the number of rows read by an index. You want this handler's value to be high compared to other read-related handlers. In general, a high number here means your queries are using indexes properly.

Note that handlers still can hide some issues. If a query only reads rows using indexes, but does so inefficiently, then `Select_scan` will not be increased, and `Handler_read_key` —our good read handler—will grow, but the end result will still be a slow query. We explain how to find specific slow queries in "The Slow Query Log", but there's also a special counter for them: `Slow_queries` . This status variable counts the queries that took longer than the value of `Long_query_time` to execute, regardless of whether the slow query log is enabled. You can gradually drop the value of `Long_query_time` and see when `Slow_queries` starts to approach the total number of queries executed by your server. This is a good way to assess how many queries in your system take, for example, longer than a second without actually turning on the slow query log, which has an overhead.

Not every query executed is read-only, and MySQL also counts the number of rows inserted, updated, or deleted under, respectively, the `Handler_insert` , `Handler_update` , and `Handler_delete` status variables. Unlike with `SELECT` queries, it's hard to make conclusions about the quality of your write statements based on the status variables alone. However, you can monitor these to see if, for example, your database clients start updating more rows. Without a change in the number of `UPDATE` statements (the `Com_update` and `Com_update_multi` status variables), that may indicate a change in the parameters passed to the same queries: wider ranges, more items in the `IN` clauses, and so on. This may not indicate a problem on its own, but it may be used during investigation of slowness to see whether more strain is being put on the database.

Apart from `INSERT` statements, `UPDATE` , `DELETE` , and even `INSERT SELECT` statements have to look for rows to change. Thus, for example, a `DELETE` statement will increase read-related counters and may result in an

unexpected situation: no `Select_scan` growth, but an increasing `Handler_read_rnd_next` value. Do not forget about this peculiarity if you see a discrepancy between status variables. The slow query log will include `SELECT` as well as DML statements.

Temporary objects

Sometimes, when queries execute, MySQL needs to create and use temporary objects, which may reside in memory or on disk. Examples of reasons for temporary object creation include use of the `UNION` clause, derived tables, common table expressions, and some variations of `ORDER BY` and `GROUP BY` clauses, among other things. We've been saying this about almost everything in this chapter, but temporary objects are not a problem: some number of them is unavoidable and actually desired. Yet they do eat into your server's resources: if temporary tables are small enough, they'll be kept in memory and use it up, and if they grow large, MySQL will start offloading them to disk, using up both the disk space and affecting performance.

MySQL maintains three status variables related to temporary objects created during query execution. Note that this doesn't include temporary tables created explicitly through the `CREATE TEMPORARY TABLE` statement; look for those under the `Com_create_table` counter.

`Created_tmp_tables` counts the number of temporary tables created implicitly by MySQL server while executing various queries. You cannot know why or for which queries these were created, but every table will be accounted for here. Under a stable workload, you should see a uniform number of temporary tables created, as roughly the same queries run the same amount of times. Growth of this counter is usually associated with changing queries or their plans, for example due to growth of the database, and may be problematic. Although useful, creating temporary tables, even in memory, takes resources. You cannot completely avoid temporary tables, but you should check why their number is growing by performing a query audit with the slow query log, for example.

`Created_tmp_disk_tables` counts the number of temporary tables that "spilled," or were written to disk after their size surpassed the configured upper limits for in-memory temporary tables. With the older Memory engine, the limit was controlled by `tmp_table_size` or `max_heap_table_size`. MySQL 8.0 moved by default to a new

TempTable engine for the temporary tables, which, by default, does not spill to disk in the same way Memory tables did. If the `temptable_use_mmap` variable is set to its default of `ON`, then TempTable temporary tables do not increase this variable even if they are written to disk.

`Created_tmp_files` counts the number of temporary files created by MySQL. This is different from Memory engine temporary tables spilling to disk, but will account for TempTable tables being written out to disk. We understand that this may seem complicated, and it truly is, but major changes don't usually come without some downsides.

Whatever configuration you're using, sizing the temporary tables is important, as is monitoring the rate of their creation and spillage. If a workload creates a lot of temporary tables of roughly 32 MB in size, but the upper limit for in-memory tables is 16 MB, then the server will see an increased rate of I/O due to those tables being written out to and read back from the disk. That's fine for a server strapped for memory, but it's a waste if you have memory available. Conversely, setting the upper limit too high may result in the server swapping or outright crashing, as explained in "Memory".

We've seen servers brought down by memory spikes when lots of simultaneously open connections all ran queries requiring temporary tables. We've also seen servers where the bulk of I/O load was produced by temporary tables spilling to disk. As with most things related to operating databases, the table sizing decision is a balancing act. The three counters we've shown can help you make an informed choice.

InnoDB I/O and transaction metrics

So far, we've been mostly talking about overall MySQL metrics and ignoring the fact that there are things like transactions and locking. In this subsection we'll take a look at some of the useful metrics the InnoDB storage engine exposes. Some of those metrics relate to how much data InnoDB reads and writes, and why. Some, however, can show important information on locking, which can be combined with MySQL-wide counters to get a solid grasp on the current locking situation in a database.

The InnoDB storage engine provides 61 status variables showing various information about its internal state. By looking at their change over time you can see how loaded InnoDB is and how much load on the OS it produces.

Given that InnoDB is the default storage engine, that will likely be most of the load MySQL produces.

We perhaps should've put these in the section about query quality, but InnoDB maintains its own counters for the number of rows it has read, inserted, updated, or deleted. The variables are, respectfully, `Inndb_rows_read`, `Inndb_rows_inserted`, `Inndb_rows_updated`, and `Inndb_rows_deleted`. Usually their values correspond pretty well to the values of the related `Handler_%` variables. If you primarily use InnoDB tables, it may be simpler to use the `Innodb_rows_%` counters instead of the `Handler_%` ones to monitor relative load from queries expressed in the number of rows processed.

Other important and useful status variables InnoDB provides show the amount of data that the storage engine reads and writes. In "Disk" we saw how to check and monitor overall and per-process I/O utilization. InnoDB allows you to see exactly why it's reading and writing data, and how much of it:

`Innodb_data_read`

The amount of data expressed in bytes read from disk since server startup. If you take measurements of this variable's value over time, you can translate it into bandwidth utilization in bytes/second. This metric is tightly related to InnoDB buffer pool sizing and its effectiveness, and we'll get to that in a bit. All of this data can be assumed to be read from the data files to satisfy queries.

`Innodb_data_written`

The amount of data expressed in bytes written to disk since server startup. This is the same as `Innodb_data_read`, but in the other direction. Usually, this value will account for a large portion of the overall amount of write bandwidth MySQL will generate. Unlike with reading data, InnoDB writes data out in a variety of situations; thus, there are additional variables specifying parts of this I/O, as well as other sources of I/O.

`Innodb_os_log_written`

The amount of data expressed in bytes written by InnoDB into its redo logs. This amount is also included in `Innodb_data_written`, but

it's worth monitoring individually to see if your redo logs may need a size change. See "Redo log size" for more details.

Innodb_pages_written

The amount of data expressed in pages (16 KiB by default) written by InnoDB during its operation. This is the second half of the `Innodb_data_written` status variable. It's useful to see the amount of non-redo I/O that InnoDB generates.

Innodb_buffer_pool_pages_flushed

The amount of data expressed in pages written by InnoDB due to flushing. Unlike the writes covered by two previous counters, writes caused by flushing do not happen immediately after an actual write is performed. Flushing is a complex background operation, the details of which are beyond the scope of our book. However, you should at least know that flushing exists and that it generates I/O independent of other counters.

By combining `Innodb_data_written` and `Innodb_buffer_pool_pages_flushed`, you should be able to come up with a pretty accurate figure for the disk bandwidth utilized by InnoDB and MySQL Server. Adding `Innodb_data_read` completes the I/O profile of InnoDB. MySQL doesn't only use InnoDB, and there can be I/O from other parts of the system, like temporary tables spilling to disk, as we discussed earlier. Yet often InnoDB I/O matches that of MySQL Server observed from the OS.

One use of this information is to see how close your MySQL Server is to hitting the limits of your storage system's performance capacity. This is especially important in the cloud where storage often has strict limits. During incidents related to database performance, you can check the I/O-related counters to see if MySQL is writing or reading more, perhaps indicating increased load, or instead doing fewer actual I/O operations. The latter may mean that MySQL is currently limited by some other resource, like CPU, or suffers from other issues, like locking. Unfortunately, decreased I/O may also mean that the storage is having issues.

There are some status variables in InnoDB that may help to find issues with storage or its performance: `Innodb_data_pending_fsyncs`, `Innodb_data_pending_reads`, `Innodb_data_pending_writes`,

`Innodb_os_log_pending_fsyncs`, and `Innodb_os_log_pending_writes`. You can expect to see some amount of pending data reads and writes, though as always it's helpful to look at the trends and previous data. The most important of all of these is `Innodb_os_log_pending_fsyncs`. Redo logs are synced often, and the performance of the syncing operation is extremely important for the overall performance and transaction throughput of InnoDB.

Unlike many other status variables, all of these are gauges, meaning that their values go up and down and don't just increase. You should sample these variables and look at how often there are pending operations, in particular for the redo log sync. Even small increases in `Innodb_os_log_pending_fsyncs` may indicate serious issues with storage: either you're running out of performance capacity or there are hardware issues.

While writing about the `Innodb_data_read` variable, we mentioned that the amount of data that InnoDB reads is related to its buffer pool size and usage. Let's elaborate on that. InnoDB caches pages it reads from disk inside its buffer pool. The larger the buffer pool is, the more pages will be stored there, and the less frequently pages will have to be read from disk. We talk about that in "Buffer pool size". Here, while discussing monitoring, let's see how to monitor the effectiveness of the buffer pool. That's easily done with just two status variables:

`Innodb_buffer_pool_read_requests`

The MySQL documentation defines this as "the number of logical read requests." Put simply, this is the number of pages that various operations within InnoDB wanted to read from the buffer pool. Usually most of the pages are read due to query activity.

`Innodb_buffer_pool_reads`

This is the number of pages that InnoDB had to read from disk to satisfy the read requests by queries or other operations. The value of this counter is usually smaller than or equal to `Innodb_buffer_pool_read_requests` even in the very worst case with a completely empty (or "cold") buffer pool, because reads from disk are performed to satisfy the read requests.

Under normal conditions, even with a small buffer pool, you won't get a 1:1 ratio between these variables. That is, it will be possible to satisfy at least some reads from the buffer pool. Ideally, you should try keep the number of disk reads to a minimum. That may not always be possible, especially if the database size is much larger than server memory.

You may attempt estimating a buffer pool hit ratio, and there are formulas available online. However, comparing the two variables' values is not exactly correct, like comparing apples and oranges. If you think that your `Innodb_buffer_pool_reads` is too high, it may be worth going through queries running on the system (for example, using slow query log) instead of trying to increase the buffer pool size. Of course, you should try to keep buffer pool as large as possible to cover most or all of the hot data in the database. However, there will still be queries that may cause high read I/O through getting pages from disk (and increasing `Innodb_buffer_pool_reads` while doing so), and attempting to fix them by increasing the buffer pool size even more will provide diminishing returns.

Finally, to close our discussion of InnoDB, we'll move on to transaction and locking. A lot of information on both topics was given in Chapter 6, so here we'll do a brief overview of the related status variables:

Transaction-related command counters

`BEGIN`, `COMMIT`, and `ROLLBACK` are all special MySQL commands. Thus, MySQL will count the number of times they were executed with `Com_%` status variables: `Com_begin`, `Com_commit`, and `Com_rollback`. By looking at these counters you can see how many transactions are started explicitly and either committed or rolled back.

Locking-related status variables

You know by now that InnoDB provides locking with row-level granularity. This is a huge improvement over MyISAM's table-level locking, as the impact of each individual lock is minimized. Still, there can be an impact if transactions are waiting for each other even for a short time.

InnoDB provides status variables that let you see just how many locks are being created and give you details on the lock waits that are

happening:

`Innodb_row_lock_current_waits` shows how many transactions operating on InnoDB tables are currently waiting for a lock to be released by some other transactions. The value of this variable will go up from zero when there are blocked sessions then go back to zero as soon as locking is resolved.

`Innodb_row_lock_waits` shows how many times since server startup transactions on InnoDB tables have waited for row-level locks. This variable is a counter and will continually increase until MySQL Server is restarted.

`Innodb_row_lock_time` shows the total time in milliseconds spent by sessions trying to acquire locks on InnoDB tables.

`Innodb_row_lock_time_avg` shows an average time in milliseconds that it takes for a session to acquire a row-level lock on an InnoDB table. You can arrive at the same value by dividing `Innodb_row_lock_time` by `Innodb_row_lock_waits` . This value may go up and down depending on how many lock waits are encountered and how much accumulated lock time grows.

`Innodb_row_lock_time_max` shows the maximum time in milliseconds it took to obtain a lock on an InnoDB table. This value will go up only if the record is broken by some other unfortunate transaction.

Here's an example from a MySQL server running a moderate read/write load:

```
mysql> SHOW GLOBAL STATUS LIKE 'Innodb_row_lock%'
```

```
+-------------------------------+--------+
| Variable_name                 | Value  |
+-------------------------------+--------+
| Innodb_row_lock_current_waits | 0      |
| Innodb_row_lock_time          | 367465 |
| Innodb_row_lock_time_avg      | 165    |
| Innodb_row_lock_time_max      | 51056  |
```

```
| Innodb_row_lock_waits          | 2226   |
+--------------------------------+--------+
5 rows in set (0.00 sec)
```

There were 2,226 individual transactions waiting for locks, and it took 367,465 milliseconds to obtain all of those locks, with an average lock acquisition duration of 165 milliseconds and a maximum duration of just over 51 seconds. There are currently no sessions waiting for locks. On its own this information doesn't tell us much: it's neither a lot nor a little. However, we know that at the same time more than 100,000 transactions were executed by this MySQL server. The resulting locking metric values are more than reasonable for the level of concurrency.

Locking issues are a frequent source of headache for database administrators and application developers alike. While these metrics, like everything we've discussed so far, are aggregated across every session running, deviation from the normal values may help you in pinning down some of the issues. To find and investigate individual locking situations, you may use the InnoDB status report; see "InnoDB Engine Status Report" for more details.

The Slow Query Log

In "Query types and quality" we showed how to look for tell-tale signs of unoptimized queries in MySQL. However, that's not enough to start optimizing those queries. We need specific examples. There are few ways to do that, but probably the most robust one is using the slow query log facility. The slow query log is exactly what it sounds like: a special text log where MySQL puts information about *slow* queries. Just how slow those queries should be is controllable, and you can go as far as logging every query.

To enable the slow query log, you must change the setting of the `slow_query_log` system variable to `ON` from its default of `OFF`. By default, when the slow query log is enabled, MySQL will log queries taking longer than 10 seconds. That's configurable by changing the `long_query_time` variable, which has a minimum value of 0, meaning every query executed by the server will be logged. The log location is controlled by the `slow_query_log_file` variable, which defaults to a value of *`hostname`*`-slow.log`. When the path to the slow query log is

relative, meaning it doesn't start from / on Linux or, for example, *C:* on Windows, then this file will be located in the MySQL data directory.

You can also tell MySQL to log queries not using indexes regardless of the time they take to execute. To do so, the `log_queries_not_using_indexes` variable has to be set to `ON`. By default, DDL and administrative statements are not logged, but this behavior can be changed by setting `log_slow_admin_statements` to `ON`.

MariaDB and Percona Server expand the functionality of the slow query log by adding filtering capabilities, rate limiting, and enhanced verbosity. If you're using those products, it's worth reading their documentation on the subject to see if you can utilize the enhanced slow query log.

Here's an example from a record in the slow query log showing a `SELECT` statement taking longer than the configured `long_query_time` value of 1 second:

```
# Time: 2021-05-29T17:21:12.433992Z
# User@Host: root[root] @ localhost []  Id:    11
# Query_time: 1.877495  Lock_time: 0.000823 Rows_sent:
use employees;
SET timestamp=1622308870;
SELECT
        dpt.dept_name
      , emp.emp_no
      , emp.first_name
      , emp.last_name
      , sal.salary
FROM
        departments dpt
    JOIN dept_emp ON dpt.dept_no = dept_emp.dept_no
    JOIN employees emp ON dept_emp.emp_no = emp.emp_no
    JOIN salaries sal ON emp.emp_no = sal.emp_no
    JOIN (SELECT dept_emp.dept_no, MAX(sal.salary) maxs
        FROM dept_emp JOIN salaries sal
            ON dept_emp.emp_no = sal.emp_no
        WHERE
                sal.from_date < now()
            AND sal.to_date > now()
        GROUP BY dept_no
```

```
    ) largest_sal_by_dept ON dept_emp.dept_no = largest
        AND sal.salary = largest_sal_by_dept.maxsal;
```

By analyzing this output you can immediately start making conclusions about this query. This is much more descriptive than looking at server-wide metrics. For example, we can see that this query was executed at 17:21:12 UTC by user `root@localhost` in the `employees` database, took 1.88 seconds to run, and produced 9 rows but had to scan 3,473,725 rows to produce that result. That information on its own can tell you a whole lot about the query, especially once you've gained more experience with MySQL. You now also have the complete query text, which you can turn into execution plan information to see exactly how MySQL is executing this query. You can find more details on that process in "The EXPLAIN Statement".

If the `long_query_time` is set low, the slow query log may grow large. Sometimes this may be reasonable, but reading through the resulting log may be nearly impossible if the number of queries is high. There's a tool called `mysqldumpslow` that solves this problem. It takes the path to the slow query log file as an argument and will summarize queries from that file and sort them (by default by time). In the following example, the command is run such that it will show the two top queries sorted by the number of rows returned:

```
$ mysqldumpslow -s r -t 2 /var/lib/mysql/mysqldb1-slow.
```

```
Reading mysql slow query log from /var/lib/mysql/mysqlc
Count: 2805  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=100
sbuser]@localhost
  SELECT c FROM sbtest1 WHERE id BETWEEN N AND N

Count: 2760  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=100
sbuser]@localhost
  SELECT c FROM sbtest1 WHERE id BETWEEN N AND N ORDER
```

You can see that these two queries alone were recorded in the slow query log 5,565 times. Imagine trying to get that information without help! Another tool that can help with summarizing the information in the slow query log is `pt-query-digest` from Percona Toolkit. The tool is a bit more advanced and more difficult to use than `mysqldumpslow`, but gives a lot of information and has many features. The report it produces starts with a summary:

```
$ pt-query-digest /var/lib/mysql/mysqldb1-slow.log


# 7.4s user time, 60ms system time, 41.96M rss, 258.35M
# Current date: Sat May 29 22:36:47 2021
# Hostname: mysqldb1
# Files: /var/lib/mysql/mysqldb1-slow.log
# Overall: 109.42k total, 15 unique, 7.29k QPS, 1.18x c
# Time range: 2021-05-29T19:28:57 to 2021-05-29T19:29:1
# Attribute          total     min     max     avg
# ============     ======= ======= ======= ======= ====
# Exec time            18s     1us    10ms   161us
# Lock time             2s       0     7ms    16us    1
# Rows sent          1.62M       0     100   15.54   97
# Rows examine       3.20M       0     200   30.63  192
# Query size         5.84M       5     245   55.93  151

# Profile
# Rank Query ID                           Response tim
# ==== ================================== ============
#    1 0xFFFCA4D67EA0A78…  11.1853 63.1%  5467 0.0020
#    2 0xB2249CB854EE3C2…   1.5985  9.0%  5467 0.0003
#    3 0xE81D0B3DB4FB31B…   1.5600  8.8% 54670 0.0000
#    4 0xF0C5AE75A52E847…   0.8853  5.0%  5467 0.0002
#    5 0x9934EF6887CC7A6…   0.5959  3.4%  5467 0.0001
#    6 0xA729E7889F57828…   0.4748  2.7%  5467 0.0001
#    7 0xFF7C69F51BBD3A7…   0.4511  2.5%  5467 0.0001
#    8 0x6C545CFB5536512…   0.3092  1.7%  5467 0.0001
# MISC 0xMISC               0.6629  3.7% 16482 0.0000
```

And each query is then summarized as follows:

```
# Query 2: 546.70 QPS, 0.16x concurrency, ID 0xB2249CB8
# Scores: V/M = 0.00
# Time range: 2021-05-29T19:29:02 to 2021-05-29T19:29:1
# Attribute    pct   total     min     max     avg
# ============ === ======= ======= ======= ======= ====
# Count          4    5467
# Exec time      9      2s    54us     7ms   292us
# Lock time     61      1s     7us     7ms   203us
# Rows sent      0       0       0       0       0
# Rows examine   0   5.34k       1       1       1
# Query size     3 213.55k      40      40      40
# String:
```

```
# Databases    sysbench
# Hosts        localhost
# Users        sbuser
# Query_time distribution
#   1us
#  10us  ##################################################
# 100us  ##################################################
#   1ms  ############
#  10ms
# 100ms
#    1s
#  10s+
# Tables
#    SHOW TABLE STATUS FROM `sysbench` LIKE 'sbtest2'\G
#    SHOW CREATE TABLE `sysbench`.`sbtest2`\G
UPDATE sbtest2 SET k=k+1 WHERE id=497658\G
# Converted for EXPLAIN
# EXPLAIN /*!50100 PARTITIONS*/
select  k=k+1 from sbtest2 where  id=497658\G
```

That's a lot of valuable information in a dense format. One of the distinguishing features of this output is the query duration distribution visualization, which allows you to quickly see whether a query has parameter-dependent performance issues. Explaining every feature of `pt-query-digest` would take another chapter, and it's an advanced tool, so we leave this for you to try once you're done learning MySQL.

The slow query log is a powerful tool that allows you to get a very detailed view of the queries executed by MySQL Server. We recommend using the slow query log like this:

- Set `long_query_time` to a value large enough that it covers most of the queries normally running in your system, but small enough that you catch outliers. For example, in an OLTP system, where most of the queries are expected to complete in milliseconds, a value of `0.5` may be reasonable, only catching relatively slow queries. On the other hand, if your system has queries running in minutes, then `long_query_time` should be set accordingly.
- Logging to the slow query log has some performance cost, and you should avoid logging more queries than you need. If you have the slow query log enabled, make sure you adjust the `long_query_time` setting if you find the log too noisy.

- Sometimes you may want to perform a “query audit,” where you temporarily (for a few minutes) set `long_query_time` to `0` to catch every query. This is a good way to get a snapshot of your database load. Such snapshots may be saved and compared later. However, we recommend strongly against setting `long_query_time` too low.
- If you have the slow query log set up, we recommend running `mysqldumpslow`, `pt-query-digest`, or a similar tool on it periodically to see if there are new queries appearing or if existing ones start behaving worse than usual.

InnoDB Engine Status Report

The InnoDB storage engine has a built-in report that exposes deep technical details on the current state of the engine. A lot can be said about InnoDB load and performance from reading just this one report, ideally sampled over time. Reading the InnoDB status report is an advanced topic that requires more instruction than we can convey in our book, and also a lot of practice. Still, we believe you should know that this report exists, and we’ll give you some hints as to what to look for there.

To view the report, you need to run only a single command. We recommend using the vertical result display:

```
mysql> SHOW ENGINE INNODB STATUS\G


*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
2021-05-31 12:21:05 139908633830976 INNODB MONITOR OUTP
=====================================
Per second averages calculated from the last 35 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 121 srv_active, 0 srv_shutdowr
...
----------
ROW OPERATIONS
----------
```

```
0 queries inside InnoDB, 0 queries in queue
2 read views open inside InnoDB
Process ID=55171, Main thread ID=139908126139968 , stat
Number of rows inserted 2946375, updated 87845, deletec
572.50 inserts/s, 1145.00 updates/s, 572.50 deletes/s,
Number of system rows inserted 109, updated 367, delete
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 re
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

The output, which we've truncated here, is presented broken down in sections. At first, it may seem intimidating, but over time you will come to appreciate the details. For now, we'll walk through a few sections that we believe provide information that will benefit operators of any experience level:

Transactions

This section provides information about the transactions of every session, including duration, current query, number of locks held, and information on lock waits. You can also find some data here on transaction visibility, but that's rarely required. Usually, you want to look at the transactions section to see the current state of transactions active within InnoDB. A sample record from this section looks like this:

```
---TRANSACTION 252288, ACTIVE (PREPARED) 0 sec
5 lock struct(s), heap size 1136, 3 row lock(s),
MySQL thread id 82, OS thread handle 139908634125
...query id 925076 localhost sbuser waiting for h
COMMIT
Trx read view will not see trx with id >= 252287,
```

This tells us that the transaction is currently waiting for a `COMMIT` to finish it holds three row locks, and it's pretty fast, likely to finish in under a second. Sometimes, you will see long transactions here: you should try to avoid those. InnoDB does not handle long transactions well, and even an idle transaction staying open for too long can cause a performance impact.

This section will also show you information on the current locking behavior if there are transactions waiting to obtain locks. Here's an example:

```
---TRANSACTION 414311, ACTIVE 4 sec starting inde
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row
MySQL thread id 84, OS thread handle 139908634125
...query id 2545483 localhost sbuser updating
UPDATE sbtest1 SET k=k+1 WHERE id=347110
Trx read view will not see trx with id >= 414310,
------- TRX HAS BEEN WAITING 4 SEC FOR THIS LOCK
RECORD LOCKS space id 333 page no 4787 n bits 144
...table `sysbench`.`sbtest1` trx id 414311 lock_
...rec but not gap waiting
Record lock, heap no 33
------------------
```

Unfortunately, the status report doesn't point at the lock holder directly, so you'll need to look for blocking transactions separately. Some information on that is available in Chapter 6. Usually, if you see one transaction active while others are waiting, it's a good indication that it's that active transaction that's holding the locks.

File I/O

This section contains information on the current I/O operations, as well as aggregated summaries over time. We discussed this in more detail in "InnoDB I/O and transaction metrics", but this is an additional way of checking whether InnoDB has pending data and log operations.

Buffer pool and memory

In this section InnoDB prints information about its buffer pool and memory usage. If you have multiple buffer pool instances configured, then this section will show both totals and per-instance breakdowns. There's a lot of information, including on the buffer pool size and the internal state of the buffer pool:

```
Total large memory allocated 274071552
Dictionary memory allocated 1377188
```

```
Buffer pool size   16384
Free buffers       1027
Database pages     14657
Old database pages 5390
Modified db pages  4168
```

These are also exposed as `Innodb_buffer_pool_%` variables.

Semaphores

This section includes information on internal InnoDB semaphores, or synchronization primitives. In basic terms, *semaphores* are special internal in-memory structures that allow multiple threads to operate without interfering with each other. You will rarely see anything of value in this section unless there's semaphore contention on your system. Usually that happens when InnoDB is put under extreme load, so every operation takes longer, and there are more chances to see active semaphore waits in the status output.

Investigation Methods

Having so many available metrics that need to be monitored, checked, and understood may cause your head to spin. You may notice that we haven't defined a single metric that has a definite range from good to bad. Is the locking bad? Could be, but it's also expected and normal. The same can be said about almost every aspect of MySQL, with the exception of server availability.

This problem is not unique to MySQL server monitoring, and in fact it's common for any complex system. There are a lot of metrics, complicated dependencies, and almost no strictly definable rules for whether something is good or bad. To solve this problem, we need some approach, some methodology that we can apply to abundant data to quickly and easily come to conclusions about current system performance.

Luckily, such methodologies already exist. In this section, we will briefly describe two of them and give ideas on how to apply them to monitoring MySQL and OS metrics.

The USE Method

The Utilization, Saturation, and Errors (USE) method, popularized by Brendan Gregg, is a general-purpose methodology that can be applied to any system. Though better suited for resources with well-defined performance characteristics like CPU or disk, it can also be applied to some parts of MySQL.

The USE method is best used by creating a checklist for each individual part of the system. First, we need to define what metrics to use to measure utilization, saturation, and errors. Some example checklists for Linux can be found on the USE homepage: Gregg's website.

Let's look at the three different categories for the example of disk I/O:

Utilization

For a disk subsystem, as we've laid out in "Disk", we might look at metrics for any:

- Overall storage bandwidth utilization
- Per-process storage bandwidth utilization
- Current IOPS (where the upper IOPS limit for the device is known)

Saturation

We mentioned saturation when talking about the `iostat` output and disk metrics. It's a measure of work that the resource cannot process and that is usually queued. For disk, this is expressed as the I/O request queue size (`avgqu-sz` in `iostat` output). On most systems, values >1 mean the disk is saturated, and thus some requests will be queued and waste time doing nothing.

Errors

For disk devices, this may mean I/O errors due to hardware degradation.

The problem with the USE method is that it's difficult to apply to a complex system as a whole. Take MySQL, for example: what metrics could we use to measure overall utilization, saturation, and errors? Some attempts can be made

to apply USE to MySQL as a whole, but it's much better suited to use on some isolated parts of the system.

Let's take a look at some possible MySQL applications:

USE method for client connections

A good fit for the USE method is making sense of the client connection metrics we discussed in "Basic Monitoring Recipes".

Utilization metrics include the `Threads_connected` and `Threads_running`.

Saturation can be defined arbitrarily based on `Threads_running`. For example, in an OLTP system that is read-heavy and doesn't generate a lot of I/O, a saturation point for `Threads_running` will likely be around the number of available CPU cores. In a system that is mostly I/O-bound, a starting point could be twice the available CPU cores, but it's much better to find what number of concurrently running threads start to saturate the storage subsystem.

Errors are measured by the `Aborted_clients` and `Aborted_connects` metrics. Once the value of the `Threads_connected` status variable becomes equal to that of the `max_connections` system variable, requests to create new connections will be declined, and clients will get errors. Connections can also fail for other reasons, and existing clients may be terminating their connections uncleanly without waiting for MySQL to respond.

USE method for transactions and locking

Another example of how USE can be applied is to look at the InnoDB locking-related metrics relative to the number of queries processed by the system.

Utilization can be measured using the synthetic QPS metric we defined earlier.

Saturation can be the fact that there's some number of lock waits occurring. Remember, by the USE definition, saturation is a measure of work that cannot be performed and has to wait in the queue. There's no queue here, but transactions are waiting for locks to be acquired.

Alternatively, the regular amount of lock waits can be multiplied twofold to construct an arbitrary threshold, or better yet, some experiments can be performed to find a number of lock waits relative to QPS that result in lock timeouts occurring.

Errors can be measured as the number of times sessions timed out waiting for locks or were terminated to resolve a deadlock situation. Unlike the previously mentioned metrics, these two are not exposed as global status variables, but can instead be found in the `information_schema.innodb_metrics` table under `lock_deadlocks` (number of deadlocks registered) and `lock_timeouts` (number of times lock waits timed out).

Determining the rate of errors from monitoring metrics alone can be difficult, so frequently just the US part of USE is used. As you can see, this method allows us to look at the system from a predefined point of view. Instead of analyzing every possible metric when there's an incident, an existing checklist can be reviewed, saving time and effort.

RED Method

The Rate, Errors, and Duration (RED) method was created to address the shortcomings of USE. The methodology is similar, but it's is easier to apply to complex systems and services.

The logical application of RED to MySQL is done by looking at query performance:

Rate

QPS of the database

Errors

Number or rate of errors and failing queries

Duration

Average query latency

One problem with RED in the context of this book and learning MySQL in general is that applying this method requires monitoring data that can't be

obtained just by reading status variables. Not every existing monitoring solution for MySQL can provide the necessary data, either, though you can see an example of how it's done in the blog post "RED Method for MySQL Performance Analyses" by Peter Zaitsev. One way to apply RED to MySQL or any other database system is by looking at the metrics from the application side instead of from the database side. Multiple monitoring solutions allow you to instrument (manually or automatically) your applications to capture data such as number of queries, rate of failures, and query latency. Just what's needed for RED!

You can, and should, use RED and USE together. In the article explaining RED, The RED Method: How to Instrument Your Services, its author Tom Wilkie mentions that "It's really just two different views on the same system."

One perhaps unexpected benefit of applying RED, USE, or any other method is that you do so before an incident happens. Thus, you are forced to understand what it is you monitor and measure and how that relates to what actually matters for your system and its users.

MySQL Monitoring Tools

Throughout this chapter we've been talking about metrics and monitoring methodologies, but we haven't mentioned a single actual tool that can turn those metrics into dashboards or alerts. In other words, we haven't talked about the actual monitoring tools and systems. The first reason for that is simple enough: we believe that in the beginning it's much more important for you to know *what* to monitor and *why*, rather than *how*. If we'd concentrated on the *how*, we could have spent this whole chapter talking about the peculiarities and differences of various monitoring tools. The second reason is that MySQL and OS metrics don't change often, but if you're reading this book in 2025, our choice of monitoring tools may already seem antiquated.

Nevertheless, as a starting point we've put together a list of notable popular open source monitoring tools that can be used to monitor MySQL availability and performance. We're not going to go too deep into the specifics of their setup or configuration, or into comparing them. We cannot also list every possible monitoring solution available: brief research shows us that almost anything that is a "monitoring tool" or "monitoring system" can monitor MySQL in some way. We're also not covering non–open source and non-free

monitoring solutions, with the one exception of the Oracle Enterprise Monitor. We hold nothing against such systems in general, and a lot of them are great. Most of them, though, have excellent documentation and support available, so you should be able to get familiar with them quickly.

The following monitoring systems will be mentioned here:

- Prometheus
- InfluxDB and TICK stack
- Zabbix
- Nagios Core
- Percona Monitoring and Management
- Oracle Enterprise Monitor

We'll start with Prometheus and InfluxDB and its TICK stack. Both of these systems are a modern take on monitoring, well suited to monitoring microservices and vast cloud deployments, but are also widely used as general-purpose monitoring solutions:

Prometheus

Born out of Google's internal monitoring system Borgmon, Prometheus is an extremely popular general-purpose monitoring and alerting system. At its core is a time-series database and a data-gathering engine based around a pull model. What that means is that it's the Prometheus server that actively gathers data from its monitoring targets.

Actual data gathering from Prometheus targets is performed by special programs called *exporters*. Exporters are purpose-built: there's a dedicated MySQL exporter, a PostgreSQL exporter, a basic OS metrics/node exporter, and many more. What these programs do is collect metrics from the system they are written to monitor and present those metrics in a format suitable for the Prometheus server to consume.

MySQL monitoring with Prometheus is done by running the `mysqld_exporter` program. Like most parts of Prometheus ecosystem, it's written in Go and is available for Linux, Windows, and many other operating systems, making it a good choice for a heterogeneous environment.

The MySQL exporter gathers all the metrics we've covered in this chapter (and many more!), and since it actively tries to get information from MySQL, it can also report on the MySQL server's availability. In addition to standard metrics, it is possible to supply custom queries that exporter will execute, turning their results into additional metrics.

Prometheus offers only very basic visualization capabilities, so the Grafana analytics and data visualization web application is usually added to the setup.

InfluxDB and the TICK stack

Built around the InfluxDB time-series database, TICK, standing for Telegraf, InfluxDB, Chronograf, and Kapacitor, is a complete time-series and monitoring platform. Comparing this to Prometheus, Telegraf takes the place of the exporters; it's a unified program that is capable of monitoring a multitude of targets. Unlike exporters, Telegraf actively pushes data to InfluxDB instead of data being pulled by the server. Chronograf is an administrative and data interface. Kapacitor is a data processing and alerting engine.

Where you had to install a dedicated exporter for MySQL, Telegraf is extended using plugins. The MySQL plugin is part of a standard bundle and provides a detailed overview of MySQL database metrics. Unfortunately, it is not capable of running arbitrary queries, so extensibility is limited. As a workaround, the `exec` plugin can be used. Telegraf is also a multiplatform program, and it does support Windows among other OSs.

The TICK stack is frequently used in part, with Grafana added to InfluxDB and Telegraf.

A common thread between Prometheus and TICK is that they are collections of building blocks allowing you to build your own monitoring solution. Neither of them offers any out-of-the-box recipes for dashboards, alerting, and so forth. They are powerful, but they may require some getting used to. Apart from that, they are very automation- and infrastructure-as-code-oriented. Prometheus especially, but TICK as well, provides a minimal GUI and was not initially conceived for data exploration and visualization. It's monitoring as in reacting to metric value changes by alerting, not by visually inspecting various metrics. Adding Grafana to the equation, especially with either home-

brewed or community dashboards for MySQL, makes visual inspection possible. Still, most of the configuration and setup will not be done in a GUI.

Both of these systems saw an influx in popularity in the mid- and late-2010s, with the shift to running a multitude of small servers compared to running a few large servers. That shift required some changes in monitoring approaches, and these systems became almost the de facto standard monitoring solutions for a lot of companies.

Next in our review are a couple of more "old-school" monitoring solutions:

Zabbix

A completely free and open source monitoring system first released in 2001, Zabbix is proven and powerful. It supports a wide range of monitored targets and advanced auto-discovery and alerting capabilities.

MySQL monitoring with Zabbix can be done by using plugins or with the official MySQL templates. Metrics coverage is quite good, with every recipe we defined available. However, both `mysqld_exporter` and Telegraf offer more data. The standard MySQL metrics Zabbix collects are sufficient to set up basic MySQL monitoring, but for deeper insight you will need to go custom or use some of the community templates and plugins.

The Zabbix agent is cross-platform, so you can monitor MySQL running on almost any OS.

While Zabbix offers quite powerful alerting, its visualization capabilities may feel slightly dated. It is possible to set up custom dashboards based on MySQL data, and it's also possible to use Zabbix as a data source for Grafana.

Zabbix is fully configurable through its GUI. The commercial offering includes various levels of support and consulting.

Nagios Core

Like Zabbix, Nagios is a veteran monitoring system, with its first release seeing light in 2002. Unlike other systems we've seen so far,

Nagios is an “open core” system. The Nagios Core distribution is free and open source, but there’s also a commercial Nagios system.

Monitoring of MySQL is set up by use of plugins. They should provide enough data to set up basic monitoring similar to that of the official Zabbix templates. It is possible to extend the collected metrics if needed.

Alerting, visualizations, and configuration are similar to Zabbix. One notable feature of Nagios is that at its peak of popularity it was forked multiple times. Some of the most popular Nagios forks are Icinga and Shinken. Check_MK was also initially a Nagios extension that eventually moved on to become its own commercial product.

Both Nagios and its forks and Zabbix can and are successfully used by many companies to monitor MySQL. Even though they may feel outdated in their architecture and data representation, they can get the job done. Their biggest problem is that the standard data they collect may feel limited compared with alternatives, and you’ll need to use community plugins and extensions. Percona used to maintain a set of monitoring plugins for Nagios, as well as for Zabbix, but has deprecated them and now concentrates on its own monitoring offering, Percona Monitoring and Management (PMM), which we’ll discuss shortly.

All the systems we’ve covered so far have one thing in common: they are general-purpose monitoring solutions, not tailor-made for database monitoring. It’s their power, and also their weakness. When it comes to monitoring and investigating deep database internals, you’ll often be forced to manually extend those systems’ capabilities. One feature, for example, that none of them have is storing individual query execution statistics. Technically, it’s possible to add that feature, but it may be cumbersome and problematic.

We will finish this section by looking at two database-oriented monitoring solutions, MySQL Enterprise Monitor from Oracle and Percona Monitoring and Management. As you’ll see, they are similar in the functionality they provide, and both are big improvements over nonspecialized monitoring systems:

MySQL Enterprise Monitor

Part of MySQL Enterprise Edition, Enterprise Monitor is a complete monitoring and management platform for MySQL databases.

In terms of monitoring, MySQL Enterprise Monitor extends the usual metrics gathered by monitoring systems by adding details on MySQL memory utilization, per-file I/O details, and a wide variety of dashboards based on InnoDB's status variables. The data is taken from MySQL itself without any agents involved. Theoretically, all the same data can be gathered and visualized by any other monitoring system, but here it's tightly packed with well-thought-out dashboards and categories.

Enterprise Monitor includes the Events subsystem, which is a set of predefined alerts. Adding to the database-specific features, the Enterprise Monitor includes a replication topology overview for regular and multisource replication, Group Replication, and NDB Cluster. Another feature is monitoring of backup execution status (for backups done with MySQL Enterprise Backup).

We mentioned that individual query execution statistics and query history are usually missing in the general-purpose monitoring systems. MySQL Enterprise Monitor includes a Query Analyzer that provides insight into the history of queries executed over time, as well as statistics gathered about the queries. It's possible to view information like the average numbers of rows read and returned and a duration distribution, and even see the execution plans of the queries.

Enterprise Monitor is a good database monitoring system. Its biggest downside, really, is that it's only available in the Enterprise Edition of MySQL. Unfortunately, most MySQL installations cannot benefit from the Enterprise Monitor and the level of insight it provides into the database and OS metrics. It's also not suitable for monitoring anything apart from MySQL, the queries it's executing, and the OS it's running on, and MySQL monitoring is limited in scope to Oracle's products.

There's a 30-day trial of MySQL Enterprise Edition available, which includes the Enterprise Monitor, and Oracle also maintains a list of visual demos of the system.

Percona Monitoring and Management

Percona's monitoring solution, PMM, is similar in functionality to Oracle's Enterprise Monitor, but is fully free and open source. Intended to be a "single pane of glass," it tries to provide deep insight into MySQL and OS performance and can also be used to monitor MongoDB and PostgreSQL databases.

PMM is built on top of existing open source components like the already reviewed Prometheus and its exporters, and Grafana. Percona maintains forks of the database exporters it uses, including the one for MySQL, and adds functionality and metrics that were lacking in the original versions. In addition to that, PMM hides complexity usually associated with deploying and configuring those tools and instead provides its own bundled package and configuration interface.

Like Enterprise Monitor, PMM offers a selection of dashboards visualizing pretty much every aspect of MySQL's and InnoDB's operation, as well as giving a lot of details on the underlying OS state. This has been extended to include technologies like PXC/Galera, discussed in Chapter 13, and ProxySQL, discussed in Chapter 15. As PMM uses Prometheus and exporters, it's possible to extend the range of monitored databases by adding external exporters. In addition to that, PMM supports DBaaS systems like RDS and CloudSQL.

PMM ships with a custom application called Query Analytics (QAN), which is a query monitoring and metrics system. Like the Enterprise Monitor's Query Analyzer, QAN shows the overall history of queries executed in a given system, as well as information about the individual queries. That includes a history of the number of executions of the query over time, rows read and sent, locking, and temporary tables created, among other things. QAN allows you to view the query plan and structures of the involved tables.

The *Management* part of PMM for now exists only in its name, as at the time of writing it is purely a monitoring system. PMM supports alerting through Prometheus's standard AlertManager or through the use of internal templates.

One significant problem with PMM is that out of the box it only supports targets running on Linux. Since Prometheus exporters are cross-platform, you can add Windows (or other OS) targets to PMM,

but you won't be able to utilize some of the benefits of the tool, like simplified exporter configuration and bundled installation of the client software.

Both authors of this book are currently employed by Percona, so you may be tempted to dismiss our description of PMM as an advertisement. However, we've tried to give a fair overview of a few monitoring systems, and we don't claim that PMM is perfect. If your company is already using the Enterprise version of MySQL, then you should absolutely first see what MySQL Enterprise Monitor has to offer.

Before we close this section, we want to note that in many cases the actual monitoring system you use doesn't matter much. Every system we've mentioned provides MySQL availability monitoring, as well as some level of insight into the internal metrics—enough for the selection of recipes we gave earlier. Especially while you're learning your way around MySQL, and perhaps starting to operate its installations in production, you should try to leverage existing monitoring infrastructure. Rushing to change everything to the best often leads to unsatisfactory results. As you get more experienced, you will see more and more data missing in the tools you have and will be able to make an informed decision on which new tool to choose.

Incident/Diagnostic and Manual Data Collection

Sometimes, you may not have a monitoring system set up for a database, or you may not trust it to contain all the information you might need to investigate some issue. Or you may have a DBaaS instance running and want to get more data than your cloud provider gives you. In such situations, manual data collection can be a viable option in the short term to quickly get some data out of the system. We will show you few tools you can use to do just that: quickly gather a lot of diagnostic information from any MySQL instance.

The following sections are short, useful recipes that you can take away and use in your day-to-day work with MySQL databases.

Gathering System Status Variable Values Periodically

In "Status Variables" and "Basic Monitoring Recipes" we talked a lot about looking at how different status variables' values change over time. Every monitoring tool mentioned in the previous section does that to accumulate data, which is then used for plots and alerting. The same sampling of status variables can be performed manually if you want to look at raw data or simply sample at an interval lower than your monitoring system uses.

You could write a simple script that runs the MySQL monitor in a loop, but the better approach is to use the built-in `mysqladmin` utility. This program can be used to perform a wide range of administrative operations on a running MySQL server, though we should note that every one of those can also be done through the regular `mysql`. `mysqladmin` can, however, be used to sample global status variables easily, which is exactly how we're going to use it here.

`mysqladmin` includes two status outputs: regular and extended. The regular one is less informative:

```
$ mysqladmin status
Uptime: 176190  Threads: 5  Questions: 5287160 ...
... Slow queries: 5114814  Opens: 761  Flush tables: 3
... Open tables: 671  Queries per second avg: 30.008
```

The extended output will be familiar to you at this point. It's the same as the output of `SHOW GLOBAL STATUS`:

```
$ mysqladmin extended-status
```

```
+-------------------------------------------------+------
| Variable_name                                   | Value
+-------------------------------------------------+------
| Aborted_clients                                 | 2
| Aborted_connects                                | 30
| ...
| Uptime                                          | 17630
| Uptime_since_flush_status                       | 32141
| validate_password.dictionary_file_last_parsed | 2021-
```

```
| validate_password.dictionary_file_words_count | 0
+-----------------------------------------------+------
```

Conveniently, `mysqladmin` is capable of repeating the commands it runs at a given interval a given number of times. For example, the following command will cause `mysqladmin` to print status variable values every second for a minute (`ext` is a shorthand for *extended-status*):

```
$ mysqladmin -i1 -c60 ext
```

By redirecting its output to a file, you can get a minute-long sample of database metric changes. Text files are not as nice to work with as proper monitoring systems, but again, this is usually done under special circumstances. For a long time, gathering information about MySQL like this with `mysqladmin` was standard practice, so there's a tool called `pt-mext` that can turn plain `SHOW GLOBAL STATUS` outputs into a format better suited for consumption by humans. Unfortunately, the tool is available only on Linux. Here's an example of its output:

```
$ pt-mext -r -- cat mysqladmin.output | grep Bytes_sent
Bytes_sent 10836285314 15120 15120 31080 15120 15120 31
```

The initial large number is the status variable value at the first sample, and values after that represent the change to the initial number. If the value were to decrease, a negative number would be shown.

Using pt-stalk to Collect MySQL and OS Metrics

`pt-stalk` is a part of Percona Toolkit and is normally run alongside MySQL and used to continuously check for some specified condition. Once that condition is met—for example, the value of `Threads_running` is larger than 15— `pt-stalk` triggers a data collection routine gathering extensive information on MySQL and the operating system. However, it is possible to utilize the data collection part without actually stalking the MySQL server. Although it's not a correct way to use `pt-stalk` , it's a useful method to quickly glance at an unknown server or try to gather as much information as possible on a misbehaving server.

`pt-stalk`, like other tools in the Percona Toolkit, is available only for Linux, even though the target MySQL server can run on any OS. The basic invocation of `pt-stalk` to achieve that is simple:

```
$ sudo pt-stalk --no-stalk --iterations=2 --sleep=30 \
--dest="/tmp/collected_data" \
-- --user=root --password=<root password>;
```

The utility will run two data collection rounds, each of which will span a minute, and will sleep for 30 seconds between them. In case you don't need OS information, or can't get it because your target is a DBaaS instance, you can use the `--mysql-only` flag:

```
$ sudo pt-stalk --no-stalk --iterations=2 --sleep=30 \
--mysql-only --dest="/tmp/collected_data" \
-- --user=root --password=<root password> \
--host=<mysql host> --port=<mysql port>;
```

Here is the list of files created by a single collection round. We've omitted the OS-related files deliberately, but there are quite a lot of them:

```
2021_04_15_04_33_44-innodbstatus1
2021_04_15_04_33_44-innodbstatus2
2021_04_15_04_33_44-log_error
2021_04_15_04_33_44-mutex-status1
2021_04_15_04_33_44-mutex-status2
2021_04_15_04_33_44-mysqladmin
2021_04_15_04_33_44-opentables1
2021_04_15_04_33_44-opentables2
2021_04_15_04_33_44-processlist
2021_04_15_04_33_44-slave-status
2021_04_15_04_33_44-transactions
2021_04_15_04_33_44-trigger
2021_04_15_04_33_44-variables
```

Extended Manual Data Collection

`pt-stalk` is not always available, and it doesn't run on all platforms. Sometimes, you may also want to add to (or remove some of) the data it

gathers. You can use the `mysqladmin` command introduced earlier to gather a bit more data and wrap it all up in a simple script. A version of this script is frequently used by authors of this book in their daily work.

This script, which should run on any Linux or Unix-like system, will execute continuously either until terminated or until the */tmp/exit-flag* file is found to be present. You can run `touch /tmp/exit-flag` to gracefully finish the execution of this script. We recommend putting it into a file and running it through `nohup … &` or executing it within a `screen` or `tmux` session. If you're unfamiliar with the terms we've just mentioned, they are all ways to make sure a script continues to execute when your session disconnects. Here's the script:

```
DATADEST="/tmp/collected_data";
MYSQL="mysql --host=127.0.0.1  -uroot -proot";
MYSQLADMIN="mysqladmin  --host=127.0.0.1 -uroot -proot"
[ -d "$DATADEST" ] || mkdir $DATADEST;
while true; do {
  [ -f /tmp/exit-flag ] \
    && echo "exiting loop (/tmp/exit-flag found)" \
    && break;
  d=$(date +%F_%T |tr ":" "-");
  $MYSQL -e "SHOW ENGINE INNODB STATUS\G" > $DATADEST/$
  $MYSQL -e "SHOW ENGINE INNODB MUTEX;" > $DATADEST/$d-
  $MYSQL -e "SHOW FULL PROCESSLIST\G" > $DATADEST/$d-pr
  $MYSQLADMIN -i1 -c15 ext > $DATADEST/$d-mysqladmin ;
} done;
$MYSQL -e "SHOW GLOBAL VARIABLES;" > $DATADEST/$d-varia
```

WARNING

Do not forget to adjust user credentials in these scripts before executing. Note that output files scripts produce may take a substantial amount of disk space. Always test any scripts in a safe environment before executing on critical servers.

We've also created a Windows version of the same script written in PowerShell. It behaves exactly in the same way as the previous script and will terminate on its own as soon as the file *C:\tmp\exit-flag* is found:

```
$mysqlbin='C:\Program Files\MySQL\MySQL Server 8.0\bin\
$mysqladminbin='C:\Program Files\MySQL\MySQL Server 8.0

$user='root'
$password='root'
$mysqlhost='127.0.0.1'

$destination='C:\tmp\collected_data'
$stopfile='C:\tmp\exit-flag'

if (-Not (Test-Path -Path '$destination')) {
  mkdir -p '$destination'
}

Start-Process -NoNewWindow $mysqlbin -ArgumentList `
  '-h$mysqlhost','-u$user','-p$password','-e 'SHOW GLOB
  -RedirectStandardOutput '$destination\variables'

while(1) {
  if (Test-Path -Path '$stopfile') {
    echo 'Found exit monitor file, aborting'
    break;
  }
  $d=(Get-Date -Format 'yyyy-MM-d_HHmmss')
  Start-Process -NoNewWindow $mysqlbin -ArgumentList `
    '-h$mysqlhost','-u$user','-p$password','-e 'SHOW EN
    -RedirectStandardOutput '$destination\$d-innodbstat
  Start-Process -NoNewWindow $mysqlbin -ArgumentList `
    '-h$mysqlhost','-u$user','-p$password','-e 'SHOW EN
    -RedirectStandardOutput '$destination\$d-innodbmute
  Start-Process -NoNewWindow $mysqlbin -ArgumentList `
    '-h$mysqlhost','-u$user','-p$password','-e 'SHOW FU
    -RedirectStandardOutput '$destination\$d-processlis
  & $mysqladminbin '-h$mysqlhost' -u'$user' -p'$passwor
    -i1 -c15 ext > '$destination\$d-mysqladmin';
}
```

You should remember that script-based data collection is not a substitute for proper monitoring. It has its uses, which we described at the start of this section, but it should always be an addition to what you already have, not the only way to look at MySQL metrics.

By now, after reading through this chapter, you should have a pretty good idea of how to approach MySQL monitoring. Remember that issues, incidents, and outages will happen—they are unavoidable. With proper monitoring, however, you can make sure that the same issue doesn't happen twice, as you'll be able to find the root cause after the first occurrence. Of course, you'll also be able to avoid some issues by changing your system to fix the problems revealed by your MySQL monitoring efforts.

We'll leave you with a closing thought: perfect monitoring is unattainable, but even pretty basic monitoring is better than no monitoring at all.