Chapter 10. Backups and Recovery

The most important task for any DBA is backing up the data. Correct and tested backup and recovery procedures can save a company and thus a job. Mistakes happen, disasters happen, and errors happen. MySQL is a robust piece of software, but it’s not completely free of bugs or crashes. Thus, it is crucial to understand why and how to perform backups.

Apart from preserving database contents, most backup methods can also be used for another important purpose: copying the contents of the database between separate systems. Though probably not as important as saving the day when corruption happens, this copying is a routine operation for the vast majority of database operators. Developers will often need to use downstream environments, which should be similar to production. QA staff may need a volatile environment with a lifespan of an hour. Analytics may be run on a dedicated host. Some of these tasks can be solved by replication, but any replica starts from a restored backup.

This chapter first briefly reviews two major types of backups and discusses their fundamental properties. It then looks at some of the tools available in the MySQL world for the purpose of backup and recovery. Covering each and every tool and their parameters would be beyond the scope of this book, but by the end of the chapter you should know your way around backing up and recovering MySQL data. We’ll also explore some basic data transfer scenarios. Finally, the chapter outlines a robust backup architecture that you can use as a foundation for your work.

An overview of what we think is a good backup strategy is given in “Database Backup Strategy Primer”. We believe it’s important to understand the tools and moving parts before deciding on the strategy, and therefore that section comes last.

Physical and Logical Backups

Broadly speaking, most if not all of the backup tools fit into just two wide categories: logical and physical. *Logical* backups operate on the internal structures: databases (schemas), tables, views, users, and other objects. *Physical* backups are concerned with the OS-side representation of the database structures: data files, transaction journals, and so on.

It might be easier to explain with an example. Imagine backing up a single MyISAM table in MySQL database. As you will see later in this chapter, the InnoDB storage engine is more complex to back up correctly. Knowing that MyISAM is not transactional and that there are no ongoing writes to this table, we may go ahead and copy the files related to it. In doing so, we create a physical backup of the table. We could instead go ahead and run `SELECT *` and `SHOW CREATE TABLE` statements against this table and preserve the outputs of those statements somewhere. That's a very basic form of a logical backup. Of course, these are just simple examples, and in reality the process of obtaining both types of backup will be more complex and nuanced. The conceptual differences between these imaginary backups can, however, be transferred and applied to any logical and physical backups.

Logical Backups

Logical backups are concerned with the *actual data*, and not its *physical representation*. As you've already seen, such backups don't copy any existing database files and instead rely on queries or other means to obtain needed database contents. The result is usually some textual representation, though that's not granted, and a logical backup's output may well be binary-encoded. Let's see some more examples of how such backups might look and then discuss their properties.

Here are some examples of logical backups:

- Table data queried and saved into an external *.csv* file using the `SELECT ... INTO OUTFILE` statement that we cover in "Writing Data into Comma-Delimited Files".
- A table or any other object's definition saved as a SQL statement.
- One or more `INSERT` SQL statements that, run against a database and an empty table, would populate that table up to a preserved state.

- A recording of all statements ever run that touched a particular table or database and modified data or schema objects. By this we mean DML and DDL commands; you should be familiar with both types, covered in Chapters 3 and 4

NOTE

That last example actually represents how both replication and point-in-time recovery work in MySQL. We tackle those topics later, and you'll see that the term *logical* doesn't just apply to backups.

Recovery of a logical backup is usually done by executing one or more SQL statements. Continuing with our earlier examples, let's review the options for recovery:

- Data from a *.csv* file can be loaded into a table using the `LOAD DATA INFILE` command.
- The table can be created or re-created by running a DDL SQL statement.
- `INSERT` SQL statements can be executed using the `mysql` CLI or any other client.
- A replay of all the statements run in a database will restore it to its state after the last statement.

Logical backups have some interesting properties that make them extremely useful in some situations. More often than not, a logical backup is some form of text file, consisting mostly of SQL statements. That is not necessary, however, and is not a defining property (albeit useful one). The process of creating logical backups also usually involves the execution of some queries. These are important features because they allow for a great degree of flexibility and portability.

Logical backups are flexible because they make it very easy to back up a part of a database. For example, you can back up schema objects without their contents or easily back up only a few of the database's tables. You can even back up part of a table's data, which is usually impossible with physical backups. Once the backup file is ready, there are tools you can use to review and modify it either manually or automatically, which is something not easily done with copies of database files.

Portability comes from the fact that logical backups can be loaded easily into different versions of MySQL running on different operating systems and architectures. With some modification, you can actually load logical backups taken from one RDBMS into an absolutely different one. Most database migration tools use logical replication internally due to this fact. This property also makes this backup type suitable for backing up cloud-managed databases off-site, and for migrations between them.

Another interesting property of logical backups is that they are effective in combating *corruption*—that is, physical corruption of a physical data file. Errors in data can still be introduced, for example, by bugs in software or by gradual degradation of storage media. The topic of corruption and its counterpart, integrity, is very wide, but this brief explanation should suffice for now.

Once a data file becomes corrupted, a database might not be able to read data from it and serve the queries. Since corruption tends to happen silently, you might not know when it occurred. However, if a logical backup was generated without error, that means it's sound and has good data. Corruption could happen in a *secondary index* (any non-primary index; see Chapter 4, *Working with Database Structures* for more details), so a logical backup doing a full table scan might generate normally and not face an error. In short, a logical backup can both help you detect corruption early (as it scans all tables) and help you save the data (as the last successful logical backup will have a sound copy).

The inherent problem with all logical backups comes from the fact that they are created and restored by executing SQL statements against a running database system. While that allows for flexibility and portability, it also means that these backups result in load on the database and are generally quite slow. DBAs always frown when someone runs a query that reads all the data from a table indiscriminately, and that's exactly what logical backup tools usually do. Similarly, the restore operation for a logical backup usually results in the interpreting and running of each statement as if it came from a regular client. This doesn't mean that logical backups are bad or shouldn't be used, but it's a trade-off that must be remembered.

Physical Backups

Whereas logical backups are all about data as in database contents, physical backups are all about data as in operating system files and internal RDBMS workings. Remember, in the example with a MyISAM table being backed up, a physical backup was a copy of the files representing that table. Most of the backups and tools of this type are concerned with copying and transferring all or parts of database files.

Some examples of physical backups include the following:

- A *cold* database directory copy, meaning it's done when the database is shut down (as opposed to a *hot* copy, done while the database is running).
- A storage snapshot of volumes and filesystems used by database.
- A copy of table data files.
- A stream of changes to database data files of some form. Most RDBMSs use a stream like this for crash recovery, and sometimes for replication; InnoDB's redo log is a similar concept.

Recovery of a physical backup is usually done by copying back the files and making them consistent. Let's review the recovery options for the previous examples:

- A cold copy can be moved to a desired location or server and then used as a data directory by a MySQL instance, old or new.
- A snapshot can be restored in place or on another volume and then used by MySQL.
- Table files can be put in place of existing ones.
- A replay of the changes stream against the data files will recover their state to the last point in time.

Of these, the simplest physical backup that can be performed is a cold database directory backup. Yes, it's simple and basic, but it's a very powerful tool.

Physical backups, unlike logical ones, are very rigid, giving little leeway in terms of control over what can be backed up and where the backup can be used. Generally speaking, most physical backups can only be used to restore the exact same state of a database or a table. Usually, these backups also put constraints on the target database software version and operating system. With some work, you can restore a logical backup from MySQL to PostgreSQL. However, a cold copy of the MySQL data directory done on Linux may not work if restored on Windows. Also, you cannot take a physical backup if you don't have physical access to the database server. This means that performing such a backup on a managed database in the cloud is impossible: the vendor might be performing physical backups in the background, but you may not have a way to get them out.

Since a physical backup is by nature a copy of all or a subset of the original backup pages, any corruption present in the original will be included in the backup. It's important to remember that, because this property makes physical backups ill-suited for combating corruption.

You may wonder why you would use such a seemingly inconvenient way of backing up. The reason is that physical backups are fast. Operating on the OS or even storage level, physical backup methods are sometimes the only possible way to actually back up a database. By way of example, a storage snapshot of a multiterabyte volume might take a few seconds or minutes, whereas querying and streaming that data for a logical backup might take hours or days. The same goes for recovery.

Overview of Logical and Physical Backups

We've now covered the two categories of backups and are ready to start exploring the actual tools used for these backups in the MySQL world. Before we do that, though, let's summarize the differences between logical and physical backups and take a quick look at the properties of the tools used to create them.

Properties of logical backups:

- Contain a description and the contents of the logical structures
- Are human-readable and editable
- Are relatively slow to take and restore

Logical backup tools are:

- Very flexible, allowing you to rename objects, combine separate sources, perform partial restores, and more
- Not usually bound to a specific database version or platform

- Able to extract data from corrupted tables and safeguard from corruption
- Suitable for backing up remote databases (for example, in the cloud)

Properties of physical backups:

- Are byte-by-byte copies of parts of data files, or entire filesystems/volumes
- Are fast to take and restore
- Offer little flexibility and will always result in the same structure upon restore
- Can include corrupted pages

Physical backup tools are:

- Cumbersome to operate
- Usually don't allow for an easy cross-platform or even cross-version portability
- Cannot back up remote databases without OS access

TIP

These are not conflicting approaches. In fact, a generally good idea is to perform both types of backups on a regular basis. They serve different purposes and satisfy different requirements.

Replication as a Backup Tool

Replication is a very wide topic that upcoming chapters cover in detail. In this section, we briefly discuss how replication relates to the concept of backing up and recovering a database.

In brief, replication is not a substitute for taking backups. The specifics of replications are such that they result in a full or partial copy of a target database. This lets you use replication in a lot of, but not all, possible failure scenarios involving MySQL. Let's review two examples. They will be helpful later in the chapter as well.

NOTE

In the MySQL world, replication is a type of logical backup. That's because it's based on transferring logical SQL statements.

Infrastructure Failure

Infrastructure is prone to failure: drives go bad, power goes out, fires happen. Almost no system can provide 100% uptime, and only vastly distributed ones can even get close. What that means is that eventually *any* database will crash due to its host server failing. In a good case, a restart might be enough to recover. In a bad case, part or all of the data may be gone.

Restoring and recovering a backup is by no means an instantaneous operation. In a replicated environment, a special operation called *switchover* can be performed to put a replica in place of the failed database. In many cases, switchover saves a lot of time and allows for work on a failed system to proceed without too much rush.

Imagine a setup with two identical servers running MySQL. One is a dedicated primary, which receives all the connections and serves all the queries. The other one is a replica. There's a mechanism to redirect connections to the replica, with switchover resulting in 5 minutes of downtime.

One day, a hard disk drive goes bad in the primary server. It's a simple server, so that alone results in a crash and downtime. Monitoring catches the issue, and the DBA immediately understands that to restore the database on that server, they'll need to install a new disk and then restore and recover the recent backup. The whole operation will take a couple of hours.

Switching over to a replica is a good idea in this case, because it saves a lot of valuable uptime.

Deployment Bug

Software bugs are a fact of life that has to be accepted. The more complex the system, the higher the possible incidence of logical errors. While we all strive to limit and reduce bugs, we must understand that they will happen and plan accordingly.

Imagine that a new version of an application is released that includes a database migration script. Even though both the new version and the script were tested in downstream environments, there's a bug. Migration irrecoverably corrupts all customers' last names that have "special" non-ASCII symbols. The corruption is silent, since the script finishes successfully, and the issue is noticed only a week later by an angry customer, whose name is now incorrect.

Even though there's a replica of the production database, it has the same data and the same logical corruption. Switching over to the replica *won't* help in this case, and a backup taken prior to the migration must be restored to obtain a list of correct last names.

NOTE

Delayed replicas can protect you in such situations, but the longer the delay, the less practical it is to operate such a replica. You can create a replica with a delay of a week, but you may need data from an hour ago. Usually, replica delays are measured in minutes and hours.

The two failure scenarios just discussed cover two distinct domains: physical and logical. Replication is a good fit for protection in case of physical issues, whereas it provides no (or little) protection from logical issues. Replication is a useful tool, but it's no substitute for backups.

The mysqldump Program

Possibly the simplest way to back up a database online is to dump its contents as SQL statements. This is the paramount logical backup type. *Dumping* in computing usually means outputting the contents of some system or its parts, and the result is a *dump*. In the database world, a dump is usually a logical backup, and dumping is the action of obtaining such a backup. Restoring the backup involves applying the statements to the database. You can generate dumps manually using, for example, `SHOW CREATE TABLE` and some

CONCAT operations to get INSERT statements from data rows in the tables, like this:

```
mysql> SHOW CREATE TABLE sakila.actor\G
```

```
*************************** 1. row *******************
       Table: actor
Create Table: CREATE TABLE `actor` (
  `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIME
        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`actor_id`),
  KEY `idx_actor_last_name` (`last_name`)
) ENGINE=InnoDB AUTO_INCREMENT=201 DEFAULT CHARSET=utf8
        COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

```
mysql> SELECT CONCAT("INSERT INTO actor VALUES",
    -> "(",actor_id,",'",first_name,"','",
    -> last_name,"','",last_update,"');")
    -> AS insert_statement FROM actor LIMIT 1\G
```

```
*************************** 1. row *******************
insert_statement: INSERT INTO actor VALUES
(1,'PENELOPE','GUINESS','2006-02-15 04:34:33');
1 row in set (0.00 sec)
```

That, however, becomes extremely impractical very fast. Moreover, there are more things to consider: *order of statements*, so that upon restore an INSERT doesn't run before the table is created, and *ownership* and *consistency*. Even though generating logical backups manually is good for understanding, it is tedious and error-prone. Fortunately, MySQL is bundled with a powerful logical backup tool called mysqldump that hides most of the complexity.

The mysqldump program bundled with MySQL allows you to produce dumps from running database instances. The output of mysqldump is a number of SQL statements that can later be applied to the same or another instance of MySQL. `mysqldump` is a cross-platform tool, available on all the operating systems on which the MySQL server itself is available. As the resulting backup file is just a lot of text, it's also platform-independent.

The command-line arguments to `mysqldump` are numerous, so it is wise to review the MySQL Reference Manual before jumping into using the tool. However, the most basic scenario requires just one argument: the target database name.

TIP

We recommend that you set up a `client` login path following the instructions in "Login Path Configuration File" to the `root` user and password. You then won't need to specify an account and give its credentials to any of the commands we show in this chapter.

In the following example, `mysqldump` is invoked without output redirection, and the tool will print all the statements to standard output:

```
$ mysqldump sakila
```

```
...
--
-- Table structure for table `actor`
--

DROP TABLE IF EXISTS `actor`;
/*!40101 SET @saved_cs_client     = @@character_set_cli
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `actor` (
  `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIME
        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`actor_id`),
  KEY `idx_actor_last_name` (`last_name`)
) ENGINE=InnoDB AUTO_INCREMENT=201 DEFAULT CHARSET=utf8
        COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */
```

```
--
-- Dumping data for table `actor`
--

LOCK TABLES `actor` WRITE;
/*!40000 ALTER TABLE `actor` DISABLE KEYS */;
INSERT INTO `actor` VALUES
(1,'PENELOPE','GUINESS','2006-02-15 01:34:33'),
(2,'NICK','WAHLBERG','2006-02-15 01:34:33'),
...
(200,'THORA','TEMPLE','2006-02-15 01:34:33');
/*!40000 ALTER TABLE `actor` ENABLE KEYS */;
UNLOCK TABLES;
...
```

NOTE

The outputs of `mysqldump` are lengthy and ill-suited for printing in books. Here and elsewhere, the outputs are truncated to include only the lines we're interested in.

You may notice that this output is more nuanced than you might expect. For example, there's a `DROP TABLE IF EXISTS` statement, which prevents an error for the following `CREATE TABLE` command when the table already exists on the target. The `LOCK` and `UNLOCK TABLES` statements will improve data insertion performance, and so on.

Speaking of schema structure, it is possible to generate a dump that has no data. This can be useful to create a logical clone of the database, for example, for a development environment. Flexibility like this is one of the key features of logical backups and `mysqldump`:

```
$ mysqldump --no-data sakila
```

```
...
--
-- Table structure for table `actor`
--

DROP TABLE IF EXISTS `actor`;
/*!40101 SET @saved_cs_client     = @@character_set_cli
```

```
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `actor` (
  `actor_id` smallint unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIME
        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`actor_id`),
  KEY `idx_actor_last_name` (`last_name`)
) ENGINE=InnoDB AUTO_INCREMENT=201 DEFAULT CHARSET=utf8
        COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */

--
-- Temporary view structure for view `actor_info`
--
...
```

It’s also possible to create a dump of a single table in a database. In the next example, `sakila` is the database and `category` is the target table:

```
$ mysqldump sakila category
```

Turning the flexibility up a notch, you can dump just a few rows from a table by specifying the `--where` or `-w` argument. As the name suggests, the syntax is the same as for the `WHERE` clause in a SQL statement:

```
$ mysqldump sakila actor --where="actor_id > 195"
```

```
...
--
-- Table structure for table `actor`
--

DROP TABLE IF EXISTS `actor`;
CREATE TABLE `actor` (
...

--
-- Dumping data for table `actor`
--
-- WHERE:  actor_id > 195
```

```
LOCK TABLES `actor` WRITE;
/*!40000 ALTER TABLE `actor` DISABLE KEYS */;
INSERT INTO `actor` VALUES
(196,'BELA','WALKEN','2006-02-15 09:34:33'),
(197,'REESE','WEST','2006-02-15 09:34:33'),
(198,'MARY','KEITEL','2006-02-15 09:34:33'),
(199,'JULIA','FAWCETT','2006-02-15 09:34:33'),
(200,'THORA','TEMPLE','2006-02-15 09:34:33');
/*!40000 ALTER TABLE `actor` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
```

The examples covered so far have only covered dumping all or part of a single database: `sakila`. Sometimes it's necessary to output every database, every object, and even every user. `mysqldump` is capable of that. The following command will effectively create a full and complete logical backup of a database instance:

```
$ mysqldump --all-databases --triggers --routines --eve
```

Triggers are dumped by default, so this option won't appear in future command outputs. In the event you don't want to dump triggers, you can use `--no-triggers`.

There are a couple of problems with this command, however. First, even though we have redirected the output of the command to a file, the resulting file can be huge. Fortunately, its contents are likely going to be well suited for compression, though this depends on the actual data. Regardless, it's a good idea to compress the output:

```
$ mysqldump --all-databases \
--routines --events | gzip > dump.sql.gz
```

On Windows, compressing output through a pipe is difficult, so just compress the *dump.sql* file produced by running the previous command. On a system that is CPU-choked, like the little VM we're using here, compression might add significant time to the backup process. That's a trade-off that will have to be weighted for your particular system:

```
$ time mysqldump --all-databases \
--routines --events > dump.sql


real    0m24.608s
user    0m15.201s
sys     0m2.691s


$ time mysqldump --all-databases \
--routines --events | gzip > dump.sql.gz


real    2m2.769s
user    2m4.400s
sys     0m3.115s


$ ls -lh dump.sql


-rw... 2.0G ... dump.sql
-rw... 794M ... dump.sql.gz
```

The second problem is that to ensure consistency, locks will be placed on tables, preventing writes while a database is being dumped (writes to other databases can continue). This is bad both for performance and backup consistency. The resulting dump is consistent only within the database, not across the whole instance. This default behavior is necessary because some of the storage engines that MySQL uses are nontransactional (mainly the older MyISAM). The default InnoDB storage engine, on the other hand, has a multiversion concurrency control (MVCC) model that allows maintenance of a *read snapshot*. We covered different storage engines in more depth in "Alternative Storage Engines", and locking in Chapter 6.

Utilizing InnoDB's transaction capabilities is possible by passing the `--single-transaction` command-line argument to `mysqldump`. However, that removes table locking, thus making nontransactional tables prone to inconsistencies during the dump. If your system uses, for example, both InnoDB and MyISAM tables, it may be necessary to dump them separately, if no interruption of writes and consistency are required.

NOTE

Although `--single-transaction` ensures that writes can continue while `mysqldump` is running, there are still some caveats: DDL statements that are run concurrently might cause inconsistencies, and long-running transactions, such as one initiated by `mysqldump`, can have a negative impact on the overall instance performance.

The basic command to make a dump of a system using mainly InnoDB tables, which guarantees limited impact on concurrent writes, is as follows:

```
$ mysqldump --single-transaction --all-databases \
--routines --events | gzip > dump.sql.gz
```

In the real world, you will probably have some more arguments to specify connection options. You might also script around the `mysqldump` statement to catch any issues and notify you if anything went wrong.

Dumping with `--all-databases` includes internal MySQL databases such as `mysql`, `sys`, and `information_schema`. That information is not always needed to restore your data and might cause problems when restoring into an instance that already has some databases. However, you should remember that MySQL user details will only be dumped as part of the `mysql` database.

In general, using `mysqldump` and the logical backups it produces allows for the following:

- Easy transfer of the data between environments.
- Editing of the data in place both by humans and programs. For example, you can delete personal or unnecessary data from the dump.
- Finding certain data file corruptions.
- Transfer of the data between major database versions, different platforms, and even databases.

Bootstrapping Replication with mysqldump

The `mysqldump` program can be used to create a replica instance either empty or with data. To facilitate that, multiple command-line arguments are available. For example, when `--master-data` is specified, the resulting output will contain a SQL statement (`CHANGE MASTER TO`) that will set replication coordinates correctly on the target instance. When replication is later started using these coordinates on the target instance, there will be no gaps in data. In a GTID-based replication topology, `--set-gtid-purged` can be used to achieve the same result. However, `mysqldump` will detect that `gtid_mode=ON` and include the necessary output even without any additional command-line argument.

An example of setting up replication with `mysqldump` is provided in "Creating a Replica Using mysqldump".

Loading Data from a SQL Dump File

When performing a backup, it's always important to keep in mind that you're doing that to be able to later restore the data. With logical backups, the restoration process is as simple as *piping* contents of the backup file to the `mysql` CLI. As discussed earlier, the fact that MySQL has to be up for a logical backup restore makes for both good and bad consequences:

- You can restore a single object while other parts of your system are working normally, which is a plus.
- The process of restoration is inefficient and will load a system just like any regular client would if it decided to insert a large amount of data. That's a minus.

Let's take a look at a simple example with a single database backup and restore. As we've seen before, `mysqldump` will include the necessary `DROP` statements into the dump, so even if the objects are present, they will be successfully restored:

```
$ mysqldump sakila > /tmp/sakila.sql
$ mysql -e "CREATE DATABASE sakila_mod"
$ mysql sakila_mod < /tmp/sakila.sql
$ mysql sakila_mod -e "SHOW TABLES"
```

```
+----------------------------+
| Tables_in_sakila_mod       |
+----------------------------+
| actor                      |
| actor_info                  |
| ...                         |
| store                       |
+-----------------------------+
```

Restoring SQL dumps like the one produced by `mysqldump` or `mysqlpump` (discussed in the next section) is a resource-heavy operation. By default, it's also a serial process, which might take a significant amount of time. There are a couple of tricks you can use to make this process faster, but keep in mind that mistakes can lead to missing or incorrectly restored data. Options include:

- Parallel restore per-schema/per-database
- Parallel restore of objects within a schema

The first one is easily done if the dumping with `mysqldump` is done on a per-database basis. The backup process can also be parallelized if consistency across databases isn't required (it won't be guaranteed). The following example uses the `&` modifier, which instructs the shell to execute the preceding command in the background:

```
$ mysqldump sakila > /tmp/sakila.sql &
$ mysqldump nasa > /tmp/nasa.sql &
```

The resulting dumps are independent. `mysqldump` doesn't process users and grants unless the `mysql` database is dumped, so you need to take care of that. Restoration is just as straightforward:

```
$ mysql sakila < /tmp/sakila.sql &
$ mysql nasa < /tmp/nasa.sql &
```

On Windows, it's also possible to send command execution to the background using the PowerShell command `Start-Process` or, in later versions, the same `&`.

The second option is a bit more involved. Either you need to dump on a per-table basis (e.g., `mysqldump sakila artists > sakila.artists.sql`), which results in a straightforward restore, or you need to go ahead and edit the dump file to split it into multiple ones. Taken to the extreme, you can even parallelize data insertion on the table level, although that’s probably not going to be practical.

Although this is doable, it’s preferable to use tools that are purpose-built for this task.

mysqlpump

`mysqlpump` is a utility program bundled with MySQL versions 5.7 and later that improves `mysqldump` in several areas, mainly around performance and usability. The key differentiators are as follows:

- Parallel dump capability
- Built-in dump compression
- Improved restore performance though delayed creation of secondary indexes

- Easier control over what objects are dumped
- Modified behavior of dumping user accounts

Using the program is very similar to using `mysqldump`. The main immediate difference is that when no arguments are passed, `mysqlpump` will default to dumping all of the databases (excluding `INFORMATION_SCHEMA`, `performance_schema`, `ndbinfo`, and the `sys` schema). The other notable things are that there’s a progress indicator and that `mysqlpump` defaults to parallel dump with two threads:

```
$ mysqlpump > pump.out


Dump progress: 1/2 tables, 0/530419 rows
Dump progress: 80/184 tables, 2574413/646260694 rows
...
Dump progress: 183/184 tables, 16297773/646260694 rows
Dump completed in 10680
```

The concept of parallelism in `mysqlpump` is somewhat complicated. You can use concurrency between different databases and between different objects within a given database. By default, when no other parallel options are specified, `mysqlpump` will use a single queue with two parallel threads to process all databases and user definitions (if requested). You can control the level of parallelism of the default queue using the `--default-parallelism` argument. To further fine-tune concurrency, you can set up multiple parallel queues to process separate databases. Take care when choosing your desired concurrency level, since you could end up using most of the database resources for the backup run.

An important distinction from `mysqldump` when using `mysqlpump` lies in how the latter handles user accounts. `mysqldump` managed users by dumping `mysql.user` and other relevant tables. If the `mysql` database wasn't included in the dump, no user information will be preserved. `mysqlpump` improves on that by introducing the command-line arguments `--users` and `--include-users`. The first one tells the utility to add user-related commands to the dump for all users, and the second accepts a list of usernames. This is a great improvement on the old way of doing things.

Let's combine all the new features to produce a compressed dump of non-system databases and user definitions, and use concurrency in the process:

```
$ mysqlpump --compress-output=zlib --include-users=bob,
--include-databases=sakila,nasa,employees \
--parallel-schemas=2:employees \
--parallel-schemas=sakila,nasa > pump.out
```

```
Dump progress: 1/2 tables, 0/331579 rows
Dump progress: 19/23 tables, 357923/3959313 rows
...
Dump progress: 22/23 tables, 3755358/3959313 rows
Dump completed in 10098
```

NOTE

`mysqlpump` output can be compressed with the ZLIB or LZ4 algorithms. When the OS-level commands `lz` and `openssl zlib` aren't available, you can use the `lz4_decompress` and `zlib_decompress` utilities included in your MySQL distribution.

A dump resulting from a `mysqlpump` run is not suitable for parallel restore because the data inside it is interleaved. For example, the following is the result of a `mysqlpump` execution showing table creation amidst inserts to tables in different databases:

```
...,(294975,"1955-07-31","Lucian","Rosis","M","1986-12-
CREATE TABLE `sakila`.`store` (
`store_id` tinyint unsigned NOT NULL AUTO_INCREMENT,
`manager_staff_id` tinyint unsigned NOT NULL,
`address_id` smallint unsigned NOT NULL,
`last_update` timestamp NOT NULL DEFAULT
CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
PRIMARY KEY (`store_id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT
CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
;
INSERT INTO `employees`.`employees` VALUES
(294976,"1961-03-19","Rayond","Khalid","F","1989-11-03"
```

`mysqlpump` is an improvement over `mysqldump` and adds important concurrency, compression, and object control features. However, the tool doesn't allow parallel restore of the dump and in fact makes it impossible. The only improvement to the restore performance is that secondary indexes are added after the main load is complete.

mydumper and myloader

`mydumper` and `myloader` are both part of the open source project `mydumper`. This set of tools attempts to make logical backups more performant, easier to manage, and more human-friendly. We won't go into too much depth here, as we could easily run out of space in the book covering every possible MySQL backup variety.

These programs can be installed either by taking the freshest release from the project's GitHub page or by compiling the source. At the time of writing, the latest release is somewhat behind the main branch. Step-by-step installation instructions are available in "Setting up the mydumper and myloader utilities".

We previously showed how `mysqlpump` improves dumping performance but mentioned that its intertwined outputs don't help with restoration. `mydumper` combines the parallel dumping approach with preparing ground for parallel restore with `myloader`. That's achieved by dumping every table into a separate file.

The default invocation of `mydumper` is very simple. The tool tries to connect to the database, initiates a consistent dump, and creates a directory under the current one for the export files. Note that each table has its own file. By default, `mydumper` will also dump the `mysql` and `sys` databases. The default parallelism setting for the dump operation is `4`, meaning four separate tables will be read simultaneously. `myloader` invoked on this directory will be able to restore the tables in parallel.

To create the dump and explore it, execute the following commands:

```
$ mydumper -u root -a
```

```
Enter the MySQL password:
```

```
$ ls -ld export
```

```
drwx... export-20210613-204512
```

```
$ ls -la export-20210613-204512
```

```
...
-rw... sakila.actor.sql
-rw... sakila.address-schema.sql
-rw... sakila.address.sql
-rw... sakila.category-schema.sql
-rw... sakila.category.sql
-rw... sakila.city-schema.sql
-rw... sakila.city.sql
...
```

Apart from parallel dumping and restore capabilities, `mydumper` has some more advanced features:

- Lightweight backup locks support. Percona Server for MySQL implements some additional lightweight locking that's used by Percona XtraBackup. `mydumper` utilizes these locks by default when possible. These locks do not block concurrent reads and writes to InnoDB tables, but will block any DDL statements, which could otherwise render the backup invalid.
- Use of savepoints. `mydumper` uses a trick with transaction savepoints to minimize metadata locking.
- Limits on duration of metadata locking. To work around prolonged metadata locking, a problem we described in "Metadata Locks", `mydumper` allows two options: failing quickly or killing long-running queries that prevent `mydumper` from succeeding.

`mydumper` and `myloader` are advanced tools taking logical backup capabilities to the maximum. However, as part of a community project, they lack the documentation and polish that other tools provide. Another major downside is the lack of any support or guarantees. Still, they can be a useful addition to a database operator's toolbelt.

Cold Backup and Filesystem Snapshots

The cornerstone of physical backups, a *cold backup* is really just a copy of the data directory and other necessary files, done while the database instance is down. This technique isn't frequently used, but it can save the day when you need to create a consistent backup quickly. With databases now regularly approaching the multiterabyte size range, just copying the files can take a very long time. However, the cold backup still has its good points:

- Very fast (arguably the fastest backup method apart from snapshots)
- Straightforward
- Easy to use, hard to do wrong

Modern storage systems and some filesystems have readily available snapshot capabilities. They allow you to create near-instantaneous copies of volumes of arbitrary size by utilizing internal mechanisms. The properties of different snapshot-capable systems vary widely, making it impossible for us to cover all of them. However, we can still talk a bit about them from the database perspective.

Most snapshots will be *copy-on-write* (COW) and internally consistent to some point in time. However, we already know that database files aren't consistent on disk, especially with transactional storage engines like InnoDB. This makes it somewhat difficult to get the snapshot backup right. There are two options:

Cold backup snapshot

> When the database is shut down, its data files may still not be perfectly consistent. But if you do a snapshot of all of the database files (including InnoDB redo logs, for example), together they will allow for the database to start. That's only natural, because otherwise the database would lose data on every restart. Don't forget that you may have database files split among many volumes. You will need to have all of them. This method will work for all storage engines.

Hot backup snapshot

> With a running database, taking a snapshot correctly is a greater challenge than when the database is down. If your database files are located over multiple volumes, you cannot guarantee that snapshots, even initiated simultaneously, will be consistent to the same point in time, which can lead to disastrous results. Moreover, nontransactional storage engines like MyISAM don't guarantee consistency for files on disk while the database is running. That's actually true for InnoDB as well, but InnoDB's redo logs are always consistent (unless safeguards are disabled), and MyISAM lacks this functionality.

The recommended way to do a hot backup snapshot would therefore be to utilize some amount of locking. Since the snapshot-taking process is usually a quick one, the resulting downtime shouldn't be significant. Here's the process:

1. Create a new session, and lock all of the tables with the `FLUSH TABLES WITH READ LOCK` command. This session cannot be closed, or else locks will be released.
2. Optionally, record the current binlog position by running the `SHOW MASTER STATUS` command.

3. Create snapshots of all volumes where MySQL’s database files are located according to the storage system’s manual.
4. Unlock the tables with the `UNLOCK TABLES` command in the session opened initially.

This general approach should be suitable for most if not all of the current storage system and filesystems capable of doing snapshots. Note that they all differ subtly in the actual procedure and requirements. Some cloud vendors require you to additionally perform an `fsfreeze` on the filesystems.

Always test your backups thoroughly before implementing them in production and trusting them with your data. You can only trust a solution that you’ve tested and are comfortable using. Copying arbitrary backup strategy suggestions is not a very good idea.

Percona XtraBackup

The logical step forward in physical backups is implementing so-called *hot backups*—that is, making a copy of database files while the database is running. We’ve already mentioned that MyISAM tables can be copied, but that doesn’t work for InnoDB and other transactional storage engines like MyRocks. The problem therefore is that you can’t just copy the files because the database is constantly undergoing changes. For example, InnoDB might be flushing some dirty pages in the background even if no writes are hitting the database right now. You can try your luck and copy the database directory under a running system and then try to restore that directory and start a MySQL server using it. Chances are, it won’t work. And while it may work sometimes, we strongly recommend against taking chances with database backups.

The capability to perform hot backups is built into three main MySQL backup tools: Percona XtraBackup, MySQL Enterprise Backup, and `mariabackup`. We’ll briefly talk about all of them, but will mainly concentrate on the XtraBackup utility. It’s important to understand that all the tools share properties, so knowing how to use one will help you use the others.

Percona XtraBackup is a free and open source tool maintained by Percona and the wider MySQL community. It’s capable of performing online backups of MySQL instances with InnoDB, MyISAM, and MyRocks tables. The program is available only on Linux. Note that it's impossible to use XtraBackup with recent versions of MariaDB: only MySQL and Percona Server are supported. For MariaDB, use the utility we cover in "mariabackup".

Here is an overview of how XtraBackup operates:

1. Records the current log sequence number (LSN), an internal version number for the operation
2. Starts accumulating InnoDB *redo data* (the type of data InnoDB stores for crash recovery)
3. Locks tables in the least intrusive way possible
4. Copies InnoDB tables
5. Locks nontransactional tables completely
6. Copies MyISAM tables
7. Unlocks all tables
8. Processes MyRocks tables, if present
9. Puts accumulated redo data alongside the copied database files

The main idea behind XtraBackup and hot backups in general is combining the no-downtime nature of logical backups with the performance and relative lack of performance impact of cold backups. XtraBackup doesn't guarantee no disruption of service, but it's a great step forward compared with a regular cold backup. The lack of performance impact means that XtraBackup will use some CPU and I/O, but only that needed to copy the database files. Logical backups, on the other hand, must pass each row through all of the database internals, making them inherently slow.

NOTE

XtraBackup requires physical access to the database files and cannot be run remotely. This makes it unsuitable for doing offsite backups of managed databases (DBaaS), for example. Some cloud vendors, however, allow you to import databases using backups made by this tool.

The XtraBackup utility is widely available in various Linux distributions' repositories and thus can easily be installed using a package manager. Alternatively, you can download packages and binary distributions directly from the XtraBackup Downloads page on Percona's website.

WARNING

To back up MySQL 8.0, you must use XtraBackup 8.0. The minor versions of XtraBackup and MySQL ideally should also match: XtraBackup 8.0.25 is guaranteed to work with MySQL 8.0.25. For MySQL 5.7 and older releases, use XtraBackup 2.4.

Backing Up and Recovering

Unlike other tools we've mentioned previously, XtraBackup, by nature of it being a physical backup tool, requires not only access to the MySQL server but also read access to the database files. On most MySQL installations, that usually means that the `xtrabackup` program should be run by the `root` user, or `sudo` must be used. We'll be using the `root` user throughout this section, and we set up a login path using the steps from "Login Path Configuration File".

First, we need to run the basic `xtrabackup` command:

```
# xtrabackup --host=127.0.0.1 --target-dir=/tmp/backup
```

```
...
Using server version 8.0.25
210613 22:23:06 Executing LOCK INSTANCE FOR BACKUP...
...
210613 22:23:07 [01] Copying ./sakila/film.ibd
    to /tmp/backup/sakila/film.ibd
210613 22:23:07 [01]        ...done
...
210613 22:23:10 [00] Writing /tmp/backup/xtrabackup_inf
210613 22:23:10 [00]        ...done
xtrabackup: Transaction log of lsn (6438976119)
    to (6438976129) was copied.
210613 22:23:11 completed OK!
```

If the login path doesn't work, you can pass `root` user's credentials to `xtrabackup` using the `--user` and `--password` command-line arguments. XtraBackup will usually be able to identify the target server's data directory by reading the default option files, but if that doesn't work or you have multiple installations of MySQL, you may need to specify the `--`

`datadir` option, too. Even though `xtrabackup` only works locally, it still needs to connect to a local running MySQL instance and thus has `--host`, `--port`, and `--socket` arguments. You may need to specify some of them according to your particular setup.

TIP

While we use */tmp/backup* as the backup's destination path for our example, you should avoid storing important files under */tmp*. That's especially true for backups.

The result of that `xtrabackup --backup` invocation is a bunch of database files, which are actually not consistent to any point in time, and a chunk of redo data that InnoDB won't be able to apply:

```
# ls -l /tmp/backup/


...
drwxr-x---.  2 root root       160 Jun 13 22:23 mysql
-rw-r-----.  1 root root  46137344 Jun 13 22:23 mysql.i
drwxr-x---.  2 root root        60 Jun 13 22:23 nasa
drwxr-x---.  2 root root       580 Jun 13 22:23 sakila
drwxr-x---.  2 root root       580 Jun 13 22:23 sakila_
drwxr-x---.  2 root root        80 Jun 13 22:23 sakila_
drwxr-x---.  2 root root        60 Jun 13 22:23 sys
...
```

To make the backup ready for future restore, another phase must be performed —preparation. There's no need to connect to a MySQL server for that:

```
# xtrabackup --target-dir=/tmp/backup --prepare


...
xtrabackup: cd to /tmp/backup/
xtrabackup: This target seems to be not prepared yet.
...
Shutdown completed; log sequence number 6438976524
210613 22:32:23 completed OK!
```

The resulting data directory is actually perfectly ready to be used. You can start up a MySQL instance pointing directly to this directory, and it will work. A very common mistake here is trying to start MySQL Server under the `mysql` user while the restored and prepared backup is owned by `root` or another OS user. Make sure to incorporate `chown` and `chmod` as required into your backup recovery procedure. However, there's a useful user experience feature of `--copy-back` available. `xtrabackup` preserves the original database file layout locations, and invoked with `--copy-back` will restore all files to their original locations:

```
# xtrabackup --target-dir=/tmp/backup --copy-back
```

```
...
Original data directory /var/lib/mysql is not empty!
```

That didn't work, because our original MySQL Server is still running, and its data directory is not empty. XtraBackup will refuse to restore a backup unless the target data directory is empty. That should protect you from accidentally restoring a backup. Let's shut down the running MySQL Server, remove or move its data directory, and restore the backup:

```
# systemctl stop mysqld
# mv /var/lib/mysql /var/lib/mysql_old
# xtrabackup --target-dir=/tmp/backup --copy-back
```

```
...
210613 22:39:01 [01] Copying ./sakila/actor.ibd
    to /var/lib/mysql/sakila/actor.ibd
210613 22:39:01 [01]        ...done
...
210613 22:39:01 completed OK!
```

After that, the files are in their correct locations, but owned by `root`:

```
# ls -l /var/lib/mysql/
```

```
drwxr-x---. 2 root root      4096 Jun 13 22:39 sakila
drwxr-x---. 2 root root      4096 Jun 13 22:38 sakila_m
drwxr-x---. 2 root root      4096 Jun 13 22:39 sakila_r
```

You'll need to change the owner of the files back to `mysql` (or the user used in your system) and fix the directory permissions. Once that's done, you can start MySQL and verify the data:

```
# chown -R mysql:mysql /var/lib/mysql/
# chmod o+rx /var/lib/mysql/
# systemctl start mysqld
# mysql sakila -e "SHOW TABLES;"
```

```
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
...
| store                      |
+----------------------------+
```

TIP

The best practice is to do both the backup and prepare work during the backup phase, minimizing the number of possible surprises later. Imagine having the prepare phase fail while you're trying to recover some data! However, note that incremental backups that we cover later have special handling procedures contradicting this tip.

Advanced Features

In this section we discuss some of XtraBackup's more advanced features. They are not required to use the tool, and we give them just as a brief overview:

Database file verification

While performing the backup, XtraBackup will verify the checksums of all of the pages of the data files it's processing. This is an attempt to alleviate the inherent problem of physical backups, which is that they will contain any corruptions in the source database. We recommend augmenting this check with other steps listed in "Testing and Verifying Your Backups".

Compression

Even though copying physical files is much faster than querying the database, the backup process can be limited by disk performance. You cannot decrease the amount of data you read, but you can utilize compression to make the backup itself smaller, decreasing the amount of data that has to be written. That's especially important when a backup destination is a network location. In addition, you will just use less space for storing backups. Note that, as we showed in "The mysqldump Program", on a CPU-choked system compression may actually increase the time it takes to create a backup. XtraBackup uses the `qpress` tool for compression. This tool is available in the `percona-release` package:

```
# xtrabackup --host=127.0.0.1 \
--target-dir=/tmp/backup_compressed/ \
--backup --compress
```

Parallelism

It's possible to run the backup and copy-back processes in parallel by using the `--parallel` command-line argument.

Encryption

In addition to being able to work with encrypted databases, it's also possible for XtraBackup to create encrypted backups.

Streaming

Instead of creating a directory full of backed-up files, XtraBackup can stream the resulting backup in an `xbstream` format. This results in more portable backups and allows integration with `xbcloud`. You can stream backups over SSH, for example.

Cloud upload

Backups taken with XtraBackup can be uploaded to any S3-compatible storage using `xbcloud`. S3 is Amazon's object storage facility and an API that is widely adopted by many companies. This tool only works with backups streamed through the `xbstream` .

Incremental Backups with XtraBackup

As described earlier, a hot backup is a copy of every byte of information in the database. This is how XtraBackup works by default. But in a lot of cases, databases undergo change at an *irregular* rate—new data is added frequently, while old data doesn't change that much (or at all). For example, new financial records may be added every day, and accounts get modified, but in a given week only a small percentage of accounts are changed. Thus, the next logical step in improving hot backups is adding the ability to perform so-called *incremental backups*, or backups of only the changed data. That will allow you to perform backups more frequently by decreasing the need for space.

For incremental backups to work, you need first to have a full backup of the database, called a *base backup*—otherwise there's nothing to increment from. Once your base backup is ready, you can perform any number of incremental backups, each consisting of the changes made since the previous one (or from the base backup in the case of the first incremental backup). Taken to the extreme, you could create an incremental backup every minute, achieving something called *point-in-time recovery* (PITR), but this is not very practical, and as you will soon learn there are better ways to do that.

Here's an example of the XtraBackup commands you could use to create a base backup and then an incremental backup. Notice how the incremental backup points to the base backup via the `--incremental-basedir` argument:

```
# xtrabackup --host=127.0.0.1 \
--target-dir=/tmp/base_backup --backup
# xtrabackup --host=127.0.0.1 --backup \
--incremental-basedir=/tmp/base_backup \
--target-dir=/tmp/inc_backup1
```

If you check the backup sizes, you'll see that the incremental backup is very small compared to the base backup:

```
# du -sh /tmp/base_backup
```

```
2.2G    /tmp/base_backup
6.0M    /tmp/inc_backup1
```

Let's create another incremental backup. In this case, we'll pass the previous incremental backup's directory as the base directory:

```
# xtrabackup --host=127.0.0.1 --backup \
--incremental-basedir=/tmp/inc_backup1 \
--target-dir=/tmp/inc_backup2
```

```
210613 23:32:20 completed OK!
```

You may be wondering whether it's possible to specify the original base backup's directory as the `--incremental-basedir` for each new incremental backup. In fact, that results in a completely valid backup, which is a variation of an incremental backup (or the other way around). Such incremental backups that contain changes made not just since the previous incremental backup but since the base backup are usually called *cumulative* backups. Incremental backups targeting any previous backup are called *differential* backups. Cumulative incremental backups usually consume more space, but can considerably decrease the time needed for the prepare phase when a backup is restored.

Importantly, the prepare process for incremental backups differs from that for regular backups. Let's prepare the backups we've just taken, starting with the base backup:

```
# xtrabackup --prepare --apply-log-only \
--target-dir=/tmp/base_backup
```

The `--apply-log-only` argument tells `xtrabackup` to not finalize the prepare process, as we still need to apply changes from the incremental backups. Let's do that:

```
# xtrabackup --prepare --apply-log-only \
--target-dir=/tmp/base_backup \
--incremental-dir=/tmp/inc_backup1
# xtrabackup --prepare --apply-log-only \
--target-dir=/tmp/base_backup \
--incremental-dir=/tmp/inc_backup2
```

All commands should report `completed OK!` at the end. Once the `--prepare --apply-log-only` operation is run, the base backup advances to the point of the incremental backup, making PITR to an earlier time impossible. So, it's not a good idea to prepare immediately when performing incremental backups. To finalize the prepare process, the base backup with the changes applied from incremental backups must be prepared normally:

```
# xtrabackup --prepare --target-dir=/tmp/base_backup
```

Once the base backup is "fully" prepared, attempts to apply incremental backups will fail with the following message:

```
xtrabackup: This target seems to be already prepared.
xtrabackup: error: applying incremental backup needs
    target prepared with --apply-log-only.
```

Incremental backups are inefficient when the relative amount of changes in the database is high. In the worst case, where every row in the database was changed between full a backup and an incremental backup, the latter will actually just be a full backup, storing 100% of the data. Incremental backups are most efficient when most of the data is appended and the relative amount of old data being changed is low. There are no rules regarding this, but if 50% of your data changes between your base backup and an incremental backup, consider not using incremental backups.

Other Physical Backup Tools

XtraBackup isn't the only tool available that's capable of performing hot MySQL physical backups. Our decision to explain the concepts using that particular tool was driven by our experience with it. However, that doesn't mean that other tools are worse in any way. They may well be better suited to your needs. However, we have limited space, and the topic of backing up is very wide. We could write a *Backing Up MySQL* book of considerable volume!

That said, to give you an idea of some of the other options, let's take a look at two other readily available physical backup tools.

MySQL Enterprise Backup

Called MEB for short, this tool is available as part of Oracle's MySQL Enterprise Edition. It's a closed-source proprietary tool that is similar in functionality to XtraBackup. You'll find comprehensive documentation for it on the MYSQL website. The two tools are currently at feature parity, so almost everything that was covered for XtraBackup will be true for MEB as well.

MEB's standout property is that it's truly a cross-platform solution. XtraBackup works only on Linux, whereas MEB also works on Solaris, Windows, macOS, and FreeBSD. MEB doesn't support flavors of MySQL other than Oracle's standard one.

Some additional features that MEB has, which are not available in XtraBackup, include the following:

- Backup progress reporting
- Offline backups
- Tape backups through Oracle Secure Backups
- Binary and relay log backups
- Table rename at restore time

mariabackup

`mariabackup` is a tool by MariaDB for backing up MySQL databases. Originally forked from XtraBackup, this is a free open source tool that is available on Linux and Windows. The standout property of `mariabackup` is its seamless work with the MariaDB fork of MySQL, which continues to diverge significantly from both the mainstream MySQL and Percona Server. Since this is a direct fork of XtraBackup, you'll find many similarities in how the tools are used and in their properties. Some of XtraBackup's newer features, like backup encryption and secondary index omission, are not present in `mariabackup`. However, using XtraBackup to back up MariaDB is currently impossible.

Point-in-Time Recovery

Now that you're familiar with the concept of hot backups, you have almost everything you need to complete your backup toolkit. So far all the backup types that we've discussed share a similar trait—a deficiency. They allow restore only at the point in time when they were taken. If you have two backups, one done at 23:00 on Monday and the second at 23:00 on Tuesday, you cannot restore to 17:00 on Tuesday.

Remember the infrastructure failure example given at the beginning of the chapter? Now, let's make it worse and say that the data is gone, all the drives failed, and there's no replication. The event happened on Wednesday at 21:00. Without PITR and with daily backups taken at 23:00, this means that you've just lost a full day's worth of data irrevocably. Arguably, incremental backups done with XtraBackup allow you to make that problem somewhat less pronounced, but they still leave some room for data loss, and it's less than practical to be running them very often.

MySQL maintains a journal of transactions called the *binary log*. By combining any of the backup methods we've discussed so far with binary logs, we get the ability to restore to a transaction at an arbitrary point in time. It's very important to understand that you need both a backup *and* binary logs from after the backup for this to work. You also cannot go back in time, so you cannot recover the data to a point in time before your oldest base backup or dump was created.

Binary logs contain both transaction timestamps and their identifiers. You can rely on either for recovery, and it is possible to tell MySQL to recover to a certain timestamp. This is not a problem when you want to recover to the latest point in time, but can be extremely important and helpful when trying to perform a restore to fix a logical inconsistency, like the one described in "Deployment Bug". However, in most situations, you will need to identify a specific problematic transaction, and we'll show you how to do that.

One interesting peculiarity of MySQL is that it allows for PITR for logical backups. “Loading Data from a SQL Dump File” discusses storing the binlog position for replica provisioning using `mysqldump`. The same binlog position can be used as a starting point for PITR. Every backup type in MySQL is suitable for PITR, unlike in other databases. To facilitate this property, make sure to note the binlog position when taking your backup. Some backup tools do that for you. When using those that don’t, you can run `SHOW MASTER STATUS` to get that data.

Technical Background on Binary Logs

MySQL differs from a lot of other mainstream RDBMS in that it supports multiple storage engines, as discussed in “Alternative Storage Engines”. Not only that, but it supports multiple storage engines for tables inside a single database. As a result, some concepts in MySQL are different from in other systems.

Binary logs in MySQL are essentially transaction logs. When binary logging is enabled, every transaction (excluding read-only transactions) will be reflected in the binary logs. There are three ways to write transactions to binary logs:

Statement

In this mode, statements are logged to the binary logs as they were written, which might cause indeterministic execution in replication scenarios.

Row

In this mode, statements are broken down into minimal DML operations, each modifying a single specific row. Although it guarantees deterministic execution, this mode is the most verbose and results in the largest files and thus the greatest I/O overhead.

Mixed

In this mode, “safe” statements are logged as is, while others are broken down.

Usually, in database management systems, the transaction log is used for crash recovery, replication, and PITR. However, because MySQL supports multiple storage engines, its binary logs can’t be used for crash recovery. Instead, each engine maintains its own crash recovery mechanism. For example, MyISAM is not crash-safe, whereas InnoDB has its own redo logs. Every transaction in MySQL is a distributed transaction with two-phase commit, to allow for this multiengined nature. Each committed transaction is guaranteed to be reflected in the storage engine’s redo logs, if the engine is transactional, as well as in MySQL’s own transaction log (the binary logs).

NOTE

Binary logging has to be enabled in your MySQL instance for PITR to be possible. You should also default to having `sync_binlog=1` , which guarantees the durability of each write. Refer to the MySQL documentation to understand the trade-offs of disabling binlog syncing.

We’ll talk more about how binary logs work in Chapter 13.

Preserving Binary Logs

To allow PITR, you must preserve the binary logs starting from the binlog position of the oldest backup. There are few ways to do this:

- Copy or sync binary logs “manually” using some readily available tool like `rsync` . Remember that MySQL continues to write to the current binary log file. If you’re copying files instead of continuously syncing them, do not copy the current binary log file. Continuously syncing files will take care of this problem by overwriting the partial file once it becomes non-current.
- Use `mysqlbinlog` to copy individual files or stream binlogs continuously. Instructions are available in the documentation.
- Use MySQL Enterprise Backup, which has a built-in binlog copy feature. Note that it’s not a continuous copying, but relies on incremental backups to have binlog copies. This allows for PITR between two backups.
- Allow MySQL Server to store all the needed binary logs in its data directory by setting a high value for the `binlog_expire_logs_seconds` or `expire_logs_days` variables. This option should ideally not be used on its own, but can be used in addition to any of the others. If anything happens to the data directory, like filesystem corruption, binary logs stored there may also get lost.

Identifying a PITR Target

You may use the PITR technique to achieve two objectives:

1. Recover to the latest point in time.
2. Recover to an arbitrary point in time.

The first one, as discussed earlier, is useful to recover a completely lost database to the latest available state. The second is useful to get data as it was before. An example of a case when this can be useful was given in "Deployment Bug". To recover lost or incorrectly modified data, you can restore a backup and then recover it to a point in time just before the deployment was executed.

Identifying the actual specific time when an issue happened can be a challenge. More often than not, the only way for you to find the desired point in time is by inspecting binary logs written around the time when the issue occurred. For example, if you suspect that a table was dropped, you may look for the table name, then for any DDL statements issued on that table, or specifically for a `DROP TABLE` statement.

Let's illustrate that example. First, we need to actually drop a table, so we'll drop the `facilities` table we created in "Loading Data from Comma-Delimited Files". However, before that we'll insert a record that's for sure missing in the original backup:

```
mysql> INSERT INTO facilities(center)
    -> VALUES ('this row was not here before');

Query OK, 1 row affected (0.01 sec)

mysql> DROP TABLE nasa.facilities;

Query OK, 0 rows affected (0.02 sec)
```

We could now go back and restore one of the backups we've taken throughout this chapter, but then we would lose any changes made to the database between that point and the `DROP`. Instead, we'll use `mysqlbinlog` to inspect the content of the binary logs and find the recovery target just before the `DROP` statement was run. To find the list of binary logs available in the data directory, you can run the following command:

```
mysql> SHOW BINARY LOGS;
```

```
+---------------+-----------+-----------+
| Log_name      | File_size | Encrypted |
+---------------+-----------+-----------+
| binlog.000291 |       156 | No        |
| binlog.000292 |       711 | No        |
+---------------+-----------+-----------+
2 rows in set (0.00 sec)
```

WARNING

MySQL won't keep binary logs in its data directory forever. They're removed automatically when they are older than the duration specified under `binlog_expire_logs_seconds` or `expire_log_days`, and also can be removed manually by running `PURGE BINARY LOGS`. If you want to make sure binary logs are available, you should preserve them outside of the data directory as described in the previous section.

Now that the list of binary logs is available, you can either try to search in them, from the newest one to the oldest one, or you can just dump all their contents together. In our example, the files are small, so we can use the latter approach. In any case, the `mysqlbinlog` command is used:

```
# cd /var/lib/mysql
# mysqlbinlog binlog.000291 binlog.000292 \
-vvv --base64-output='decode-rows' > /tmp/mybinlog.sql
```

Inspecting the output file, we can find the problematic statement:

```
...
#210613 23:32:19 server id 1  end_log_pos 200 ... Rotat
...
# at 499
```

```
#210614  0:46:08 server id 1  end_log_pos 576 ...
# original_commit_timestamp=1623620769019544 (2021-06-1
# immediate_commit_timestamp=1623620769019544 (2021-06-
/*!80001 SET @@session.original_commit_timestamp=162362
/*!80014 SET @@session.original_server_version=80025*//
/*!80014 SET @@session.immediate_server_version=80025*/
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 576
#210614  0:46:08 server id 1  end_log_pos 711 ... Xid =
use `nasa`/*!*/;
SET TIMESTAMP=1623620768/*!*/;
DROP TABLE `facilities` /* generated by server */
/*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlb
DELIMITER ;
...
```

We should stop our recovery before 2021-06-14 00:46:08, or at binary log position 499. We’ll also need all binary logs from the latest backup, up to and including *binlog.00291*. Using this information, we can proceed to backup restoration and recovery.

Point-in-Time-Recovery Example: XtraBackup

On its own, XtraBackup doesn’t provide PITR capabilities. You need to add the additional step of running `mysqlbinlog` to replay the binlog contents on the restored database:

1. Restore the backup. See “Backing Up and Recovering” for the exact steps.
2. Start MySQL Server. If you are restoring on the source instance directly, it is recommended to use the `--skip-networking` option to prevent nonlocal clients from accessing the database. Otherwise, some clients may change the database before you’ve actually finished the recovery.
3. Locate the backup’s binary log position. It’s available in the *xtrabackup_binlog_info* file in the backup directory:

```
# cat /tmp/base_backup/xtrabackup_binlog_info


binlog.000291   156
```

4. Find the timestamp or binlog position to which you want to recover—for example, immediately before a `DROP TABLE` was executed, as discussed earlier.
5. Replay the binlogs up to the desired point. For this example, we've preserved binary log *binlog.000291* separately, but you would use your centralized binlog storage for the source of the binary logs. You use the `mysqlbinlog` command for this:

   ```
   # mysqlbinlog /opt/mysql/binlog.000291 \
   /opt/mysql/binlog.000292 --start-position=156 \
   --stop-datetime="2021-06-14 00:46:00" | mysql
   ```

6. Make sure the recovery was successful and that no data is missing. In our case, we'll look for the record we added to the `facilities` table before dropping it:

   ```
   mysql> SELECT center FROM facilities
       -> WHERE center LIKE '%before%';
   ```

   ```
   +-------------------------------+
   | center                        |
   +-------------------------------+
   | this row was not here before |
   +-------------------------------+
   1 row in set (0.00 sec)
   ```

Point-in-Time-Recovery Example: mysqldump

The steps necessary for PITR with `mysqldump` are analogous to the steps taken earlier with XtraBackup. We're only showing this for completeness and so that you can see that PITR is similar with each and every backup type in MySQL. Here's the process:

1. Restore the SQL dump. Again, if your recovery target server is the backup source, you probably want to make it inaccessible to clients.
2. Locate the binary log position in the `mysqldump` backup file:

   ```
   CHANGE MASTER TO MASTER_LOG_FILE='binlog.000010',
   ```

```
MASTER_LOG_POS=191098797;
```

3. Find the timestamp or binlog position to which you want to recover (for example, immediately before a `DROP TABLE` was executed, as discussed before).
4. Replay the binlogs up to the desired point:

```
# mysqlbinlog /path/to/datadir/mysql-bin.000010 \
/path/to/datadir/mysql-bin.000011 \
--start-position=191098797 \
--stop-datetime="20-05-25 13:00:00" | mysql
```

Exporting and Importing InnoDB Tablespaces

One of the major downsides of physical backups is that they usually require a significant portion of your database files to be copied at the same time. Although a storage engine like MyISAM allows for the copying of idle tables' data files, you cannot guarantee consistency of InnoDB files. There are situations, though, where you need to transfer only a few tables, or just one table. So far the only option we've seen for that would be to utilize logical backups, which can be unacceptably slow. The export and import tablespaces feature of InnoDB, officially called *Transportable Tablespace*, is a way to get the best of both worlds. We will also call this feature *export/import* for brevity.

The Transportable Tablespaces feature lets you combine the performance of an online physical backup with the granularity of a logical one. In essence, it offers the ability to do an online copy of an InnoDB table's data files to be used for import into the same or a different table. Such a copy can serve as a backup, or as a way to transfer data between separate MySQL installations.

Why use export/import when a logical dump achieves the same thing? Export/import is much faster and, apart from the table being locked, doesn't impact the server significantly. This is especially true for import. With table sizes in the multigigabyte range, this is one of the few feasible options for data transfer.

Technical Background

To help you understand how this feature works, we'll briefly review two concepts: physical backups and tablespaces.

As we've seen, for a physical backup to be consistent, we can generally take two routes. The first is to shut down the instance, or otherwise make the data read-only in a guaranteed manner. The second is to make the data files consistent to point in time and then accumulate all changes between that point in time and the end of the backup. The Transportable Tablespaces feature works in the first way, requiring the table to be made read-only for a short while.

A tablespace is a file that stores a table's data and its indexes. By default, InnoDB uses the `innodb_file_per_table` option, which forces the creation of a dedicated tablespace file for each table. It's possible to create a tablespace that will contain data for multiple tables, and you can use the "old" behavior of having all tables reside in a single *ibdata* tablespace. However, export is supported only for the default configuration, where there's a dedicated tablespace for each table. Tablespaces exist separately for each partition in a partitioned table, which allows for an interesting ability to transfer partitions between separate tables or create a table from a partition.

Exporting a Tablespace

Now that those concepts have been covered, you know what needs to be done for the export. However, one thing that's still missing is the table definition. Even though most InnoDB tablespace files actually contain a redundant copy of the data dictionary records for their tables, the current implementation of Transportable Tablespaces requires a table to be present on the target before import.

The steps for exporting a tablespace are:

1. Get the table definition.
2. Stop all writes to the table (or tables) and make it consistent.
3. Prepare the extra files necessary for import of the tablespace later:
    1. The *.cfg* file stores metadata used for schema verification.
    2. The *.cfp* file is generated only when encryption is used and contains the transition key that the target server needs to decrypt the tablespace.

To get the table definition, you can use the `SHOW CREATE TABLE` command that we've shown quite a few times throughout this book. All the other steps are done automatically by MySQL with a single command: `FLUSH TABLE ... FOR EXPORT`. That command locks the table and generates the additional required file (or files, if encryption is used) near the regular *.ibd* file of the target table. Let's export the `actor` table from the `sakila` database:

```
mysql> USE sakila
mysql> FLUSH TABLE actor FOR EXPORT;
```

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

The session where `FLUSH TABLE` was executed should remain open, because the `actor` table will be released as soon as the session is terminated. A new file, *actor.cfg*, should appear near the regular *actor.ibd* file in the MySQL data directory. Let's verify:

```
# ls -1 /var/lib/mysql/sakila/actor.
```

```
/var/lib/mysql/sakila/actor.cfg
/var/lib/mysql/sakila/actor.ibd
```

This pair of *.ibd* and *.cfg* files can now be copied somewhere and used later. Once you've copied the files, it's generally advisable to release the locks on the table by running the `UNLOCK TABLES` statement, or closing the session where `FLUSH TABLE` was called. Once all that is done, you have a tablespace ready for import.

NOTE

Partitioned tables have multiple *.ibd* files, and each of them gets a dedicated *.cfg* file. For example:

- *learning_mysql_partitioned#p#p0.cfg*
- *learning_mysql_partitioned#p#p0.ibd*
- *learning_mysql_partitioned#p#p1.cfg*
- *learning_mysql_partitioned#p#p1.ibd*

Importing a Tablespace

Importing a tablespace is quite straightforward. It consists of the following steps:

1. Create a table using the preserved definition. It is not possible to change the table's definition in any way.
2. Discard the table's tablespace.
3. Copy over the *.ibd* and *.cfg* files.
4. Alter the table to import the tablespace.

If the table exists on the target server and has the same definition, then there's no need to perform step 1.

Let's restore the `actor` table in another database on the same server. The table needs to exist, so we'll create it:

```
mysql> USE nasa
mysql> CREATE TABLE `actor` (
    ->  `actor_id` smallint unsigned NOT NULL AUTO_INCR
    ->  `first_name` varchar(45) NOT NULL,
    ->  `last_name` varchar(45) NOT NULL,
    ->  `last_update` timestamp NOT NULL DEFAULT CURREN
    ->    ON UPDATE CURRENT_TIMESTAMP,
    ->  PRIMARY KEY (`actor_id`),
    ->  KEY `idx_actor_last_name` (`last_name`)
    -> ) ENGINE=InnoDB AUTO_INCREMENT=201 DEFAULT
    ->    CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

```
Query OK, 0 rows affected (0.04 sec)
```

As soon as the `actor` table is created, MySQL creates an *.ibd* file for it:

```
# ls /var/lib/mysql/nasa/
```

```
actor.ibd  facilities.ibd
```

This brings us to the next step: discarding this new table's tablespace. That's done by running a special `ALTER TABLE`:

```
mysql> ALTER TABLE actor DISCARD TABLESPACE;
```

```
Query OK, 0 rows affected (0.02 sec)
```

Now the *.ibd* file will be gone:

```
# ls /var/lib/mysql/nasa/
```

```
facilities.ibd
```

WARNING

Discarding the tablespace leads to total deletion of the associated tablespace files and is not a recoverable operation. You will need to recover from a backup if you run `ALTER TABLE ... DISCARD TABLESPACE` accidentally.

We can now copy the exported tablespace of the original `actor` table along with the *.cfg* file:

```
# cp -vip /opt/mysql/actor.* /var/lib/mysql/nasa/
```

```
'/opt/mysql/actor.cfg' -> '/var/lib/mysql/nasa/actor.cf
'/opt/mysql/actor.ibd' -> '/var/lib/mysql/nasa/actor.ib
```

With all the steps done, it's now possible to import the tablespace and verify the data:

```
mysql> ALTER TABLE actor IMPORT TABLESPACE;
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> SELECT * FROM nasa.actor LIMIT 5;
```

```
+----------+------------+--------------+--------------
| actor_id | first_name | last_name    | last_update
+----------+------------+--------------+--------------
|        1 | PENELOPE   | GUINESS      | 2006-02-15 04:
|        2 | NICK       | WAHLBERG     | 2006-02-15 04:
|        3 | ED         | CHASE        | 2006-02-15 04:
|        4 | JENNIFER   | DAVIS        | 2006-02-15 04:
|        5 | JOHNNY     | LOLLOBRIGIDA | 2006-02-15 04:
+----------+------------+--------------+--------------
5 rows in set (0.00 sec)
```

You can see that we have the data from `sakila.actor` in `nasa.actor`.

The best thing about Transportable Tablespaces is probably the efficiency. You can move very large tables between databases easily using this feature.

XtraBackup Single-Table Restore

Perhaps surprisingly, we're going to mention XtraBackup once again in the context of Transportable Tablespaces. That's because XtraBackup allows for the export of the tables from any existing backup. In fact, that's the most convenient way to restore an individual table, and it's also a first building block for a single-table or partial-database PITR.

This is one of the most advanced backup and recovery techniques, and it's completely based on the Transportable Tablespaces feature. It also carries over all of the limitations: for example, it won't work on non-file-per-table tablespaces. We won't give the exact steps here, and only cover this technique so that you know it's possible.

To perform a single-table restore, you should first run `xtrabackup` with the `--export` command-line argument to prepare the table for export. You may notice that the table's name isn't specified in this command, and in reality each table will be exported. Let's run the command on one of the backups we took earlier:

```
# xtrabackup --prepare --export --target-dir=/tmp/base_
# ls -1 /tmp/base_backup/sakila/
```

```
actor.cfg
actor.ibd
address.cfg
address.ibd
category.cfg
category.ibd
...
```

You can see that we have a *.cfg* file for each table: every tablespace is now ready to be exported and imported in another database. From here, you can repeat the steps from the previous section to restore the data from one of the tables.

Single-table or partial-database PITR is tricky, and that's true for most of the database management systems out there. As you saw in "Point-in-Time Recovery", PITR in MySQL is based on binlogs. What that means for partial recovery is that transactions concerning all tables in all databases are recorded, but binlogs can be filtered when applied through replication. Very briefly, therefore, the partial recovery procedure is this: you export the required tables, build a completely separate instance, and feed it with binlogs through a replication channel.

You can find more information in community blogs and articles like "MySQL Single Table PITR", "Filtering Binary Logs with MySQL", and "How to Make MySQL PITR Faster".

The export/import feature is a powerful technique when used correctly and under certain circumstances.

Testing and Verifying Your Backups

Backups are good only when you're sure you can trust them. There are numerous examples of people having backup systems that failed when most needed. It's entirely possible to be taking backups frequently and still lose the data.

There are multiple ways in which backups can be unhelpful or can fail:

Inconsistent backups

The simplest example of this is a snapshot backup incorrectly taken from multiple volumes when the database is running. The resulting backup may be broken or missing data. Unfortunately, some of the backups you take may be consistent, and others may not be broken or inconsistent enough for you to notice until it's too late.

Corruption of the source database

Physical backups, as we covered extensively, will have copies of all of the database pages, corrupted or not. Some tools try to verify the data as they go, but this is not completely error-free. Your successful backups may contain bad data that cannot be read later.

Corruption of the backups

Backups are just data on their own and as such are susceptible to the same issues as the original data. Your successful backup might end up being completely useless if its data gets corrupted while it's being stored.

Bugs

Things happen. A backup tool you've been using for a dozen years might have a bug that you, of all people, will discover. In the best case, your backup will fail; in the worst case, it might fail to restore.

Operational errors

We're all human, and we make mistakes. If you automate everything, the risk here changes from human errors to bugs.

That's not a comprehensive list of the issues that you might face, but it gives you some insight into the problems you might encounter even when your backup strategy is sound. Let's review some steps you can take to make you sleep better:

- When implementing a backup system, test it thoroughly, and test it in various modes. Make sure you can back up your system, and use the backup for recovery. Test with and without load. Your backups can be consistent when no connection is modifying the data, and fail when that's not true.
- Use both physical and logical backups. They have different properties and failure modes, especially around source data corruption.

- Back up your backups, or just make sure that they are at least as durable as the database.
- Periodically perform backup restoration tests.

The last point is especially interesting. No backup should be considered safe until it's been restored and tested. That means that in a perfect world, your automation will actually try to use the backup to build a database server and report back success only when that goes well. Additionally, that new database can be attached to the source as a replica, and a data verification tool like `pt-table-checksum` from Percona Toolkit can be used to check the data consistency.

Here are some possible steps for backup data verification for physical backups:

1. Prepare the backup.
2. Restore the backup.
3. Run `innochecksum` on all of the *.ibd* files.
   The following command will run four `innochecksum` processes in parallel on Linux:

   ```
   $ find . -type f -name "*.ibd" -print0 |\
   xargs -t -0r -n1 --max-procs=4 innochecksum
   ```

4. Start a new MySQL instance using the restored backup. Use a spare server, or just a dedicated *.cnf* file, and don't forget to use nondefault ports and paths.
5. Use `mysqldump` or any alternative to dump all of the data, making sure it's readable and providing another copy of the backup.
6. Attach the new MySQL instance as a replica to the original source database, and use `pt-table-checksum` or any alternative to verify that the data matches. The procedure is nicely explained in the `xtrabackup` documentation, among other sources.

These steps are complex and might take a long time, so you should decide whether it's appropriate for your business and environment to utilize all of them.

Database Backup Strategy Primer

Now that we’ve covered many of the bits and pieces related to backups and recovery, we can piece together a robust backup strategy. Here are the elements we’ll need to consider:

Point-in-time recovery

> We need to decide whether we’ll need PITR capabilities, as that’ll drive our decisions regarding the backup strategy. You have to make the call for your specific case, but our suggestion is to default to having PITR available. It can be a lifesaver. If we decide that we’re going to need this capability, we need to set up binary logging and binlog copying.

Logical backups

> We will likely need logical backups, either for their portability or for the corruption safeguard. Since logical backups load the source database significantly, schedule them for a time when there’s the least load. In some circumstances it won’t be possible to do logical backups of your production database, either due to time or load constraints, or both. Since we still want to have the ability to run logical backups, we can use following techniques:
>
> - Run logical backups on a replicated database. It can be problematic to track binlog position in this case, so it’s recommended to use GTID-based replication in this case.
> - Incorporate creation of logical backups into the physical backup’s verification process. A prepared backup is a data directory that can by used by a MySQL server right away. If you run a server targeting the backup, you will spoil that backup, so you need to copy that prepared backup somewhere first.

Physical backups

> Based on the OS, MySQL flavor, system properties, and careful review of documentation, we need to choose the tool we’ll be using for physical backups. For the sake of simplicity, we’re choosing XtraBackup here.

The first decision to make is how important the mean time to recovery (MTTR) target is for us. For example, if we only do weekly base backups, we might end up needing to apply almost a week's worth of transactions to recover the backup. To decrease the MTTR, implement incremental backups on a daily or perhaps even hourly basis.

Taking a step back, your system might be so large that even a hot backup with one of the physical backup tools is not viable for you. In that case, you need to go for snapshots of the volumes, if that's possible.

Backup storage

We need to make sure our backups are safely, and ideally redundantly, stored. We might accomplish this with a hardware storage setup utilizing a less-performant but redundant RAID array of level 5 or 6, or with a less reliable storage setup if we also continuously stream our backups to a cloud storage like Amazon's S3. Or we might just default to using S3 if that's possible for us with the backup tools of choice.

Backup testing and verification

Finally, once we have backups in place, we need to implement a backup testing process. Depending on the budget available for implementation and maintenance of this exercise, we should decide how many steps will be run each time and which steps will be run only periodically.

With all of this done, we can say that we have our bases covered and our database safely backed up. It may seem like a lot of effort, considering how infrequently backups are used, but you have to remember that you will eventually face a disaster—it's just a question of time.