Chapter 8. Managing Users and Privileges

The simplest database system is just a bunch of files lying around, with some data in them, and no unified access process. With any RDBMS, we come to expect a significantly higher level of sophistication and abstraction. For one, we want to be able to access the database from multiple clients simultaneously. However, not all of the clients are similar, and not every one of them necessarily needs access to all of the data in the database. It's possible to imagine a database where every user is a superuser, but that would mean you'd have to install a dedicated database for every app and dataset: wasteful. Instead, databases have evolved to support multiple users and roles and provide a means to control privileges and access on a very fine-grained level to guarantee secure shared environments.

Understanding users and privileges is an important part of working efficiently with a database system. Well-planned and managed roles result in a secure system that is easy to manage and work with. In this chapter, we will review most of the things one needs to know about user and privilege management, starting from the basics and moving toward new features like roles. After finishing this chapter, you should have all the basics required to manage access within a MySQL database.

Understanding Users and Privileges

The first building block in the foundation of a shared system is the concept of a *user*. Most modern operating systems have user-based access, so it's highly likely that you already know what that means. Users in MySQL are special objects used for the purpose of:

- Authentication (making sure that a user can access the MySQL server)
- Authorization (making sure that a user can interact with objects in the database)

One thing that makes MySQL distinct from other DBMSs is that users do not *own* schema objects.

Let's consider these points in a little more detail. Every time you access the MySQL server, you must specify a user to be used during authentication. Once you've been authenticated and your identity has been confirmed, you have access to the database. Usually, the user you will be acting as when interacting with schema objects will be the same as the one you used for authentication, but that's not strictly necessary, and that's why we separate the second point. A proxy user is a user that is used for the purpose of checking privileges and actually acting within the database, when another user is used during authentication. That's a rather complex topic and requires nondefault configuration, but it's still possible.

This is an important distinction to remember between authentication and authorization. While you can authenticate with one user, you can be authorized as another and have or not have various permissions.

With these two covered, let's discuss the last point. Some DBMSs support the concept of object ownership. That is, when the user creates a database object —a database or schema, a table, or a stored procedure—that user automatically becomes the new object's owner. The owner usually has the ability to modify objects it owns and grant other users access to them. The important thing here is that MySQL does not in any way have a concept of object ownership.

This lack of ownership makes it even more important to have flexible rules so that users can create objects and then potentially share access to those objects with other users. That is achieved using *privileges*. Privileges can be thought of as sets of rules controlling what actions users are allowed to perform and what data they can access. It's important to understand that by default in MySQL a database user has no privileges at all. Granting a privilege means allowing some action that by default is forbidden.

Users in MySQL are also a bit different than in other databases, because the user object includes a network access control list (ACL). Usually, a MySQL user is represented not just by its name, like `bob`, but with an appended network address, like `bob@localhost`. This particular example defines a user that can be accessed only locally through the loopback interface or a

Unix socket connection. We will touch on this topic later, when we discuss the SQL syntax for creating and manipulating existing users.

MySQL stores all information related to users and privileges in special tables in the `mysql` system database called *grant tables*. We'll talk about this concept in a bit more depth in "Grant Tables".

This short theoretical foundation should be sufficient to form a basic understanding of MySQL's system of users and privileges. Let's get practical and review the commands and capabilities that the database provides for managing users and privileges.

The root User

Every MySQL installation comes with a few users installed by default. Most of them you don't ever need to touch, but there's one that's extremely frequently used. Some might even say that it's overused, but that's not the discussion we want to have here. The user we're talking about is the ubiquitous and all-powerful `root` user. With the same name as the default Unix and Linux superuser, this user is just that in MySQL: a user that can do anything by default.

To be more specific, the user is `root@localhost`, sometimes called the *initial user*. The `localhost` part of the username, as you now know, limits its use to only local connections. When you install MySQL, depending on the specific MySQL flavor and the OS, you might be able to access `root@localhost` from the OS root account by just executing the `mysql` command. In some other cases, a temporary password will be generated for this user.

The initial user is not the only superuser you can create, as you'll see in "The SUPER Privilege". While you can create a `root@<ip>` user, or even a `root@%` user, we strongly discourage you from doing so, as it is a security hole waiting to be exploited. Not every MySQL server even needs to listen on an interface apart from loopback (that is, localhost), let alone have a superuser with a default name available for login. Of course, you can set secure passwords for all users and should probably set one for `root`, but it is arguably that little bit safer to not allow remote superuser access, if that is possible.

For all intents and purposes, `root@localhost` is just a regular user with all privileges granted. You can even drop it, which can happen by mistake. Losing access to the `root@localhost` user is a fairly common problem when running MySQL. You may have set a password and forgotten it or you inherited a server and were not given the password or something else might have happened. We cover the procedure to recover a forgotten password for the `root@localhost` initial user in "Changing root's Password and Insecure Startup". If you dropped your last available superuser, you will have to follow the same procedure but create a new user instead of changing an existing one.

Creating and Using New Users

The first task we'll cover is creating a new user. Let's start with a rather simple example and review each part:

```
CREATE USER
                                    ❶
'bob'@'10.0.2.%'
                                    ❷
IDENTIFIED BY 'password';
                                    ❸
```

❶ SQL statement to create a user

❷ User and host definition

❸ Password specification

Here's a more complex example:

```
CREATE USER
                                    ❶
'bob'@'10.0.2.%'
                                    ❷
IDENTIFIED WITH mysql_native_password
                                    ❸
BY 'password'
                                    ❹
```

```
DEFAULT ROLE 'user_role'
                              ❺

REQUIRE SSL
                              ❻

AND CIPHER 'EDH-RSA-DES-CBC3-SHA'
                              ❼

WITH MAX_USER_CONNECTIONS 10
                              ❽

PASSWORD EXPIRE NEVER;
                              ❾
```

❶ SQL statement to create a user

❷ User and host definition

❸ Authentication plugin specification

❹ Authentication string/password

❺ Default role set once user is authenticated and connected

❻ Require SSL for connections for this user

❼ Require specific ciphers

❽ Limit maximum number of connections from this user

❾ Override global password expiration settings

This is just scratching the surface, but should give an idea of the parameters that can be changed for a user during its creation. There are quite a lot of them. Let's review the specific parts of that statement in a little more detail:

User and host definition

We mentioned in "Understanding Users and Privileges" that users in MySQL are defined not only by their name, but also by hostname. In the previous example, the user is `'bob'@'10.0.2.%'`, where *`bob`* is the username and *`10.0.2.%`* is a hostname specification. In fact, it's a hostname specification with a wildcard. Each time someone connects with the username *`bob`* using TCP, MySQL will do a few things:

1. Get the IP address of the connecting client.
2. Perform a reverse DNS lookup of the IP address to a hostname.

3. Perform a DNS lookup for that hostname (to make sure the reverse lookup wasn't compromised).
4. Check the hostname or IP address with the user's hostname specification.

Only if the hostnames match is access granted. For our user *bob*, a connection from IP address `10.0.2.121` would be allowed, while a connection from `10.0.3.22` would be denied. In fact, to allow connections from another hostname, a new user should be created. Internally, `'bob'@'10.0.2.%'` is a completely different user from `'bob'@'10.0.3.%'`. It's also possible to use fully qualified domain names (FQDNs) in the hostname specification, as in `'bob'@'acme.com'`, but DNS lookups take time, and it's a common optimization to disable them completely.

Specifying all possible hostnames for all users to connect from might be tedious, but it's a useful security feature. However, sometimes a database is set up behind a firewall, or it's simply impractical to specify hostnames. To completely subvert this system, a single wildcard can be used in the host specification, as in `'bob'@'10.0.2%'`. The `'%'` wildcard is also used when you do not specify the host at all (`'bob'@'%'`).

NOTE

When proxying connections to MySQL, pay attention to what IP address MySQL "sees" for incoming connections. For example, when HAProxy is used, by default all connections will come from the IP addresses of machines where HAProxy is running. This fact should be taken into consideration when configuring users. We cover HAProxy configuration for MySQL in Chapter 15.

You'll notice that we've enclosed both the username and the host specification in single quotes (`''`). That is not mandatory, and username and host specification follow a similar set of rules to those that were outlined for table and column names and aliases in "Creating and Using Databases" and "Aliases". For example, when creating or altering the user `bob@localhost` or `bob@172.17.0.2`, there's no need to use any quoting. You can't, though, create this user without

using quotes: `'username with a space'@'172.%'` . Double quotes, single quotes, or backticks can be used to enclose usernames and hostnames with special symbols.

Authentication plugins specification

MySQL supports a wide variety of ways to authenticate users through its system of authentication plugins. These plugins also provide a way for developers to implement new means of authentication without changing MySQL itself. You can set a particular plugin for a user in the creation phase or later.

You might never need to change the plugin for a user, but it's still worth knowing about this subsystem. In particular, LDAP authentication with MySQL can be achieved by using a special authentication plugin. MySQL Enterprise Edition supports a first-class LDAP plugin, and other MySQL versions and flavors can use PAM as a middleman.

NOTE

PAM stands for Pluggable Authentication Modules. It's a standard interface on Unix-like systems that, in very simple terms, allows MySQL to provide authentication by various methods, such as OS passwords, or LDAP. PAM hides the complexity of those authentication methods, and programs like MySQL only need to interface with PAM itself.

MySQL 8.0 uses the `caching_sha2_password` plugin by default, which provides superior security and performance compared to the legacy `mysql_native_password` but is not compatible with every client library. To change the default plugin you can configure the `default_authentication_plugin` variable, which will cause new users to be created with the specified plugin instead.

Authentication string/password

Some authentication plugins, including the default one, require you to set a password for the user. Other plugins, like the PAM one, require you to define a mapping from OS users to MySQL users. `auth_string` will be used in both cases. Let's take a look at an example mapping with PAM:

```
mysql> CREATE USER ''@'' IDENTIFIED WITH auth_pam
    -> AS 'mysqld, dba=dbausr, dev=devusr';
```

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

What's defined here is a mapping that can be read as follows: the PAM configuration file *mysqld* will be used (usually located at */etc/pam.d/mysqld*); OS users with group `dba` will be mapped to MySQL user `dbausr`, and OS users with group `dev` to `devusr`. The mapping alone is not enough, however, as the necessary permissions have to be assigned.

Note that either the Percona PAM plugin or MySQL Enterprise Edition is required for this to work. This example creates a proxy user, which we briefly discussed in "Understanding Users and Privileges".

Using nondefault authentication plugins is a relatively advanced topic, so we're only bringing up PAM here to show you that the authentication string is not always a password.

TIP

You can consult the documentation for details on installing the Percona plugin and the MySQL Enterprise Edition plugin.

Default role set

Roles are a fairly recent addition to MySQL. You may think of a role as a collection of privileges. We discuss them in "Roles".

SSL configuration

You can force connections from particular users to use SSL by passing `REQUIRE SSL` to the `CREATE USER` or `ALTER USER` command. Unencrypted connections to the user will be forbidden. Additionally, you can, as shown in the example we gave, specify a particular cipher suite or a number of suites that can be used for this user. Ideally, you should set the acceptable cipher suites on the system level, but setting this on the user level is useful to allow some less safe suites for specific connections. You don't need to specify `REQUIRE SSL` to

specify `REQUIRE CIPHER`, and in that case unencrypted connections can be established. However, if an encrypted connection is established, it will use only the particular set of ciphers you supply:

```
mysql> CREATE USER 'john'@'192.168.%' IDENTIFIED
    -> REQUIRE CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

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

Additional configurable options that are available include the following:

X509

Forcing a client to present a valid certificate. This, as well as the following options, implies the use of `SSL`.

ISSUER issuer

Forcing a client to present a valid certificate issued by a particular CA, specified in *issuer*.

SUBJECT subject

Forcing a client to present a valid certificate with a particular subject.

These options can be combined to specify a very particular certificate and encryption requirement:

```
mysql> CREATE USER 'john'@'192.168.%'
    -> REQUIRE SUBJECT '/C=US/ST=NC/L=Durham/
    -> O=BI Dept certificate/
    -> CN=client/emailAddress=john@nonexistent.co
    -> AND ISSUER '/C=US/ST=NC/L=Durham/
    -> O=MySQL/CN=CA/emailAddress=ca@nonexistent.
    -> AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Resource consumption limits

You can define resource consumption limits. In our example we are limiting the maximum number of concurrent connections by this user

to 10. This and other resource options default to 0, meaning unlimited. The other possible constraints are `MAX_CONNECTIONS_PER_HOUR`, `MAX_QUERIES_PER_HOUR`, and `MAX_UPDATES_PER_HOUR`. All of these options are part of the `WITH` specification.

Let's create a fairly restricted user, which can run only 10 queries during each given hour, can have only a single concurrent connection, and may not connect more than twice per hour:

```
mysql> CREATE USER 'john'@'192.168.%'
    -> WITH MAX_QUERIES_PER_HOUR 10
    -> MAX_CONNECTIONS_PER_HOUR 2
    -> MAX_USER_CONNECTIONS 1;
```

Note that the number of `MAX_QUERIES_PER_HOUR` is inclusive of `MAX_UPDATES_PER_HOUR` and will limit updates as well. The number of queries also includes everything that the MySQL CLI executes, so setting a really low value is not recommended.

Password management options override

For authentication plugins that deal with passwords, which are stored in grant tables (covered in "Grant Tables"), you can specify a variety of options related to passwords. In our example, we're setting up a user that has the `PASSWORD EXPIRE NEVER` policy, meaning that its password will never expire based on time. You could also create a user that would have a password expiring every other day, or each week.

MySQL 8.0 extends control capabilities to include tracking of failed authentication attempts and the ability to lock an account temporarily. Let's consider an important user with strict control:

```
mysql> CREATE USER 'app_admin'@'192.168.%'
    -> IDENTIFY BY '...'
    -> WITH PASSWORD EXPIRE INTERVAL 30 DAY
    -> PASSWORD REUSE INTERVAL 180 DAY
    -> PASSWORD REQUIRE CURRENT
    -> FAILED_LOGIN_ATTEMPTS 3
    -> PASSWORD_LOCK_TIME 1;
```

This user's password will need to be changed every 30 days, and previous passwords will not be eligible for reuse for 180 days. When changing the password, the current password must be presented. For good measure, we also only allow three consecutive failed login attempts and will block this user for one day if those are made.

Note that these are overrides on the default system options. It's impractical to set up each individual user manually, so we instead recommend that you set up the defaults and only use overrides for particular users. For example, you can have your DBA users' passwords expire more frequently.

There are some other options available for user creation, which we won't cover here. As MySQL evolves, more options become available, but we believe the ones we've covered so far should be enough while learning your way around MySQL.

Since this section is about not only creating but also using new users, let's talk about these uses. They typically fall into a few categories:

Connecting and authenticating

This is the default and most widespread use of any user entity. You specify the user and password, and MySQL authenticates you with that user and your origin host. Then that pair forms a user as defined within grant tables, which will be used for authorization when you access database objects. This is the default situation. You can run the following query to see the currently authenticated user as well as the user provided by the client:

```
mysql> SELECT CURRENT_USER(), USER();

+----------------+----------------+
| CURRENT_USER() | USER()         |
+----------------+----------------+
| root@localhost | root@localhost |
+----------------+----------------+
1 row in set (0.00 sec)
```

Quite unsurprisingly, the records match. This is the most common occurrence, but as you'll see next, it's not the only possibility.

Providing security for stored objects

When a stored object (like a stored procedure or a view) is created, any user can be specified within the `DEFINER` clause of that object. That allows you to execute an object from the standpoint of another user: definer, instead of invoker. This can be a useful way to provide elevated privileges for some specific operation, but it can also be a security hole in your system.

When a MySQL account is specified in the `DEFINER` clause of an object, such as a stored procedure, that account will be used for authorization when the stored procedure is executed or when a view is queried. In other words, the current user of a session changes temporarily. As we mentioned, this can be used to elevate privileges in a controlled manner. For example, instead of granting a user permission to read from some tables, you can create a view with a `DEFINER` : the account you specify will be allowed to access the tables when the view is being queried, but not under any other circumstances. Also, the view itself can further restrict what data is returned. The same is true for stored procedures. To interact with an object that has a `DEFINER` , a caller must have the necessary permissions.

Let's take a look at an example. Here's a simple stored procedure that returns the current user used for authorization, as well as the authenticated user. The `DEFINER` is set to `'bob'@'localhost'` :

```
DELIMITER ;;
CREATE DEFINER = 'bob'@'localhost' PROCEDURE test
BEGIN
    SELECT CURRENT_USER(), USER();
END;
;;
DELIMITER ;
```

If this procedure is executed by the user `john` from the previous examples, output similar to the following will be printed:

```
mysql> CALL test_proc();


+----------------+--------------------+
| CURRENT_USER() | USER()             |
+----------------+--------------------+
| bob@localhost  | john@192.168.1.174 |
+----------------+--------------------+
1 row in set (0.00 sec)
```

It's important to keep this in mind. Sometimes users are not who they appear to be, and to keep your database safe that has to be noted.

Proxying

Some authentication methods, like PAM and LDAP, do not operate with a one-to-one mapping from authenticating users to database ones. We showed before how to create a PAM-authenticated user—let's see what such a user would see if they queried the authenticating and provided users:

```
mysql> SELECT CURRENT_USER(), USER();


+------------------+------------------------+
| CURRENT_USER()   | USER()                 |
+------------------+------------------------+
| dbausr@localhost | localdbauser@localhost |
+------------------+------------------------+
1 row in set (0.00 sec)
```

Before we close this section, we should bring up a couple of important points related to the `CREATE USER` statement. First, it is possible to create multiple user accounts with a single command, instead of executing multiple `CREATE USER` statements individually. Second, if the user already exists, `CREATE USER` doesn't fail, but will change that user in subtle ways. This can be dangerous. To avoid this, you can specify an `IF NOT EXISTS` option to the command. By doing so, you tell MySQL to create the user only if no such user already exists, and do nothing if it does.

At this point, you should have a good understanding of what a MySQL user is and how it can be used. Next we'll show you how users can be modified, but

first you need to understand how user-related information is stored internally.

Grant Tables

MySQL stores both user information and privileges as records in *grant tables*. These are special internal tables in the `mysql` database, which should ideally never be modified manually and instead are implicitly modified when statements like `CREATE USER` or `GRANT` are run. For example, here is the slightly truncated output of a `SELECT` query on the `mysql.user` table, which contains user records, including their passwords (in hashed form):

```
mysql> SELECT * FROM user WHERE user = 'root'\G


*************************** 1. row ***************************
                  Host: localhost
                  User: root
              Select_priv: Y
              Insert_priv: Y
              Update_priv: Y
              Delete_priv: Y
...
      Create_routine_priv: Y
       Alter_routine_priv: Y
         Create_user_priv: Y
               Event_priv: Y
             Trigger_priv: Y
  Create_tablespace_priv: Y
                ssl_type:
              ssl_cipher: 0x
             x509_issuer: 0x
            x509_subject: 0x
           max_questions: 0
             max_updates: 0
         max_connections: 0
    max_user_connections: 0
                  plugin: mysql_native_password
   authentication_string: *E1206987C3E6057289D6C3208EAC
        password_expired: N
   password_last_changed: 2020-09-06 17:20:57
       password_lifetime: NULL
          account_locked: N
        Create_role_priv: Y
```

```
          Drop_role_priv: Y
  Password_reuse_history: NULL
     Password_reuse_time: NULL
Password_require_current: NULL
         User_attributes: NULL
1 row in set (0.00 sec)
```

You can immediately see that a lot of the fields directly correspond to specific invocations of the `CREATE USER` or `ALTER USER` statements. For example, you can see that this `root` user doesn't have any specific rules set regarding its password's lifecycle. You can also see quite a lot of privileges, though we have omitted some for brevity. These are privileges that don't require a target, like a table. Such privileges are called *global*. We'll show you how to view targeted privileges later.

As of MySQL 8.0, the other grant tables are:

`mysql.user`

User accounts, static global privileges, and other nonprivilege columns

`mysql.global_grants`

Dynamic global privileges

`mysql.db`

Database-level privileges

`mysql.tables_priv`

Table-level privileges

`mysql.columns_priv`

Column-level privileges

`mysql.procs_priv`

Stored procedure and function privileges

`mysql.proxies_priv`

Proxy-user privileges

`mysql.default_roles`

Default user roles

mysql.role_edges

Edges for role subgraphs

mysql.password_history

Password change history

You don't need to remember all of these tables, let alone their structure and contents, but you should remember that they exist. When necessary, you can easily look up the necessary structure information in the docs or in the database itself.

Internally, MySQL caches grant tables in memory and refreshes this cached representation every time an account management statement is run and thus modifies grant tables. Cache invalidation happens only for the specific user affected. Ideally, you should never modify these grant tables directly, and there's rarely a use case for that. However, in the unfortunate event that you do need to modify a grant table, you can tell MySQL to reread them by running the `FLUSH PRIVILEGES` command. Failure to do so will mean that the in-memory cache won't get updated until either the database is restarted, an account management statement is run against the same user that was updated directly in the grant tables, or `FLUSH PRIVILEGES` is executed for some other purpose. Even though the command's name suggests it only affects privileges, MySQL will reread information from all of the tables and refresh its cache in memory.

User Management Commands and Logging

There are some direct consequences of the fact that all the commands we're discussing in this chapter are, under the hood, modifying the grant tables. They are close to DML operations in some regards. They are atomic: any `CREATE USER`, `ALTER USER`, `GRANT`, or other such operation either succeeds or fails without actually changing its target. They are logged: all of the changes done to grant tables either manually or through the relevant commands are logged to the binary log. Thus, they are replicated (see Chapter 13) and will also be available for point-in-time recovery (see Chapter 10).

Application of these statements on the source can break replication if the targeted user doesn't exist on the replica. We therefore recommend that you keep your replicas consistent with their sources not only in data, but also in users and other metadata. Of course, it's only "meta" in the sense that it exists outside of your real application data; users are records in the `mysql.user` table, and that should be remembered when setting up replication.

Mostly, replicas are full copies of their sources. In more complex topologies, like fan-in, that may not be true, but even in such cases we recommend keeping users consistent across the topology. In general, it is easier and safer than fixing broken replicas or remembering whether you need to disable binary logging before altering a user.

While we say that execution of `CREATE USER` is similar to an `INSERT` to the `mysql.user` table, the `CREATE USER` statement itself is not changed in any way before being logged. That is true for the binary log, the slow query log (with a caveat), the general query log, and audit logs. The same is true for every other operation discussed in this chapter. The caveat for the slow query log is that an extra server option, `log_slow_admin_statements`, has to be enabled for administrative statements to be logged here.

TIP

You can find the locations of the logs we've mentioned under the following system variable names: `log_bin_basename`, `slow_query_log_file`, and `general_log_file`. Their values can include the full path to the file or just the filename. In the latter case, that file will be in the MySQL server's data directory. Binary logs always have a numeric suffix: for example, *binlog.000271*. We do not cover audit log configuration in this book.

Consider the following example:

```
mysql> CREATE USER 'vinicius' IDENTIFIED BY '...';

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

Here's an example of how the same `CREATE USER` command is reflected in the general, slow query, and binary logs:

General log

```
2020-11-22T15:53:17.354270Z        29 Query
    CREATE USER 'vinicius'@'%' IDENTIFIED BY <sec
```

Slow query log

```
# Time: 2020-11-22T15:53:17.368010Z
# User@Host: root[root] @ localhost []  Id:    29
# Query_time: 0.013772  Lock_time: 0.000161 Rows_
SET timestamp=1606060397;
CREATE USER 'vinicius'@'%' IDENTIFIED BY <secret>
```

Binary log

```
#201122 18:53:17 server id 1  end_log_pos 4113 CR
    Query   thread_id=29    exec_time=0     error
SET TIMESTAMP=1606060397.354238/*!*/;
CREATE USER 'vinicius'@'%' IDENTIFIED WITH 'cachi
    AS '$A$005$|v>\ZKe^R...'
/*!*/;
```

Don’t worry if the binary log output is intimidating. It’s not intended for easy human consumption. However, you can see that the actual hash of the password, as it would appear in `mysql.user`, gets written to the binary log. We’ll talk more about this log in Chapter 10.

Modifying and Dropping Users

Creating a user usually isn’t the end of your interaction with it. You may later need to change its properties, perhaps to require an encrypted connection. It also happens that users need to be dropped. Neither of these operations is too different from user creation, but you need to know how to perform them in order to fully grasp user management.

Modifying a User

Any parameter that it's possible to set during user creation can also be altered at a later time. This is generally achieved using the `ALTER USER` command. MySQL 5.7 and before also have `RENAME USER` and `SET PASSWORD` shortcuts, while version 8.0 expanded that list to include `SET DEFAULT ROLE` (we'll cover the role system in "Roles"). Note that `ALTER USER` can be used to change everything about the user, and the other commands are just convenient ways to run common maintenance operations.

NOTE

We called `RENAME USER` a shortcut, but it's special in that it doesn't have a "full" `ALTER USER` alternative. As you'll see, the privileges required to run the `RENAME USER` command are also different, and are the same as for user creation (we'll talk more about privileges soon).

We will start with the regular `ALTER USER` command. In the first example, we're going to modify the authentication plugin used. A lot of older programs do not support the new and standard in MySQL 8.0 `caching_sha2_password` plugin and require you to either create the users using the older `mysql_native_password` plugin or alter them after creation to use that plugin. We can check the current plugin in use by querying one of the grant tables (see "Grant Tables" for more information on these):

```
mysql> SELECT plugin FROM mysql.user WHERE
    -> user = 'bob' AND host = 'localhost';


+-----------------------+
| plugin                |
+-----------------------+
| caching_sha2_password |
+-----------------------+
1 row in set (0.00 sec)
```

And now we can alter the plugin for this user and make sure the change is reflected:

```
mysql> ALTER USER 'bob'@'localhost' IDENTIFIED WITH mys
```

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

```
mysql> SELECT plugin FROM mysql.user WHERE
    -> user = 'bob' AND host = 'localhost';
```

```
+-----------------------+
| plugin                |
+-----------------------+
| mysql_native_password |
+-----------------------+
1 row in set (0.00 sec)
```

Since the change was made via an `ALTER` command, there's no need to run `FLUSH PRIVILEGES`. Once this command executes successfully, each new authentication attempt will use the new plugin. You could modify the user record directly to make this change, but again, we recommend against that.

The properties that `ALTER USER` can modify are pretty numerous and were explained or at least outlined in "Creating and Using New Users". There are, however, some frequently required operations that you should know a bit more about:

Changing a user's password

This is probably the single most frequent operation that's ever done on a user. Changing a user's password can be done by another user that has the necessary privileges, or by anyone authorized as that user, using a command like the following:

```
mysql> ALTER USER 'bob'@'localhost' IDENTIFIED by
```

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

This change takes effect immediately, so the next time this user authenticates they'll need to use the updated password. There's also a

`SET PASSWORD` shortcut for this command. It can be executed by the authenticated user without any target specification like this:

```
mysql> SET PASSWORD = 'new password';
```

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

Or it can be executed by another user, with the target specification.

```
mysql> SET PASSWORD FOR 'bob'@'localhost' = 'new
```

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

Locking and unlocking a user

If you need to temporarily (or permanently) block access to a specific user, you can do that using the `ACCOUNT LOCK` option of `ALTER USER`. The user in this case is only blocked for authentication. While nobody will be able to connect to MySQL as the blocked user, it can still be used both as a proxy and in a `DEFINER` clause. That makes such users slightly more secure and easier to manage. The `ACCOUNT LOCK` can also be used to, for example, block traffic from an application connecting as a specific user that is generating excessive load.

We can block `bob` from authenticating using the following command:

```
mysql> ALTER USER 'bob'@'localhost' ACCOUNT LOCK;
```

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

Only new connections will be affected. The message that MySQL produces in this case is clear:

```
$ mysql -ubob -p
```

```
Enter password:
ERROR 3118 (HY000): Access denied for user 'bob'@
Account is locked.
```

The counterpart to `ACCOUNT LOCK` is `ACCOUNT UNLOCK`. This option to `ALTER USER` does exactly what it says. Let's allow access to `bob` again:

```
mysql> ALTER USER 'bob'@'localhost' ACCOUNT UNLOC
```

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

Now the connection attempt will succeed:

```
$ mysql -ubob -p
```

```
Enter password:
```

```
mysql>
```

Expiring a user's password

Instead of blocking a user's access completely or changing the password for them, you may instead want to force them to change their password. That is possible in MySQL with the `PASSWORD EXPIRE` option of the `ALTER USER` command. After this command is executed, the user will still be able to connect to the server using the previous password. However, as soon as they run a query from the new connection—that is, as soon as their privileges are checked—the user will be presented with an error and forced to change the password. Existing connections are not affected.

Let's see what this looks like for the user. First, the actual alter:

```
mysql> ALTER USER 'bob'@'localhost' PASSWORD EXPI
```

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

Now, what the user gets. Note the successful authentication:

```
$ mysql -ubob -p

Enter password:

mysql> SELECT id, data FROM bobs_db.bobs_private_

ERROR 1820 (HY000): You must reset your password
statement before executing this statement.
```

Even though the error states you have to run `ALTER USER`, you now know that you can use `SET PASSWORD` instead. It also doesn't matter who changes the password: the user in question or another user. The `PASSWORD EXPIRE` option just forces the password change. If another user changes the password, then sessions authenticated with the old password after the password has been expired will need to be reopened. As we saw earlier, the authenticated user can change the password without a target specification, and they'll be able to continue with their session as normal (new connections, however, will need to be authenticated with the new password):

```
mysql> SET PASSWORD = 'new password';

Query OK, 0 rows affected (0.06 sec)

mysql> SELECT id, data FROM bobs_db.bobs_private_

Empty set (0.00 sec)
```

In this case, you should also be aware that without password reuse and history controls in place, the user could just reset the password to the original one.

Renaming a user

Changing a username is a relatively rare operation, but it is sometimes necessary. This operation has a special command: `RENAME USER`. It requires the `CREATE USER` privilege, or the `UPDATE` privilege on the `mysql` database or just the grant tables. There's no `ALTER USER` alternative for this command.

You can change both the "name" part of the username and the "host" part. Since, as you know by now, the "host" part acts as a firewall, be cautious when changing it, as you may cause outages (actually, the same is true of the "name" part as well). Let's rename our `bob` user to something more formal:

```
mysql> RENAME USER 'bob'@'localhost' TO 'robert'@
```

```
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

When the username changes, MySQL automatically scans through its internal tables to see whether that user has been named in the `DEFINER` clause of a view or a stored object. Whenever that is the case, a warning is produced. Since we did get a warning when we renamed `bob`, let's check it out:

```
mysql> SHOW WARNINGS\G
```

```
*************************** 1. row **************
  Level: Warning
   Code: 4005
Message: User 'bob'@'localhost' is referenced as
  account in a stored routine.
1 row in set (0.00 sec)
```

Failure to address this issue can potentially result in orphaned objects that will error out when accessed or executed. We discuss this in detail in the next section.

Dropping a User

The final part of the lifecycle of a database user is its end of life. Like any database object, users can be deleted. In MySQL, the `DROP USER` command is used to achieve that. This is one of the simplest commands discussed in this chapter, and potentially in the whole book. `DROP USER` takes a user or, optionally, a list of users as an argument, and has a single modifier: `IF NOT EXISTS`. Successful execution of the command irrevocably deletes the user-related information from the grant tables (with a caveat we'll discuss later) and thus prevents further logins.

When you drop a user that has made one or more connections to the database that are still open, even though the drop succeeds, the associated records will be removed only when the last of those connections ends. The next attempt to connect with the given user will result in the `ERROR 1045 (28000): Access denied` message.

The `IF NOT EXISTS` modifier works similarly to with `CREATE USER`: if the user you target with `DROP USER` does not exist, no error will be returned. This is useful in unattended scripts. If the host part of the username is not specified, the wildcard `%` is used by default.

In its most basic form, the `DROP USER` command looks like this:

```
mysql> DROP USER 'jeff'@'localhost';


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

Executing the same command again will result in an error:

```
mysql> DROP USER 'jeff'@'localhost';


ERROR 1396 (HY000): Operation DROP USER failed for 'jef
```

If you want to construct an idempotent command that won't fail, then use the following construct instead:

```
mysql> DROP USER IF EXISTS 'jeff'@'localhost';

Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;

+-------+------+----------------------------------------
| Level | Code | Message
+-------+------+----------------------------------------
| Note  | 3162 | Authorization ID 'jeff'@'localhost' dc
+-------+------+----------------------------------------
1 row in set (0.01 sec)
```

Again, if you do not specify the host part of the username, MySQL will assume the default `%`. It's also possible to drop multiple users at once:

```
mysql> DROP USER 'jeff', 'bob'@'192.168.%';

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

In MySQL, as users do not own objects, they can be dropped quite easily and without much preparation. However, as we have already discussed, users can fulfill extra roles. If the dropped user is used as a proxy user or is part of the `DEFINER` clause of some object, then dropping it can create an orphaned record. Whenever the user you drop is part of such a relationship, MySQL emits a warning. Note that the `DROP USER` command will still succeed, so it's up to you to resolve the resulting inconsistencies and fix any orphaned records:

```
mysql> DROP USER 'bob'@'localhost';

Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
mysql> SHOW WARNINGS\G


*************************** 1. row ******************
  Level: Warning
   Code: 4005
Message: User 'bob'@'localhost' is referenced as a defi
  account in a stored routine.
1 row in set (0.00 sec)
```

We recommend that you check this before actually dropping the user. If you fail to notice the warning and take action, the objects are left orphaned. Orphaned objects will produce errors when used:

```
mysql> CALL test.test_proc();


ERROR 1449 (HY000): The user specified as a definer ('b
  does not exist
```

For users in a proxy relationship, no warning is produced. However, a subsequent attempt to use the proxied user will result in an error. As proxy users are used in authentication, the end result will be the inability to log into MySQL with users dependent on the dropped one. This arguably can be more impactful than temporarily losing the ability to call a procedure or query a view, but still, no warning will be emitted. If you are using pluggable authentication relying on proxy users, remember this.

If you find yourself in a situation where you dropped a user and unexpectedly got a warning, you can easily create the user again to avoid the errors. Note how the following `CREATE USER` statement results in the now-familiar warning:

```
mysql> CREATE USER 'bob'@'localhost' IDENTIFIED BY 'new


Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
mysql> SHOW WARNINGS\G


*************************** 1. row ********************
  Level: Warning
   Code: 4005
Message: User 'bob'@'localhost' is referenced as a defi
  account in a stored routine.
1 row in set (0.00 sec)
```

However, the problem here is that if you don't know or remember the initial privileges of the account, the new one can become a security issue.

To identify orphaned records, you need to manually review MySQL's catalog tables. Specifically, you're going to need to look at the `DEFINER` column of the following tables:

```
mysql> SELECT table_schema, table_name FROM informatior
    -> WHERE column_name = 'DEFINER';
```

```
+--------------------+------------+
| TABLE_SCHEMA       | TABLE_NAME |
+--------------------+------------+
| information_schema | EVENTS     |
| information_schema | ROUTINES   |
| information_schema | TRIGGERS   |
| information_schema | VIEWS      |
+--------------------+------------+
```

Now that you know that, you can easily construct a query to check if a user you're going to drop or have already dropped is specified within any `DEFINER` clauses:

```
SELECT EVENT_SCHEMA AS obj_schema
    , EVENT_NAME obj_name
    , 'EVENT' AS obj_type
FROM INFORMATION_SCHEMA.EVENTS
WHERE DEFINER = 'bob@localhost'
UNION
SELECT ROUTINE_SCHEMA AS obj_schema
```

```
    , ROUTINE_NAME AS obj_name
    , ROUTINE_TYPE AS obj_type
FROM INFORMATION_SCHEMA.ROUTINES
WHERE DEFINER = 'bob@localhost'
UNION
SELECT TRIGGER_SCHEMA AS obj_schema
    , TRIGGER_NAME AS obj_name
    , 'TRIGGER' AS obj_type
FROM INFORMATION_SCHEMA.TRIGGERS
WHERE DEFINER = 'bob@localhost'
UNION
SELECT TABLE_SCHEMA AS obj_scmea
    , TABLE_NAME AS obj_name
    , 'VIEW' AS obj_type
FROM INFORMATION_SCHEMA.VIEWS
WHERE DEFINER = 'bob@localhost';
```

That query might look intimidating, but by now you should've seen `UNION` used in "The Union", and the query is just a union of four simple queries. Each individual query looks for an object with a `DEFINER` value of `bob@localhost` in one of the following tables: `EVENTS`, `ROUTINES`, `TRIGGERS`, and `VIEWS`.

In our example, the query returns a single record for `bob@localhost`:

```
+------------+-----------+-----------+
| obj_schema | obj_name  | obj_type  |
+------------+-----------+-----------+
| test       | test_proc | PROCEDURE |
+------------+-----------+-----------+
```

It's similarly easy to check if the proxy privilege was granted for this user:

```
mysql> SELECT user, host FROM mysql.proxies_priv
    -> WHERE proxied_user = 'bob'
    -> AND proxied_host = 'localhost';
```

```
+------+------+
| user | host |
+------+------+
| jeff | %    |
+------+------+
```

We recommend that you always check for possible orphaned objects and proxy privileges before you drop a particular user. Such gaps left in the database will not only cause obvious issues (errors) but are, in fact, a security risk.

Privileges

When a user connects to a MySQL server, authentication is performed using the username and host information, as explained before. However, the permissions the user has to perform different actions aren't checked before any commands are executed. MySQL grants privileges according to the identity of the connected user and the actions it performs. As discussed at the beginning of this chapter, privileges are sets of permissions to perform actions on various objects. By default, a user is not entitled to any permissions, and thus it has no privileges assigned after `CREATE USER` is executed.

There are a lot of privileges that you can grant to a user and later revoke. For example, you can allow a user to read from a table or modify data in it. You can grant a privilege to create tables and databases, and another to create stored procedures. The list is vast. Curiously, you will not find a connection privilege anywhere: it's impossible to disallow a user from connecting to MySQL, assuming the host part of username matches. That's a direct consequence of what was outlined in the previous paragraph: privileges are checked only when an action is performed, so by nature they will apply only once a user is authenticated.

To get a full list of privileges supported and provided by your MySQL installation, we always recommend checking the manual. However, we'll cover the few broad categories of privileges here. We'll also talk about levels of privileges, as the same privileges can be allowed on multiple levels. Actually, that's what we'll start with. There are four levels of privileges in MySQL:

Global privileges

These privileges allow the *grantee* (the user who is granted the privilege—we cover the `GRANT` command in "Privilege Management Commands") to either act on every object in every database or act on the cluster as a whole. The latter applies to commands that are usually

considered administrative. For example, you can allow a user to shut down the cluster.

Privileges in this category are stored within the tables `mysql.user` and `mysql.global_grants`. The first one stores conventional static privileges, and the second one stores dynamic privileges. The difference is explained in the following section. MySQL versions prior to 8.0 store all global privileges in `mysql.user`.

Database privileges

Privileges granted on a database level will allow the user to act upon objects within that database. As you can imagine, the list of privileges is narrower at this level, since there's little sense in breaking down the `SHUTDOWN` privilege below the global level, for example. Records for these privileges are stored within the `mysql.db` table and include the ability to run DDL and DML queries within the target database.

Object privileges

A logical continuation of database-level privileges, these target a particular object. Tracked in `mysql.tables_priv`, `mysql.procs_priv`, and `mysql.proxies_priv`, they respectively cover tables and views, all types of stored routines, and finally the proxy user permissions. Proxy privileges are special, but the other privileges in this category are again regular DDL and DML permissions.

Column privileges

Stored in `mysql.columns_priv`, these are an interesting set of privileges. You can separate permissions within a particular table by column. For example, a reporting user may not have the need to read the `password` column of a particular table. This is a powerful tool, but column privileges can be difficult to manage and maintain.

The complete list of privileges, frankly, is very long. It is always advisable to consult the MySQL documentation for your particular version for the complete details. You should remember that any action a user can perform either will have a dedicated privilege assigned or will be covered by a privilege controlling a wider range of behaviors. In general, database- and object-level privileges will have a dedicated privilege name that you can grant (`UPDATE`, `SELECT`, and so on), and global privileges will be quite broadly

grouped together, allowing multiple actions at once. For example, the `GROUP_REPLICATION_ADMIN` privilege allows five different actions at once. Global privileges will also usually be granted on a system level (the `*.*` object).

You can always access the list of privileges available in your MySQL instance by running the `SHOW PRIVILEGES` command:

```
mysql> SHOW PRIVILEGES;
```

```
+---------------------------+-----------------------+--
| Privilege                 | Context               | C
+---------------------------+-----------------------+--
| Alter                     | Tables                | T
| Alter routine             | Functions,Procedures  | .
| ...
| REPLICATION_SLAVE_ADMIN   | Server Admin          |
| AUDIT_ADMIN               | Server Admin          |
+---------------------------+-----------------------+--
58 rows in set (0.00 sec)
```

Static Versus Dynamic Privileges

Before we go on to review the commands used to manage privileges in MySQL, we must pause and talk about an important distinction. There are two types of privileges in MySQL, starting with version 8.0: static and dynamic. The *static* privileges are built into the server, and every installation will have them available and usable. The *dynamic* privileges are, on the other hand, "volatile": they are not guaranteed to be present all the time.

What are these dynamic privileges? They are privileges that are registered within the server at runtime. Only registered privileges can be granted, so it is possible that some privileges will never be registered and will never be grantable. All of that is a fancy way of saying that it's now possible to extend privileges via plugins and components. However, most of the dynamic privileges currently available are registered by default in a regular Community Server installation.

The important role of the dynamic privileges provided with MySQL 8.0 is that they are aimed at reducing the necessity of using the `SUPER` privilege, which was previously abused (we'll talk about this privilege in the next section). The other distinction of dynamic privileges is that they usually control a set of activities users can perform. For example, unlike a direct `SELECT` privilege on a table, which just allows querying the data, the `CONNECTION_ADMIN` privilege allows a whole list of actions. In this particular example, that includes killing other accounts' connections, updating data in a read-only server, connecting through an extra connection when the limit is reached, and more. You can easily spot the difference.

The SUPER Privilege

This section is not long, but it is important. We mentioned in "The root User" that there's a superuser created by default with every MySQL installation: `root@localhost`. Sometimes you might want to provide the same capabilities to another user, for example one used by a DBA. The convenient built-in way of doing so is by using the special `SUPER` privilege.

`SUPER` is basically a catchall privilege that turns a user to which it is assigned into a superuser. As with any privilege, it can be assigned via a `GRANT` statement, which we'll review in the following section.

There are two huge problems with the `SUPER` privilege, however. First, starting with MySQL 8.0 it is deprecated, and it is going to be removed in a future release of MySQL. Second, it is a security and operational nightmare. The first point is clear, so let's talk about the second one, and about the alternatives we have.

Using the `SUPER` privilege poses the same risks and results in the same issues as using the default `root@localhost` user. Instead of carefully inspecting the scope of privileges required, we're resorting to using an all-purpose hammer to solve all problems. The main problem with `SUPER` is its all-encompassing scope. When you create a superuser, you create a liability: the user must be restricted and ideally audited, and operators and programs authenticating as the user must be extremely precise and careful in their actions. With great power comes great responsibility—and, among other things, the ability to just outright shut down the MySQL instance. Imagine executing that by mistake.

In MySQL versions before 8.0, it's not feasible to avoid using the `SUPER` privilege, as there are no alternatives provided for some of the permissions. Starting with version 8.0, which deprecates `SUPER`, MySQL provides a whole set of dynamic privileges that are aimed at removing the need for the single catchall privilege. You should try to avoid using the `SUPER` privilege, if possible.

Consider the example of a user that needs to control group replication. In MySQL 5.7, you would need to grant the `SUPER` privilege to that user. Starting with version 8.0, however, you can instead grant the special `GROUP_REPLICATION_ADMIN` privilege, which only allows users to perform a very small subset of actions related to group replication.

Sometimes, you will still need a full-on DBA user that can do anything. Instead of granting `SUPER`, consider looking at the `root@localhost` privileges and copying them instead. We show you how to do that in "Checking Privileges". Taking this further, you can skip granting some of the privileges, such as the `INNODB_REDO_LOG_ENABLE` privilege, which authorizes a user to basically enable a crash-unsafe mode. It is much safer to not have that privilege granted at all, and to be able to grant it to yourself when absolutely required, than to open up the risk of someone running that statement by mistake.

Privilege Management Commands

Now that you know a bit about privileges, we can proceed to the basic commands that allow you to control them. You can never `ALTER` a privilege itself, though, so by controlling privileges here we mean giving them to and removing them from users. These actions are achieved with the `GRANT` and `REVOKE` statements.

GRANT

The `GRANT` statement is used to grant users (or roles) permissions to perform activities, either in general or on specific objects. The same statement can also be used to assign roles to users, but you cannot at the same time alter permissions and assign roles. To be able to grant a permission (privilege), you need to have that privilege assigned yourself and have the special `GRANT OPTION` privilege (we'll review that later). Users with the `SUPER` (or newer

`CONNECTION_ADMIN`) privilege can grant anything, and there's a special condition related to grant tables, which we'll discuss shortly.

For now, let's check out the basic structure of a `GRANT` statement:

```
mysql> GRANT SELECT ON app_db.* TO 'john'@'192.168.%';
```

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

That statement, once executed, tells MySQL that user `'john'@'192.168.%'` is allowed to perform read-only queries (`SELECT`) on any table in the `app_db` database. Note that we have used a wildcard in the object specification. We could allow a particular user access to every table of every database by specifying a wildcard for the database as well:

```
mysql> GRANT SELECT ON *.* TO 'john';
```

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

The preceding invocation notably lacks the host specification for the user `'john'`. This shortcut translates to `'john'@'%'`; thus, it will not be the same user as the `'john'@'192.168.%'` we used before. Speaking of wildcards and users, it is not possible to specify a wildcard for the username portion of the user. Instead, you can specify multiple users or roles in one go like this:

```
mysql> GRANT SELECT ON app_db.* TO 'john'@'192.168.%',
    -> 'kate'@'192.168.%';
```

```
Query OK, 0 row affected (0.06 sec)
```

We cautioned you about granting too many privileges in the previous section, but it can be useful to remember that there's an `ALL` shortcut that allows you to grant every possible privilege on an object or set of objects. That can come

handy when you define permissions for the “owner” user—for example, a read/write application user:

```
mysql> GRANT ALL ON app_db.* TO 'app_db_user';

Query OK, 0 row affected (0.06 sec)
```

You cannot chain object specifications, so you won’t be able to grant the `SELECT` privilege on two tables at once, unless that statement can be expressed using wildcards. As you’ll see in the next section, you can combine wildcard grants and specific revokes for extra flexibility.

An interesting property of the `GRANT` command is that it doesn’t check for the presence of the objects that you allow. That is, a wildcard is not expanded, but stays a wildcard forever. No matter how many new tables are added to the `app_db` database, both `john` and `kate` will be able to issue `SELECT` statements on them. Earlier versions of MySQL also would create a user to whom privileges were granted if it wasn’t found in the `mysql.user` table, but that behavior is deprecated starting with MySQL 8.0.

As we discussed in depth in “Grant Tables” the `GRANT` statement updates grant tables. One thing that follows from the fact that there’s an update on grant tables is that if a user has the `UPDATE` privilege on those tables, that user can grant any account any privilege. Be extremely careful with permissions on objects in the `mysql` database: there’s little benefit to granting users any privileges there. Note also that when the `read_only` system variable is enabled, any grant requires super user privileges (`SUPER` or `CONNECTION_ADMIN`).

There are a few other points we’d like to make about `GRANT` before moving on. In the introduction to this section, we mentioned column privileges. This set of privileges controls whether a user can read and update data in a particular column of a table. Like all other privileges, they can be permitted using the `GRANT` command:

```
mysql> GRANT SELECT(id), INSERT(id, data)
    -> ON bobs_db.bobs_private_table TO 'kate'@'192.168
```

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

The user `kate` will now be able to issue the statement `SELECT id FROM bobs_db.bobs_private_table`, but not `SELECT *` or `SELECT data`.

Finally, you can grant every static privilege on a particular object, or globally, by running `GRANT ALL PRIVILEGES` instead of specifying each privilege in turn. `ALL PRIVILEGES` is just a shorthand and is not itself a special privilege, unlike `SUPER`, for example.

REVOKE

The `REVOKE` statement is the opposite of the `GRANT` statement: you can use it to revoke privileges and roles assigned using `GRANT`. Unless otherwise specified, every property of `GRANT` applies to `REVOKE`. For example, to revoke privileges you need to have the `GRANT OPTION` privilege and the particular privileges that you are revoking.

Starting with MySQL version 8.0.16, it's possible to revoke privileges for particular schemas from users that have privileges granted globally. That makes it possible to easily restrict access to some databases while allowing access to all others, including ones that are newly created. For example, consider a database system where you have a single restricted schema. You need to create a user for your BI application. You start by running the usual command:

```
mysql> GRANT SELECT ON *.* TO 'bi_app_user';
```

```
Query OK, 0 rows affected (0.03 sec)
```

However, this user has to be forbidden from querying any data in the restricted database. This is extremely easy to set up using partial revokes:

```
mysql> REVOKE SELECT ON restricted_database.* FROM 'bi_
```

```
Query OK, 0 rows affected (0.03 sec)
```

Before version 8.0.16 to achieve this you would need to fall back to explicitly running `GRANT SELECT` for each individual allowed schema.

Just as you can grant all privileges, a special invocation of `REVOKE` exists that allows the removal of all privileges from a particular user. Remember that you need to have all the privileges you are revoking, and thus this option is likely to be used only by an administrative user. The following statement will strip a user of their privileges, including the ability to assign any privileges:

```
mysql> REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'john'@
```

```
Query OK, 0 rows affected (0.03 sec)
```

The `REVOKE` statement doesn't under any circumstances remove the user itself. You can use the `DROP USER` statement for that, as described earlier in this chapter.

Checking Privileges

An important part of managing privileges is reviewing them—but it would be impossible to remember every privilege granted to every user. You can query the grant tables to see what users have what privileges, but that's not always convenient. (It is still an option, however, and it can be a good way to find, for example, every user that has write privileges on a certain table.) The more straightforward option for viewing the privileges granted to a particular user is to use the built-in `SHOW GRANTS` command. Let's take a look at it:

```
mysql> SHOW GRANTS FOR 'john'@'192.168.%';
```

```
+-----------------------------------------------+
| Grants for john@192.168.%                     |
+-----------------------------------------------+
| GRANT UPDATE ON *.* TO `john`@`192.168.%`     |
| GRANT SELECT ON `sakila`.* TO `john`@`192.168.%` |
+-----------------------------------------------+
2 rows in set (0.00 sec)
```

In general, you can expect to see every privilege in this output, but there's a special case. When a user has every static privilege granted for a particular object, instead of listing each and every one of them, MySQL will output `ALL PRIVILEGES` instead. This is not a special privilege itself, but rather a shorthand for every possible privilege. Internally, `ALL PRIVILEGES` just translates to `Y` set for every privilege in the respective grant table:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';


+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
...
| GRANT ALL PRIVILEGES ON `bobs_db`.* TO `bob`@`localhc
...
```

You can also view the permissions granted to roles using the `SHOW GRANTS` command, but we'll talk about that in more detail in "Roles". To review the permissions of the currently authenticated and authorized user, you can use any of the following statements, which are synonymous:

```
SHOW GRANTS;
SHOW GRANTS FOR CURRENT_USER;
SHOW GRANTS FOR CURRENT_USER();
```

Whenever you do not remember what a specific privilege means, you can either consult the documentation or run the `SHOW PRIVILEGES` command, which will list every privilege currently available. That covers both static object privileges and dynamic server privileges.

Sometimes you might need to review privileges related to all accounts or transfer those privileges to another system. One option that you have is to use the `mysqldump` command provided with MySQL Server for all supported platforms. We will be reviewing that command in detail in Chapter 10. In short, you'll need to dump all of the grant tables, as otherwise you might miss some of the permissions. The safest way to go is to dump all of the data in the `mysql` database:

```
$ mysqldump -uroot -p mysql

Enter password:
```

The output will include all the table definitions, along with a lot of `INSERT` statements. This output can be redirected to a file and then used to seed a new database. We talk more about that in Chapter 10. If your server versions don't match or the target server already has some users and privileges stored, it might be best to avoid dropping the existing objects. Add the `--no-create-info` option to the `mysqldump` invocation to only receive the `INSERT` statements.

By using `mysqldump` you get a portable list of users and privileges, but it's not exactly easily readable. Here is an example of some of the rows in the output:

```
--
-- Dumping data for table `tables_priv`
--

LOCK TABLES `tables_priv` WRITE;
/*!40000 ALTER TABLE `tables_priv` DISABLE KEYS */;
INSERT INTO `tables_priv` VALUES ('172.%','sakila','rob
'Select,Insert,Update,Delete,Create,Drop,Grant,Referenc
('localhost','sys','mysql.sys','sys_config','root@local
    '2020-07-13 07:14:57','Select','');
/*!40000 ALTER TABLE `tables_priv` ENABLE KEYS */;
UNLOCK TABLES;
```

Another option to review the privileges would be to write custom queries over grant tables, as already mentioned. We won't give any guidelines on that, as there's no one-size-fits-all solution.

Yet another way is by running `SHOW GRANTS` for every user in the database. By combining that with the `SHOW CREATE USER` statement, you can generate the list of privileges, which can also be used to re-create the users and their privileges in another database:

```
mysql> SELECT CONCAT("SHOW GRANTS FOR `", user, "`@`",
    -> "`; SHOW CREATE USER `", user, "`@`", host, "`;'
    -> FROM mysql.user WHERE user = "bob";
```

```
+-----------------------------------------------------
| grants
+-----------------------------------------------------
| SHOW GRANTS FOR bob@%; SHOW CREATE USER bob@%;
| SHOW GRANTS FOR bob@localhost; SHOW CREATE USER bob@l
+-----------------------------------------------------
2 rows in set (0.00 sec)
```

As you can imagine, the idea of automating this procedure is not new. In fact, there's a tool in Percona Toolkit— `pt-show-grants` —that does exactly that, and more. Unfortunately, the tool can be used only on Linux officially and might not work at all on any other platform:

```
$ pt-show-grants
```

```
-- Grants dumped by pt-show-grants
-- Dumped from server Localhost via Unix socket,
    MySQL 8.0.22 at 2020-12-12 14:32:33
-- Roles
CREATE ROLE IF NOT EXISTS `application_ro`;
-- End of roles listing
...
-- Grants for 'robert'@'172.%'
CREATE USER IF NOT EXISTS 'robert'@'172.%';
ALTER USER 'robert'@'172.%' IDENTIFIED WITH 'mysql_nati
AS '*E1206987C3E6057289D6C3208EACFC1FA0F2FA56' REQUIRE
PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK PASSWORD HISTORY
PASSWORD REUSE INTERVAL DEFAULT PASSWORD REQUIRE CURREN
GRANT ALL PRIVILEGES ON `bobs_db`.* TO `robert`@`172.%`
GRANT ALL PRIVILEGES ON `sakila`.`actor` TO `robert`@`1
GRANT SELECT ON `sakila`.* TO `robert`@`172.%` WITH GRA
GRANT SELECT ON `test`.* TO `robert`@`172.%` WITH GRANT
GRANT USAGE ON *.* TO `robert`@`172.%`;
...
```

The GRANT OPTION Privilege

As discussed at the beginning of this chapter, MySQL does not have a concept of object ownership. Thus, unlike in some other systems, the fact that some user created a table does not automatically mean that the same user can allow another user to do anything with that table. To make this slightly less convoluted, let's review an example.

Suppose the user `bob` has permissions to create tables in a database called `bobs_db`:

```
mysql> CREATE TABLE bobs_db.bobs_private_table
    -> (id SERIAL PRIMARY KEY, data TEXT);


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

An operator using the `bob` user wants to allow the `john` user to read data in the newly created table—but, alas, that is not possible:

```
mysql> GRANT SELECT ON bobs_db.bobs_private_table TO '
```

```
ERROR 1142 (42000): SELECT, GRANT command denied
to user 'bob'@'localhost' for table 'bobs_private_table
```

Let's check what privileges `bob` actually has:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';


+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT SELECT ON `sakila`.* TO `bob`@`localhost`
| GRANT ALL PRIVILEGES ON `bobs_db`.* TO `bob`@`localhc
+-----------------------------------------------------
3 rows in set (0.00 sec)
```

The missing piece here is a privilege that would allow a user to grant other users privileges it has been granted. If a DBA wants to allow `bob` to grant other users access to tables in the `bobs_db` database, an extra privilege needs to be granted. `bob` can't grant that to itself, so a user with administrative privileges is required:

```
mysql> GRANT SELECT ON bobs_db.* TO 'bob'@'localhost'
    -> WITH GRANT OPTION;
```

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

Note the `WITH GRANT OPTION` addition. That's exactly the privilege that we were looking for. This option will allow the `bob` user to pass its privileges to other users. Let's confirm that by running the `GRANT SELECT` statement as `bob` again:

```
mysql> GRANT SELECT ON bobs_db.bobs_private_table TO '
```

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

As expected, the statement was accepted and executed. There are still few clarifications to make, however. First, we may want to know how granular the `GRANT OPTION` privilege is. That is, what exactly (apart from `SELECT` on `bobs_private_table`) can `bob` grant to other users? `SHOW GRANTS` can answer that question for us neatly:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';
```

```
+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT SELECT ON `sakila`.* TO `bob`@`localhost`
| GRANT ALL PRIVILEGES ON `bobs_db`.* TO `bob`@`localhc
+-----------------------------------------------------
3 rows in set (0.00 sec)
```

That's much clearer. We can see that `WITH GRANT OPTION` is applied to privileges that `bob` has on a particular database. That's important to remember. Even though we executed `GRANT SELECT ... WITH GRANT OPTION`, `bob` got the ability to grant every privilege it has in the `bobs_db` database.

Second, we may want to know if it is possible to revoke just the `GRANT OPTION` privilege:

```
mysql> REVOKE GRANT OPTION ON bobs_db.* FROM 'bob'@'loc
```

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

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';
```

```
+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT SELECT ON `sakila`.* TO `bob`@`localhost`
| GRANT ALL PRIVILEGES ON `bobs_db`.* TO `bob`@`localhc
+-----------------------------------------------------
3 rows in set (0.00 sec)
```

Finally, looking at how `GRANT OPTION` can be revoked, we may want to know whether it can be granted alone. The answer is yes, with a caveat that we'll show. Let's grant the `GRANT OPTION` privilege to `bob` on the `sakila` and `test` databases. As you can see from the preceding output, `bob` currently has the `SELECT` privilege on `sakila`, but no privileges on the `test` database:

```
mysql> GRANT GRANT OPTION ON sakila.* TO 'bob'@'localhc
```

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

```
mysql> GRANT GRANT OPTION ON test.* TO 'bob'@'localhost
```

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

Both statements succeeded. It's pretty clear what exactly `bob` can grant on `sakila`: the `SELECT` privilege. However, it's less clear what happened with `test`. Let's check it out:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';
```

```
+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT SELECT ON `sakila`.* TO `bob`@`localhost` WITH
| GRANT USAGE ON `test`.* TO `bob`@`localhost` WITH GRA
| GRANT ALL PRIVILEGES ON `bobs_db`.* TO `bob`@`localhc
+-----------------------------------------------------
4 rows in set (0.00 sec)
```

Okay, so `GRANT OPTION` alone only gives the user a `USAGE` privilege, which is the "no privileges" specifier. However, the `GRANT OPTION` privilege can be seen as a switch, and when "turned on," it'll apply for privileges `bob` is granted in the `test` database:

```
mysql> GRANT SELECT ON test.* TO 'bob'@'localhost';
```

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

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';
```

```
+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
...
| GRANT SELECT ON `test`.* TO `bob`@`localhost` WITH GR
```

```
...
+-----------------------------------------------------
4 rows in set (0.00 sec)
```

So far we've been using wildcard privileges, but it is possible to enable `GRANT OPTION` for a specific table:

```
mysql> GRANT INSERT ON sakila.actor
    -> TO 'bob'@'localhost' WITH GRANT OPTION;


Query OK, 0 rows affected (0.01 sec)


mysql> SHOW GRANTS FOR 'bob'@'localhost';


+-----------------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------------
...
| GRANT INSERT ON `sakila`.`actor` TO `bob`@`localhost`
+-----------------------------------------------------
5 rows in set (0.00 sec)
```

By now, it should be clear that `GRANT OPTION` is a powerful addition to the privileges system. Given that MySQL lacks the concept of ownership, it's the only way to make sure users that aren't superusers can grant each other permissions. However, it is also important, as always, to remember that `GRANT OPTION` applies to every permission the user has.

Roles

Roles, introduced in MySQL 8.0, are collections of privileges. They simplify user and privilege management by grouping and "containerizing" necessary permissions. You may have a few different DBA users that all need the same permissions. Instead of granting privileges individually to each of the users, you can create a role, grant permissions to that role, and assign users that role. In doing so, you also simplify management in that you won't need to update

each user individually. Should your DBAs need their privileges adjusted, you can simply adjust the role.

Roles are quite similar to users in how they are created, stored, and managed. To create a role, you need to execute a `CREATE ROLE [IF NOT EXISTS]` *`role1`* [, *`role2`* [, *`role3`* …]] statement. To remove a role, you execute a `DROP ROLE [IF EXISTS]` *`role1`* [, *`role2`* [, *`role3`* …]] statement. When you drop a role, the assignments of that role to all users is removed. Privileges required to create a role are `CREATE ROLE` or `CREATE USER`. To drop a role, the `DROP ROLE` or `DROP USER` privilege is required. As with the user management commands, if `read_only` is set, an admin privilege is additionally required to create and drop roles. Direct modification privileges on grant tables allow a user to modify anything, as we've also discussed.

Just like usernames, role names consist of two parts: the name itself and the host specification. When host is not specified, the `%` wildcard is assumed. The host specification for a role does not limit its use in any way. The reason it's there is because roles are stored just like users in the `mysql.user` grant table. As a consequence, you cannot have the same *`rolename`*@*`host`* as an existing user. To have a role with the same name as an existing user, specify a different hostname for the role.

Unlike privileges, roles are not active all the time. When a user is granted a role, they're authorized to use that role but not obliged to do so. In fact, a user can have multiple roles and can "enable" one or more of them within the same connection.

One or more roles can be assigned as defaults to a user during the user's creation or at a later time through the `ALTER USER` command. Such roles will be active as soon as the user is authenticated.

Let's review the commands, settings, and terminology related to role management:

`GRANT PRIVILEGE` and `REVOKE PRIVILEGE` commands

We covered these commands in "Privilege Management Commands". For all intents and purposes, roles can be used just the same as users with the `GRANT` and `REVOKE PRIVILEGE` commands. That is, you can assign all the same privileges to a role as you can to a user, and revoke them, too.

GRANT role [, role ...] TO user` command

The basic command related to role management. By running this command, you authorize a user to assume a particular role. As mentioned previously, the user is not obliged to use the role. Let's create a couple of roles that will be able to operate on the `sakila` database:

```
mysql> CREATE ROLE 'application_rw';

Query OK, 0 rows affected (0.01 sec)

mysql> CREATE ROLE 'application_ro';

Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL ON sakila.* TO 'application_rw';

Query OK, 0 rows affected (0.06 sec)

mysql> GRANT SELECT ON sakila.* TO 'application_r

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

Now you can assign these roles to an arbitrary number of users and change the roles only when needed. Here, we allow our `bob` user read-only access to the `sakila` database:

```
mysql> GRANT 'application_ro' TO 'bob'@'localhost

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

You can also grant more than one role in a single statement.

`WITH ADMIN OPTION` modifier

When you grant a role to a user, that user is allowed only to activate the role, but not to alter it in any way. That user may not grant the role to any other user. If you wish to allow both modification of the role and the ability to grant it to other users, you can specify `WITH ADMIN OPTION` in the `GRANT ROLE` command. The result will be reflected in grant tables and will be visible in the `SHOW GRANTS` command's output:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost';


+-----------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT `application_ro`@`%` TO `bob`@`localhost`
+-----------------------------------------------
2 rows in set (0.00 sec)
```

`SHOW GRANTS` and roles

The `SHOW GRANTS` command, which we introduced in "Checking Privileges", is capable of showing you both assigned roles and the effective permissions with one or more roles activated. This is possible by adding an optional `USING` *`role`* modifier. Here, we show the effective privileges that `bob` will have as soon as the `application_ro` role is activated:

```
mysql> SHOW GRANTS FOR 'bob'@'localhost' USING 'a


+-----------------------------------------------
| Grants for bob@localhost
+-----------------------------------------------
| GRANT USAGE ON *.* TO `bob`@`localhost`
| GRANT SELECT ON `sakila`.* TO `bob`@`localhost`
| GRANT `application_ro`@`%` TO `bob`@`localhost`
+-----------------------------------------------
3 rows in set (0.00 sec)
```

SET ROLE DEFAULT | NONE | ALL | ALL EXCEPT role [, role1 ...] | role [, role1 ...] command

The `SET ROLE` role management command is invoked by an authenticated user to assign a particular role or roles to itself. Once the role is set, its permissions apply to the user. Let's continue with our example for `bob` :

```
$ mysql -ubob
```

```
mysql> SELECT staff_id, first_name FROM sakila.st
```

```
ERROR 1142 (42000): SELECT command denied to user
table 'staff'
```

```
mysql> SET ROLE 'application_rw';
```

```
ERROR 3530 (HY000): `application_rw`@`%` is not g
`bob`@`localhost`
```

```
mysql> SET ROLE 'application_ro';
```

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

```
mysql> SELECT staff_id, first_name FROM sakila.st
```

```
+----------+------------+
| staff_id | first_name |
+----------+------------+
|        1 | Mike       |
|        2 | Jon        |
+----------+------------+
2 rows in set (0.00 sec)
```

Only when the role is assigned can `bob` use its privileges. Note that you cannot use `SET ROLE` to assign yourself a role you aren't authorized (through `GRANT ROLE`) to use.

There's no `UNSET ROLE` command, but there are few other extensions to `SET ROLE` that allow this behavior. To unset every role, run `SET ROLE NONE`. A user can also go back to its default set of roles by executing `SET ROLE DEFAULT`, or activate all the roles it has access to by running `SET ROLE ALL`. If you need to set a subset of roles which is neither default nor all, you can construct a `SET ROLE ALL EXCEPT` *`role`* [, *`role1`* ...] statement and explicitly avoid setting one or more roles.

`DEFAULT ROLE` user option

When you run `CREATE USER`, or later through `ALTER USER`, you can set one or more roles to be the default for a particular user. These roles will be implicitly set once the user is authenticated, saving you a `SET ROLE` statement. This is convenient, for example, for application users that use a single role or a known set of roles most of the time. Let's set `application_ro` as a default role for `bob`:

```
$ mysql -uroot
```

```
mysql> ALTER USER 'bob'@'localhost' DEFAULT ROLE
```

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

```
$ mysql -ubob
```

```
mysql> SELECT CURRENT_ROLE();
```

```
+----------------------+
| CURRENT_ROLE()       |
+----------------------+
| `application_ro`@`%` |
+----------------------+
1 row in set (0.00 sec)
```

As soon as `bob@localhost` is logged in, the `CURRENT_ROLE()` function returns the desired `application_ro` .

Mandatory roles

It is possible to grant one or more roles to every user in the database implicitly. This is achieved by setting the `mandatory_roles` system parameter (global in scope, and dynamic) to a list of roles. Roles granted this way are not activated until `SET ROLE` is run. It's impossible to revoke roles assigned this way, but you can grant them explicitly to a user. Roles listed in `mandatory_roles` cannot be dropped until removed from the setting.

Automatically activating roles

By default, roles are not active until `SET ROLE` is executed. However, it is possible to override that behavior and automatically activate every role available to a user upon authentication. This is analogous to running `SET ROLE ALL` upon login. This behavior can be enabled or disabled (which is the default) by changing the `activate_all_roles_on_login` system parameter (global in scope, and dynamic). When `activate_all_roles_on_login` is set to `ON` , both explicitly and implicitly (through `mandatory_roles`) granted roles will be activated for every user.

Cascading role permissions

Roles can be granted to roles. What happens then is that all permissions of the granted role are inherited by the grantee role. Once the grantee role is activated by a user, you can think of that user as having activated a granted role. Let's make our example slightly more complex. We will have an `application` role that is granted the `application_ro` and `application_rw` roles. The `application` role itself has no direct permissions assign. We will assigned the `application` role to our `bob` user and examine the result:

```
mysql> CREATE ROLE 'application';
```

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

```
mysql> GRANT 'application_rw', 'application_ro' T
```

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

```
mysql> REVOKE 'application_ro' FROM 'bob'@'localh
```

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

```
mysql> GRANT 'application' TO 'bob'@'localhost';
```

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

What happens now is that when `bob` activates the `application` role, it will have the permissions of both the `rw` and `ro` roles. We can easily verify this. Note that `bob` cannot activate any of the roles it was granted indirectly:

```
$ mysql -ubob
```

```
mysql> SET ROLE 'application';
```

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

```
mysql> SELECT staff_id, first_name FROM sakila.st
```

```
+----------+------------+
| staff_id | first_name |
+----------+------------+
|        1 | Mike       |
|        2 | Jon        |
+----------+------------+
2 rows in set (0.00 sec)
```

Roles graph

Since roles can be granted to roles, the resulting hierarchy can be pretty hard to follow. You can review it by examining the `mysql.role_edges` grant table:

```
mysql> SELECT * FROM mysql.role_edges;
```

```
+-----------+----------------+-----------+-------
| FROM_HOST | FROM_USER      | TO_HOST   | TO_USE
+-----------+----------------+-----------+-------
| %         | application    | localhost | bob
| %         | application_ro | %         | applic
| %         | application_rw | %         | applic
| %         | developer      | 192.168.% | john
| %         | developer      | localhost | bob
| 192.168.% | john           | %         | develo
+-----------+----------------+-----------+-------
6 rows in set (0.00 sec)
```

For more complex hierarchies, MySQL conveniently includes a built-in function that allows you to generate an XML document in a valid GraphML format. You can use any capable software to visualize the output. Here's the function call, and the resulting heavily formatted output (XML doesn't work well in books):

```
mysql> SELECT * FROM mysql.roles_graphml()\G
```

```
*************************** 1. row *************
roles_graphml(): <?xml version="1.0" encoding="UT
<graphml xmlns="...
...
    <node id="n0">
      <data key="key1">`application`@`%`</data>
    </node>
    <node id="n1">
      <data key="key1">`application_ro`@`%`</data
    </node>
...
```

Ideally, you should use `SELECT ... INTO OUTFILE` (see "Writing Data into Comma-Delimited Files"). Then you can use a tool such as the yEd graph editor, which is a powerful, cross-platform, free desktop application, to visualize that output. You can see a zoomed-in section of the complete graph, concentrating on our `bob` user and surrounding roles, in Figure 8-1. The privilege required to run this function is `ROLE_ADMIN`.

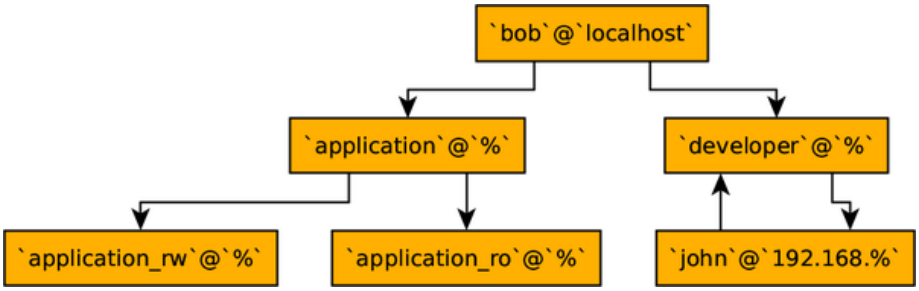

Figure 8-1. Section of a visualized MySQL roles graph

Differences between roles and users

Earlier we mentioned that the `CREATE USER` and `DROP USER` privileges allow modification of roles. Given that roles are stored along with users in `mysql.user`, you might also guess that the regular user management commands will work for roles. That's easy to test and confirm: just run `RENAME USER` or a `DROP USER` on a role. Another thing to note is how the `GRANT` and `REVOKE PRIVILEGE` commands target roles as if they were users.

Roles are, at their core, just regular users. In fact, it is possible to use `GRANT ROLE` to grant an unlocked user to another unlocked user or to a role:

```
mysql> CREATE ROLE 'developer';

Query OK, 0 rows affected (0.02 sec)

mysql> GRANT 'john'@'192.168.%' TO 'developer';

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

```
mysql> SELECT from_user, to_user FROM mysql.role_
```

```
+-----------+-----------+
| from_user | to_user   |
+-----------+-----------+
| john      | developer |
+-----------+-----------+
1 row in set (0.00 sec)
```

Roles are a powerful and flexible addition to MySQL's user and privilege system. As with almost any feature, they can be overused, resulting in unnecessarily complex hierarchies that will be hard to follow. However, if you keep things simple, roles can save you a lot of work.

Changing root's Password and Insecure Startup

Sometimes, it can become necessary to gain access to a MySQL instance without knowing any user's password. Or you could accidentally drop every user in the database, effectively locking you out. MySQL provides a workaround in such situations, but requires you to be able to change its configuration and restart the instance in question. You might think this is shady or dangerous, but actually it's just a protection from one of the simplest problems DBAs run up against: forgotten passwords. Just imagine having a production instance running that has no superuser access available: that's obviously not something desirable. Luckily, it's possible to bypass authorization when necessary.

To perform the authentication and privileges bypass, you have to restart a MySQL instance with the `--skip-grant-tables` option specified. Since most installations use service scripts to start the instance, you can specify `skip-grant-tables` in the *my.cnf* configuration file under the `[mysqld]` section. When `mysqld` is started in this mode, it (pretty obviously) skips reading grant tables, which has the following effects:

- No authentication is performed; thus, there's no need to know any usernames or passwords.
- No privileges are loaded, and no permissions are checked.
- MySQL will implicitly set `--skip-networking` to prevent any but local access while it's running in the unsafe configuration.

When you connect to a MySQL instance running with `--skip-grant-tables`, you'll be authorized as a special user. This user has access to every table and can alter any user. Before altering, for example, the `root` user's lost password, you need to run `FLUSH PRIVILEGES`; otherwise, the `ALTER` will fail:

```
mysql> SELECT current_user();

+----------------------------------+
| current_user()                   |
+----------------------------------+
| skip-grants user@skip-grants host |
+----------------------------------+
1 row in set (0.00 sec)

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'P@s
ERROR 1290 (HY000): The MySQL server is running with th
option so it cannot execute this statement

mysql> FLUSH PRIVILEGES;

Query OK, 0 rows affected (0.02 sec)

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'P@s

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

Once the password is reset, it's recommended to restart the MySQL instance in a normal mode.

There's also another way to recover `root` 's password, which is arguably more secure. One of the numerous command-line arguments that `mysqld` takes is `--init-file` (or `init_file` if used through *my.cnf*). This argument specifies a path to a file containing some SQL statements that will be executed during MySQL startup. No privilege checks are done at that time, so it's possible to put an `ALTER USER root` statement there. It's recommended to delete the file and unset the option once you've regained access or created a new `root` user.

WARNING

Both of the options presented here can potentially lead to security issues. Please use them carefully!

Some Ideas for Secure Setup

During the course of this chapter, we outlined a few practices related to user and privilege management that can help make your server more secure and safe. Here we will provide a short summary of those techniques and our recommendations for using them.

From the administrative side, we have the following recommendations:

- Avoid overusing the built-in superuser `root@localhost` . Imagine five people having access to this user. Even if you have auditing enabled in MySQL, you won't be able to effectively discern which particular person accessed the user and when. This user will also be the first one that potential attackers will try to exploit.
- Starting with MySQL 8.0, avoid creating new superusers through the `SUPER` privilege. Instead, you can create a special DBA role that has either all dynamic privileges assigned individually or just some of them that are frequently required.
- Consider organizing privileges for DBA functions into separate roles. For example, the `INNODB_REDO_LOG_ARCHIVE` and `INNODB_REDO_LOG_ENABLE` privileges could be a part of the `innodb_redo_admin` role. Since roles are not by default automatically

activated, one would first need to `SET ROLE` explicitly before running potentially dangerous administrative commands.

For regular users, the recommendations are pretty similar:

- Try to minimize the scope of permissions. Always ask if this user needs access to every database in the cluster, or even every table in a particular database.
- With MySQL 8.0, using roles is a convenient and arguably safer way to group and manage privileges. If you have three users that need the same or similar privileges, they could share a single role.
- Never allow any non-superuser modification permissions on tables in the `mysql` database. This is a simple mistake that follows from the first recommendation in this list. Granting `UPDATE` on `*.*` will allow the grantee to grant itself any permissions.

- To make things even more secure and visible, you can consider periodically saving all of the privileges currently assigned to users and comparing the result with the previously saved sample. You can easily diff the `pt-show-grants` output, or even the `mysqldump` output.

With this chapter done, you should be comfortable administering users and privileges in MySQL.