Security in MySQL

Security in MySQL

Abstract

This is the MySQL Security Guide extract from the MySQL 6.0 Reference Manual.

Document generated on: 2009-06-02 (revision: 15165)

Preface

When thinking about security within MySQL you should consider a wide range of possible topics and how they might affect the security of your MySQL server and related applications.

All of the following are issues that you should be aware of:

- Security of the installation itself. The data files, log files, and the all the application files of your installation should be protected to ensure that they are not readable or writable by unauthorized parties. For more information, see Chapter 2, *Post-Installation Setup and Testing*.
- Access control and security within the database system itself, including the users and databases granted with access to the databases, views and stored programs in use within the database. For more information, see Chapter 3, *The MySQL Access Privilege System*, Chapter 4, *MySQL User Account Management*.
- Network security of MySQL and your system. The security is related to the grants for individual users, but you may also wish to restrict MySQL so that is only available locally, or to a limited set of hosts.
- Security of your application to ensure that SQL injections and other corruption of the data does not occur. See Chapter 1, *General Security Issues*.
- Ensure that you have adequate and appropriate backups of your database files, configuration and log files. Also be sure that you have a recovery solution in place and test that you are able to successfully recover the information from your backups. See Chapter 5, *Backup and Recovery*.

Chapter 1. General Security Issues

This section describes some general security issues to be aware of and what you can do to make your MySQL installation more secure against attack or misuse. For information specifically about the access control system that MySQL uses for setting up user accounts and checking database access, see Chapter 3, *The MySQL Access Privilege System*.

For answers to some questions that are often asked about MySQL Server security issues, see MySQL 5.1 FAQ — Security.

1.1. General Security Guidelines

Anyone using MySQL on a computer connected to the Internet should read this section to avoid the most common security mistakes.

In discussing security, we emphasize the necessity of fully protecting the entire server host (not just the MySQL server) against all types of applicable attacks: eavesdropping, altering, playback, and denial of service. We do not cover all aspects of availability and fault tolerance here.

MySQL uses security based on Access Control Lists (ACLs) for all connections, queries, and other operations that users can attempt to perform. There is also support for SSL-encrypted connections between MySQL clients and servers. Many of the concepts discussed here are not specific to MySQL at all; the same general ideas apply to almost all applications.

When running MySQL, follow these guidelines whenever possible:

- **Do not ever give anyone (except MySQL `root` accounts) access to the `user` table in the `mysql` database!** This is critical.
- Learn the MySQL access privilege system. The `GRANT` and `REVOKE` statements are used for controlling access to MySQL. Do not grant more privileges than necessary. Never grant privileges to all hosts.

  Checklist:

  - Try `mysql -u root`. If you are able to connect successfully to the server without being asked for a password, anyone can connect to your MySQL server as the MySQL `root` user with full privileges! Review the MySQL installation instructions, paying particular attention to the information about setting a `root` password. See Section 2.3, “Securing the Initial MySQL Accounts”.
  - Use the `SHOW GRANTS` statement to check which accounts have access to what. Then use the `REVOKE` statement to remove those privileges that are not necessary.
- Do not store any plain-text passwords in your database. If your computer becomes compromised, the intruder can take the full list of passwords and use them. Instead, use `MD5()`, `SHA1()`, or some other one-way hashing function and store the hash value.
- Do not choose passwords from dictionaries. Special programs exist to break passwords. Even passwords like “xfish98” are very bad. Much better is “duag98” which contains the same word “fish” but typed one key to the left on a standard QWERTY keyboard. Another method is to use a password that is taken from the first characters of each word in a sentence (for example, “Mary had a little lamb” results in a password of “Mhall”). The password is easy to remember and type, but difficult to guess for someone who does not know the sentence.
- Invest in a firewall. This protects you from at least 50% of all types of exploits in any software. Put MySQL behind the firewall or in a demilitarized zone (DMZ).

  Checklist:

  - Try to scan your ports from the Internet using a tool such as `nmap`. MySQL uses port 3306 by default. This port should not be accessible from untrusted hosts. Another simple way to check whether or not your MySQL port is open is to try the following command from some remote machine, where *`server_host`* is the host name or IP number of the host on which your MySQL server runs:

    ```
    shell> telnet server_host 3306
    ```

    If you get a connection and some garbage characters, the port is open, and should be closed on your firewall or router, unless you really have a good reason to keep it open. If `telnet` hangs or the connection is refused, the port is blocked, which is how you want it to be.
- Do not trust any data entered by users of your applications. They can try to trick your code by entering special or escaped character sequences in Web forms, URLs, or whatever application you have built. Be sure that your application remains secure if a user enters something like “`; DROP DATABASE mysql;`”. This is an extreme example, but large security leaks and data loss might occur as a result of hackers using similar techniques, if you do not prepare for them.

A common mistake is to protect only string data values. Remember to check numeric data as well. If an application generates a query such as `SELECT * FROM table WHERE ID=234` when a user enters the value `234`, the user can enter the value `234 OR 1=1` to cause the application to generate the query `SELECT * FROM table WHERE ID=234 OR 1=1`. As a result, the server retrieves every row in the table. This exposes every row and causes excessive server load. The simplest way to protect from this type of attack is to use single quotes around the numeric constants: `SELECT * FROM table WHERE ID='234'`. If the user enters extra information, it all becomes part of the string. In a numeric context, MySQL automatically converts this string to a number and strips any trailing non-numeric characters from it.

Sometimes people think that if a database contains only publicly available data, it need not be protected. This is incorrect. Even if it is allowable to display any row in the database, you should still protect against denial of service attacks (for example, those that are based on the technique in the preceding paragraph that causes the server to waste resources). Otherwise, your server becomes unresponsive to legitimate users.

Checklist:

- Try to enter single and double quote marks ("`'`" and "`"`") in all of your Web forms. If you get any kind of MySQL error, investigate the problem right away.
- Try to modify dynamic URLs by adding `%22` ("`"`"), `%23` ("#"), and `%27` ("`'`") to them.
- Try to modify data types in dynamic URLs from numeric to character types using the characters shown in the previous examples. Your application should be safe against these and similar attacks.
- Try to enter characters, spaces, and special symbols rather than numbers in numeric fields. Your application should remove them before passing them to MySQL or else generate an error. Passing unchecked values to MySQL is very dangerous!
- Check the size of data before passing it to MySQL.
- Have your application connect to the database using a user name different from the one you use for administrative purposes. Do not give your applications any access privileges they do not need.

- Many application programming interfaces provide a means of escaping special characters in data values. Properly used, this prevents application users from entering values that cause the application to generate statements that have a different effect than you intend:
  - MySQL C API: Use the `mysql_real_escape_string()` API call.
  - MySQL++: Use the `escape` and `quote` modifiers for query streams.
  - PHP: Use the `mysql_real_escape_string()` function (available as of PHP 4.3.0, prior to that PHP version use `mysql_escape_string()`, and prior to PHP 4.0.3, use `addslashes()`). Note that only `mysql_real_escape_string()` is character set-aware; the other functions can be "bypassed" when using (invalid) multi-byte character sets. In PHP 5, you can use the `mysqli` extension, which supports the improved MySQL authentication protocol and passwords, as well as prepared statements with placeholders.
  - Perl DBI: Use placeholders or the `quote()` method.
  - Ruby DBI: Use placeholders or the `quote()` method.
  - Java JDBC: Use a `PreparedStatement` object and placeholders.

  Other programming interfaces might have similar capabilities.
- Do not transmit plain (unencrypted) data over the Internet. This information is accessible to everyone who has the time and ability to intercept it and use it for their own purposes. Instead, use an encrypted protocol such as SSL or SSH. MySQL supports internal SSL connections as of version 4.0. Another technique is to use SSH port-forwarding to create an encrypted (and compressed) tunnel for the communication.
- Learn to use the `tcpdump` and `strings` utilities. In most cases, you can check whether MySQL data streams are unencrypted by issuing a command like the following:

```
shell> tcpdump -l -i eth0 -w - src or dst port 3306 | strings
```

This works under Linux and should work with small modifications under other systems.

> **Warning**
>
> If you do not see plaintext data, this does not always mean that the information actually is encrypted. If you need high security, you should consult with a security expert.

1.2. Making MySQL Secure Against Attackers

When you connect to a MySQL server, you should use a password. The password is not transmitted in clear text over the connection. Password handling during the client connection sequence was upgraded in MySQL 4.1.1 to be very secure. If you are still using pre-4.1.1-style passwords, the encryption algorithm is not as strong as the newer algorithm. With some effort, a clever attacker who can sniff the traffic between the client and the server can crack the password. (See Section 4.6.3, “Password Hashing in MySQL”, for a discussion of the different password handling methods.)

> **MySQL Enterprise**
> The MySQL Enterprise Monitor enforces best practices for maximizing the security of your servers. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

All other information is transferred as text, and can be read by anyone who is able to watch the connection. If the connection between the client and the server goes through an untrusted network, and you are concerned about this, you can use the compressed protocol to make traffic much more difficult to decipher. You can also use MySQL's internal SSL support to make the connection even more secure. See Section 4.7, “Using SSL for Secure Connections”. Alternatively, use SSH to get an encrypted TCP/IP connection between a MySQL server and a MySQL client. You can find an Open Source SSH client at http://www.openssh.org/, and a commercial SSH client at http://www.ssh.com/.

To make a MySQL system secure, you should strongly consider the following suggestions:

- Require all MySQL accounts to have a password. A client program does not necessarily know the identity of the person running it. It is common for client/server applications that the user can specify any user name to the client program. For example, anyone can use the `mysql` program to connect as any other person simply by invoking it as `mysql -u other_user db_name` if `other_user` has no password. If all accounts have a password, connecting using another user's account becomes much more difficult.

  For a discussion of methods for setting passwords, see Section 4.5, “Assigning Account Passwords”.

- Never run the MySQL server as the Unix `root` user. This is extremely dangerous, because any user with the `FILE` privilege is able to cause the server to create files as `root` (for example, `~root/.bashrc`). To prevent this, `mysqld` refuses to run as `root` unless that is specified explicitly using the `--user=root` option.

  `mysqld` can (and should) be run as an ordinary, unprivileged user instead. You can create a separate Unix account named `mysql` to make everything even more secure. Use this account only for administering MySQL. To start `mysqld` as a different Unix user, add a `user` option that specifies the user name in the `[mysqld]` group of the `my.cnf` option file where you specify server options. For example:

  ```
  [mysqld]
  user=mysql
  ```

  This causes the server to start as the designated user whether you start it manually or by using `mysqld_safe` or `mysql.server`. For more details, see Section 1.5, “How to Run MySQL as a Normal User”.

  Running `mysqld` as a Unix user other than `root` does not mean that you need to change the `root` user name in the `user` table. *User names for MySQL accounts have nothing to do with user names for Unix accounts.*

- Do not allow the use of symlinks to tables. (This capability can be disabled with the `--skip-symbolic-links` option.) This is especially important if you run `mysqld` as `root`, because anyone that has write access to the server's data directory then could delete any file in the system! See Using Symbolic Links for Tables on Unix.

- Make sure that the only Unix user account with read or write privileges in the database directories is the account that is used for running `mysqld`.

- Do not grant the `PROCESS` or `SUPER` privilege to non-administrative users. The output of `mysqladmin processlist` and `SHOW PROCESSLIST` shows the text of any statements currently being executed, so any user who is allowed to see the server process list might be able to see statements issued by other users such as `UPDATE user SET password=PASSWORD('not_secure')`.

  `mysqld` reserves an extra connection for users who have the `SUPER` privilege, so that a MySQL `root` user can log in and check server activity even if all normal connections are in use.

  The `SUPER` privilege can be used to terminate client connections, change server operation by changing the value of system variables, and control replication servers.

- Do not grant the `FILE` privilege to non-administrative users. Any user that has this privilege can write a file anywhere in the file system with the privileges of the `mysqld` daemon. To make this a bit safer, files generated with `SELECT ... INTO OUTFILE` do not overwrite existing files and are writable by everyone.

The FILE privilege may also be used to read any file that is world-readable or accessible to the Unix user that the server runs as. With this privilege, you can read any file into a database table. This could be abused, for example, by using LOAD DATA to load /etc/passwd into a table, which then can be displayed with SELECT.

- If you do not trust your DNS, you should use IP numbers rather than host names in the grant tables. In any case, you should be very careful about creating grant table entries using host name values that contain wildcards.
- If you want to restrict the number of connections allowed to a single account, you can do so by setting the max_user_connections variable in mysqld. The GRANT statement also supports resource control options for limiting the extent of server use allowed to an account. See GRANT Syntax.

1.3. Security-Related mysqld Options

The following mysqld options affect security:

Table 1.1. mysqld Security Option/Variable Summary

Name	Cmd-Line	Option file	System Var	Status Var	Var Scope	Dynamic
allow-suspicious-udfs	Yes	Yes				
automatic_sp_privileges			Yes		Global	Yes
chroot	Yes	Yes				
des-key-file	Yes	Yes				
local_infile			Yes		Global	Yes
local-infile	Yes	Yes				
- *Variable*: local_infile						
old-passwords	Yes	Yes			Both	Yes
- *Variable*: old_passwords			Yes		Both	Yes
safe-show-database	Yes	Yes	Yes		Global	Yes
safe-user-create	Yes	Yes				
secure-auth	Yes	Yes			Global	Yes
- *Variable*: secure_auth			Yes		Global	Yes
secure-backup-file-priv	Yes	Yes			Global	No
- *Variable*: secure_backup_file_priv			Yes		Global	No
secure-file-priv	Yes	Yes			Global	No
- *Variable*: secure_file_priv			Yes		Global	No
skip-grant-tables	Yes	Yes				
skip-name-resolve	Yes	Yes				
skip-networking	Yes	Yes			Global	No
- *Variable*: skip_networking			Yes		Global	No
skip-show-database	Yes	Yes			Global	No
- *Variable*: skip_show_database			Yes		Global	No

- --allow-suspicious-udfs

  This option controls whether user-defined functions that have only an xxx symbol for the main function can be loaded. By default, the option is off and only UDFs that have at least one auxiliary symbol can be loaded; this prevents attempts at loading functions from shared object files other than those containing legitimate UDFs. See User-Defined Function Security Precautions.

- --local-infile[={0|1}]

  If you start the server with --local-infile=0, clients cannot use LOCAL in LOAD DATA statements. See Section 1.4, “Security Issues with LOAD DATA LOCAL”.

- `--old-passwords`

  Force the server to generate short (pre-4.1) password hashes for new passwords. This is useful for compatibility when the server must support older client programs. See Section 4.6.3, “Password Hashing in MySQL”.

  MySQL Enterprise
  The MySQL Enterprise Monitor offers advice on the security implications of using this option. For subscription information, see http://www.mysql.com/products/enterprise/advisors.html.

- `--safe-show-database` (*OBSOLETE*)

  In previous versions of MySQL, this option caused the `SHOW DATABASES` statement to display the names of only those databases for which the user had some kind of privilege. In MySQL 6.0, this option is no longer available as this is now the default behavior, and there is a `SHOW DATABASES` privilege that can be used to control access to database names on a per-account basis. See `GRANT` Syntax.

- `--safe-user-create`

  If this option is enabled, a user cannot create new MySQL users by using the `GRANT` statement unless the user has the `INSERT` privilege for the `mysql.user` table or any column in the table. If you want a user to have the ability to create new users that have those privileges that the user has the right to grant, you should grant the user the following privilege:

  ```
  GRANT INSERT(user) ON mysql.user TO 'user_name'@'host_name';
  ```

  This ensures that the user cannot change any privilege columns directly, but has to use the `GRANT` statement to give privileges to other users.

- `--secure-auth`

  Disallow authentication for accounts that have old (pre-4.1) passwords.

  The `mysql` client also has a `--secure-auth` option, which prevents connections to a server if the server requires a password in old format for the client account.

- `--secure-backup-file-priv=path`

Command Line Format	`--secure-backup-file-priv`	
Config File Format	`secure-backup-file-priv`	
Option Sets Variable	Yes, `secure_backup_file_priv`	
Variable Name	`secure_backup_file_priv`	
Variable Scope	Global	
Dynamic Variable	No	
Value Set	**Type**	`string`

  This option limits the effect of the `BACKUP DATABASE` and `RESTORE` statements to work only with files in the specified directory. This option was added in MySQL 6.0.11; in older releases, use `--secure-file-priv` instead.

- `--secure-file-priv=path`

Command Line Format	`--secure-file-priv`	
Config File Format	`secure-file-priv`	
Option Sets Variable	Yes, `secure_file_priv`	
Variable Name	`secure_file_priv`	
Variable Scope	Global	
Dynamic Variable	No	
Value Set	**Type**	`string`

  This option limits the effect of the `LOAD_FILE()` function and the `LOAD DATA` and `SELECT ... INTO OUTFILE` statements to work only with files in the specified directory. Before MySQL 6.0.11, it also applies to the `BACKUP DATABASE` and `RESTORE` statements; as of 6.0.11, `--secure-backup-file-priv` applies to those statements.

- `--skip-grant-tables`

  This option causes the server not to use the privilege system at all. This gives anyone with access to the server *unrestricted access* to *all databases*. You can cause a running server to start using the grant tables again by executing `mysqladmin flush-privileges` or `mysqladmin reload` command from a system shell, or by issuing a MySQL `FLUSH PRIVILEGES` statement. This option also suppresses loading of plugins and user-defined functions (UDFs).

- `--skip-name-resolve`

  Host names are not resolved. All `Host` column values in the grant tables must be IP numbers or `localhost`.

- `--skip-networking`

  Do not allow TCP/IP connections over the network. All connections to `mysqld` must be made via Unix socket files.

- `--skip-show-database`

  With this option, the `SHOW DATABASES` statement is allowed only to users who have the `SHOW DATABASES` privilege, and the statement displays all database names. Without this option, `SHOW DATABASES` is allowed to all users, but displays each database name only if the user has the `SHOW DATABASES` privilege or some privilege for the database. Note that any global privilege is a privilege for the database.

- `--ssl*`

  Options that begin with `--ssl` specify whether to allow clients to connect via SSL and indicate where to find SSL keys and certificates. See Section 4.7.3, “SSL Command Options”.

1.4. Security Issues with `LOAD DATA LOCAL`

The `LOAD DATA` statement can load a file that is located on the server host, or it can load a file that is located on the client host when the `LOCAL` keyword is specified.

There are two potential security issues with supporting the `LOCAL` version of `LOAD DATA` statements:

- The transfer of the file from the client host to the server host is initiated by the MySQL server. In theory, a patched server could be built that would tell the client program to transfer a file of the server's choosing rather than the file named by the client in the `LOAD DATA` statement. Such a server could access any file on the client host to which the client user has read access.
- In a Web environment where the clients are connecting from a Web server, a user could use `LOAD DATA LOCAL` to read any files that the Web server process has read access to (assuming that a user could run any command against the SQL server). In this environment, the client with respect to the MySQL server actually is the Web server, not the remote program being run by the user who connects to the Web server.

To deal with these problems, we changed how `LOAD DATA LOCAL` is handled as of MySQL 3.23.49 and MySQL 4.0.2 (4.0.13 on Windows):

- By default, all MySQL clients and libraries in binary distributions are compiled with the `--enable-local-infile` option, to be compatible with MySQL 3.23.48 and before.
- If you build MySQL from source but do not invoke `configure` with the `--enable-local-infile` option, `LOAD DATA LOCAL` cannot be used by any client unless it is written explicitly to invoke `mysql_options(... MYSQL_OPT_LOCAL_INFILE, 0)`. See `mysql_options()`.
- You can disable all `LOAD DATA LOCAL` commands from the server side by starting `mysqld` with the `--local-infile=0` option.
- For the `mysql` command-line client, enable `LOAD DATA LOCAL` by specifying the `--local-infile[=1]` option, or disable it with the `--local-infile=0` option. For `mysqlimport`, local data file loading is off by default; enable it with the `--local` or `-L` option. In any case, successful use of a local load operation requires that the server is enabled to allow it.
- If you use `LOAD DATA LOCAL` in Perl scripts or other programs that read the `[client]` group from option files, you can add the `local-infile=1` option to that group. However, to keep this from causing problems for programs that do not understand `local-infile`, specify it using the `loose-` prefix:

```
[client]
loose-local-infile=1
```

- If `LOAD DATA LOCAL` is disabled, either in the server or the client, a client that attempts to issue such a statement receives the following error message:

```
ERROR 1148: The used command is not allowed with this MySQL version
```

> **MySQL Enterprise**
> Security advisors notify subscribers to the MySQL Enterprise Monitor whenever a server is started with the `--local-infile` option enabled. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

1.5. How to Run MySQL as a Normal User

On Windows, you can run the server as a Windows service using a normal user account.

On Unix, the MySQL server `mysqld` can be started and run by any user. However, you should avoid running the server as the Unix `root` user for security reasons. To change `mysqld` to run as a normal unprivileged Unix user *`user_name`*, you must do the following:

1. Stop the server if it is running (use `mysqladmin shutdown`).
2. Change the database directories and files so that *`user_name`* has privileges to read and write files in them (you might need to do this as the Unix `root` user):

   ```
   shell> chown -R user_name /path/to/mysql/datadir
   ```

   If you do not do this, the server will not be able to access databases or tables when it runs as *`user_name`*.

   If directories or files within the MySQL data directory are symbolic links, `chown -R` might not follow symbolic links for you. If it does not, you will also need to follow those links and change the directories and files they point to.
3. Start the server as user *`user_name`*. Another alternative is to start `mysqld` as the Unix `root` user and use the `--user=`*`user_name`* option. `mysqld` starts up, then switches to run as the Unix user *`user_name`* before accepting any connections.
4. To start the server as the given user automatically at system startup time, specify the user name by adding a `user` option to the `[mysqld]` group of the `/etc/my.cnf` option file or the `my.cnf` option file in the server's data directory. For example:

   ```
   [mysqld]
   user=user_name
   ```

If your Unix machine itself isn't secured, you should assign passwords to the MySQL `root` accounts in the grant tables. Otherwise, any user with a login account on that machine can run the `mysql` client with a `--user=root` option and perform any operation. (It is a good idea to assign passwords to MySQL accounts in any case, but especially so when other login accounts exist on the server host.) See Chapter 2, *Post-Installation Setup and Testing*.

Chapter 2. Post-Installation Setup and Testing

After installing MySQL, there are some issues that you should address. For example, on Unix, you should initialize the data directory and create the MySQL grant tables. On all platforms, an important security concern is that the initial accounts in the grant tables have no passwords. You should assign passwords to prevent unauthorized access to the MySQL server. Optionally, you can create time zone tables to enable recognition of named time zones.

The following sections include post-installation procedures that are specific to Windows systems and to Unix systems. Another section, Section 2.2.3, "Starting and Troubleshooting the MySQL Server", applies to all platforms; it describes what to do if you have trouble getting the server to start. Section 2.3, "Securing the Initial MySQL Accounts", also applies to all platforms. You should follow its instructions to make sure that you have properly protected your MySQL accounts by assigning passwords to them.

When you are ready to create additional user accounts, you can find information on the MySQL access control system and account management in Chapter 3, *The MySQL Access Privilege System*, and Chapter 4, *MySQL User Account Management*.

2.1. Windows Post-Installation Procedures

On Windows, the data directory and the grant tables do not have to be created. MySQL Windows distributions include the grant tables with a set of preinitialized accounts in the `mysql` database under the data directory. It is unnecessary to run the `mysql_install_db` script that is used on Unix. Regarding passwords, if you installed MySQL using the Windows Installation Wizard, you may have already assigned passwords to the accounts. (See Using the MySQL Installation Wizard.) Otherwise, use the password-assignment procedure given in Section 2.3, "Securing the Initial MySQL Accounts".

Before setting up passwords, you might want to try running some client programs to make sure that you can connect to the server and that it is operating properly. Make sure that the server is running (see Starting the Server for the First Time), and then issue the following commands to verify that you can retrieve information from the server. The output should be similar to what is shown here:

```
C:\> C:\mysql\bin\mysqlshow
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
C:\> C:\mysql\bin\mysqlshow mysql
Database: mysql
+---------------------------+
|          Tables           |
+---------------------------+
| columns_priv              |
| db                        |
| event                     |
| func                      |
| general_log               |
| help_category             |
| help_keyword              |
| help_relation             |
| help_topic                |
| host                      |
| plugin                    |
| proc                      |
| procs_priv                |
| servers                   |
| slow_log                  |
| tables_priv               |
| time_zone                 |
| time_zone_leap_second     |
| time_zone_name            |
| time_zone_transition      |
| time_zone_transition_type |
| user                      |
+---------------------------+
C:\> C:\mysql\bin\mysql -e "SELECT Host,Db,User FROM db" mysql
+------+-------+------+
| host | db    | user |
+------+-------+------+
| %    | test% |      |
+------+-------+------+
```

You may need to specify a different directory from the one shown; if you used the Windows Installation Wizard, then the default directory is `C:\Program Files\MySQL\MySQL Server 6.0`, and the `mysql` and `mysqlshow` client programs are in `C:\Program Files\MySQL\MySQL Server 6.0\bin`. See Using the MySQL Installation Wizard, for more information.

If you have already secured the initial MySQL accounts, you may need to use the `-u` and `-p` options to supply a user name and password to the `mysqlshow` and `mysql` client programs; otherwise the programs may fail with an error, or you may not be able to view all databases. For example, if you have assigned the password "secretpass" to the MySQL `root` account, then you can in-

voke `mysqlshow` and `mysql` as shown here:

```
C:\> C:\mysql\bin\mysqlshow -uroot -psecretpass
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
C:\> C:\mysql\bin\mysqlshow -uroot -psecretpass mysql
Database: mysql
+---------------------------+
|          Tables           |
+---------------------------+
| columns_priv              |
| db                        |
| event                     |
| func                      |
| general_log               |
| help_category             |
| help_keyword              |
| help_relation             |
| help_topic                |
| host                      |
| plugin                    |
| proc                      |
| procs_priv                |
| servers                   |
| slow_log                  |
| tables_priv               |
| time_zone                 |
| time_zone_leap_second     |
| time_zone_name            |
| time_zone_transition      |
| time_zone_transition_type |
| user                      |
+---------------------------+
C:\> C:\mysql\bin\mysql -uroot -psecretpass -e "SELECT Host,Db,User FROM db" mysql
+------+-------+------+
| host | db    | user |
+------+-------+------+
| %    | test% |      |
+------+-------+------+
```

For more information about these programs, see `mysqlshow`, and `mysql`.

If you are running a version of Windows that supports services and you want the MySQL server to run automatically when Windows starts, see Starting MySQL as a Windows Service.

2.2. Unix Post-Installation Procedures

After installing MySQL on Unix, you need to initialize the grant tables, start the server, and make sure that the server works satisfactorily. You may also wish to arrange for the server to be started and stopped automatically when your system starts and stops. You should also assign passwords to the accounts in the grant tables.

On Unix, the grant tables are set up by the `mysql_install_db` program. For some installation methods, this program is run for you automatically:

- If you install MySQL on Linux using RPM distributions, the server RPM runs `mysql_install_db`.
- If you install MySQL on Mac OS X using a PKG distribution, the installer runs `mysql_install_db`.

Otherwise, you'll need to run `mysql_install_db` yourself.

The following procedure describes how to initialize the grant tables (if that has not previously been done) and then start the server. It also suggests some commands that you can use to test whether the server is accessible and working properly. For information about starting and stopping the server automatically, see Section 2.2.2, “Starting and Stopping MySQL Automatically”.

After you complete the procedure and have the server running, you should assign passwords to the accounts created by `mysql_install_db`. Instructions for doing so are given in Section 2.3, “Securing the Initial MySQL Accounts”.

In the examples shown here, the server runs under the user ID of the `mysql` login account. This assumes that such an account exists. Either create the account if it does not exist, or substitute the name of a different existing login account that you plan to use for running the server.

1. Change location into the top-level directory of your MySQL installation, represented here by *`BASEDIR`*:

   ```
   shell> cd BASEDIR
   ```

BASEDIR is likely to be something like /usr/local/mysql or /usr/local. The following steps assume that you are located in this directory.

2. If necessary, run the mysql_install_db program to set up the initial MySQL grant tables containing the privileges that determine how users are allowed to connect to the server. You'll need to do this if you used a distribution type for which the installation procedure doesn't run the program for you.

   Typically, mysql_install_db needs to be run only the first time you install MySQL, so you can skip this step if you are upgrading an existing installation, However, mysql_install_db does not overwrite any existing privilege tables, so it should be safe to run in any circumstances.

   To initialize the grant tables, use one of the following commands, depending on whether mysql_install_db is located in the bin or scripts directory:

   ```
   shell> bin/mysql_install_db --user=mysql
   shell> scripts/mysql_install_db --user=mysql
   ```

   It might be necessary to specify other options such as --basedir or --datadir if mysql_install_db does not use the correct locations for the installation directory or data directory. For example:

   ```
   shell> bin/mysql_install_db --user=mysql \
            --basedir=/opt/mysql/mysql \
            --datadir=/opt/mysql/mysql/data
   ```

   The mysql_install_db script creates the server's data directory. Under the data directory, it creates directories for the mysql database that holds all database privileges and the test database that you can use to test MySQL. The script also creates privilege table entries for root and anonymous-user accounts. The accounts have no passwords initially. A description of their initial privileges is given in Section 2.3, "Securing the Initial MySQL Accounts". Briefly, these privileges allow the MySQL root user to do anything, and allow anybody to create or use databases with a name of test or starting with test_.

   It is important to make sure that the database directories and files are owned by the mysql login account so that the server has read and write access to them when you run it later. To ensure this, the --user option should be used as shown if you run mysql_install_db as root. Otherwise, you should execute the script while logged in as mysql, in which case you can omit the --user option from the command.

   mysql_install_db creates several tables in the mysql database, including user, db, host, tables_priv, columns_priv, func, and others. See Chapter 3, *The MySQL Access Privilege System*, for a complete listing and description of these tables.

   If you don't want to have the test database, you can remove it with mysqladmin -u root drop test after starting the server.

   If you have trouble with mysql_install_db at this point, see Section 2.2.1, "Problems Running mysql_install_db".

3. Start the MySQL server:

   ```
   shell> bin/mysqld_safe --user=mysql &
   ```

   It is important that the MySQL server be run using an unprivileged (non-root) login account. To ensure this, the --user option should be used as shown if you run mysqld_safe as system root. Otherwise, you should execute the script while logged in to the system as mysql, in which case you can omit the --user option from the command.

   Further instructions for running MySQL as an unprivileged user are given in Section 1.5, "How to Run MySQL as a Normal User".

   If you neglected to create the grant tables before proceeding to this step, the following message appears in the error log file when you start the server:

   ```
   mysqld: Can't find file: 'host.frm'
   ```

   If you have other problems starting the server, see Section 2.2.3, "Starting and Troubleshooting the MySQL Server".

4. Use mysqladmin to verify that the server is running. The following commands provide simple tests to check whether the server is up and responding to connections:

   ```
   shell> bin/mysqladmin version
   shell> bin/mysqladmin variables
   ```

The output from `mysqladmin version` varies slightly depending on your platform and version of MySQL, but should be similar to that shown here:

```
shell> bin/mysqladmin version
mysqladmin  Ver 14.12 Distrib 6.0.12, for pc-linux-gnu on i686
...
Server version          6.0.12
Protocol version        10
Connection              Localhost via UNIX socket
UNIX socket             /var/lib/mysql/mysql.sock
Uptime:                 14 days 5 hours 5 min 21 sec
Threads: 1  Questions: 366  Slow queries: 0
Opens: 0  Flush tables: 1  Open tables: 19
Queries per second avg: 0.000
```

To see what else you can do with `mysqladmin`, invoke it with the `--help` option.

5. Verify that you can shut down the server:

```
shell> bin/mysqladmin -u root shutdown
```

6. Verify that you can start the server again. Do this by using `mysqld_safe` or by invoking `mysqld` directly. For example:

```
shell> bin/mysqld_safe --user=mysql --log &
```

If `mysqld_safe` fails, see Section 2.2.3, “Starting and Troubleshooting the MySQL Server”.

7. Run some simple tests to verify that you can retrieve information from the server. The output should be similar to what is shown here:

```
shell> bin/mysqlshow
+-----------+
| Databases |
+-----------+
| mysql     |
| test      |
+-----------+
shell> bin/mysqlshow mysql
Database: mysql
+---------------------------+
|          Tables           |
+---------------------------+
| columns_priv              |
| db                        |
| func                      |
| help_category             |
| help_keyword              |
| help_relation             |
| help_topic                |
| host                      |
| proc                      |
| procs_priv                |
| tables_priv               |
| time_zone                 |
| time_zone_leap_second     |
| time_zone_name            |
| time_zone_transition      |
| time_zone_transition_type |
| user                      |
+---------------------------+
shell> bin/mysql -e "SELECT Host,Db,User FROM db" mysql
+------+--------+------+
| host | db     | user |
+------+--------+------+
| %    | test   |      |
| %    | test_% |      |
+------+--------+------+
```

8. There is a benchmark suite in the `sql-bench` directory (under the MySQL installation directory) that you can use to compare how MySQL performs on different platforms. The benchmark suite is written in Perl. It requires the Perl DBI module that provides a database-independent interface to the various databases, and some other additional Perl modules:

```
DBI
DBD::mysql
Data::Dumper
Data::ShowTable
```

These modules can be obtained from CPAN (http://www.cpan.org/). See also Installing Perl on Unix.

The `sql-bench/Results` directory contains the results from many runs against different databases and platforms. To run all tests, execute these commands:

```
shell> cd sql-bench
shell> perl run-all-tests
```

If you don't have the `sql-bench` directory, you probably installed MySQL using RPM files other than the source RPM. (The source RPM includes the `sql-bench` benchmark directory.) In this case, you must first install the benchmark suite before you can use it. There are separate benchmark RPM files named `mysql-bench-VERSION.i386.rpm` that contain benchmark code and data.

If you have a source distribution, there are also tests in its `tests` subdirectory that you can run. For example, to run `auto_increment.tst`, execute this command from the top-level directory of your source distribution:

```
shell> mysql -vvf test < ./tests/auto_increment.tst
```

The expected result of the test can be found in the `./tests/auto_increment.res` file.

9. At this point, you should have the server running. However, none of the initial MySQL accounts have a password, so you should assign passwords using the instructions found in Section 2.3, “Securing the Initial MySQL Accounts”.

The MySQL 6.0 installation procedure creates time zone tables in the `mysql` database. However, you must populate the tables manually using the instructions in MySQL Server Time Zone Support.

2.2.1. Problems Running `mysql_install_db`

The purpose of the `mysql_install_db` script is to generate new MySQL privilege tables. It does not overwrite existing MySQL privilege tables, and it does not affect any other data.

If you want to re-create your privilege tables, first stop the `mysqld` server if it is running. Then rename the `mysql` directory under the data directory to save it, and then run `mysql_install_db`. Suppose that your current directory is the MySQL installation directory and that `mysql_install_db` is located in the `bin` directory and the data directory is named `data`. To rename the `mysql` database and re-run `mysql_install_db`, use these commands.

```
shell> mv data/mysql data/mysql.old
shell> bin/mysql_install_db --user=mysql
```

When you run `mysql_install_db`, you might encounter the following problems:

- **`mysql_install_db` fails to install the grant tables**

  You may find that `mysql_install_db` fails to install the grant tables and terminates after displaying the following messages:

  ```
  Starting mysqld daemon with databases from XXXXXX
  mysqld ended
  ```

  In this case, you should examine the error log file very carefully. The log should be located in the directory `XXXXXX` named by the error message and should indicate why `mysqld` didn't start. If you do not understand what happened, include the log when you post a bug report. See How to Report Bugs or Problems.

- **There is a `mysqld` process running**

  This indicates that the server is running, in which case the grant tables have probably been created already. If so, there is no need to run `mysql_install_db` at all because it needs to be run only once (when you install MySQL the first time).

- **Installing a second `mysqld` server does not work when one server is running**

  This can happen when you have an existing MySQL installation, but want to put a new installation in a different location. For example, you might have a production installation, but you want to create a second installation for testing purposes. Generally the problem that occurs when you try to run a second server is that it tries to use a network interface that is in use by the first server. In this case, you should see one of the following error messages:

  ```
  Can't start server: Bind on TCP/IP port:
  Address already in use
  Can't start server: Bind on unix socket...
  ```

  For instructions on setting up multiple servers, see Running Multiple MySQL Servers on the Same Machine.

- **You do not have write access to the `/tmp` directory**

If you do not have write access to create temporary files or a Unix socket file in the default location (the `/tmp` directory), an error occurs when you run `mysql_install_db` or the `mysqld` server.

You can specify different locations for the temporary directory and Unix socket file by executing these commands prior to starting `mysql_install_db` or `mysqld`, where *`some_tmp_dir`* is the full path name to some directory for which you have write permission:

```
shell> TMPDIR=/some_tmp_dir/
shell> MYSQL_UNIX_PORT=/some_tmp_dir/mysql.sock
shell> export TMPDIR MYSQL_UNIX_PORT
```

Then you should be able to run `mysql_install_db` and start the server with these commands:

```
shell> bin/mysql_install_db --user=mysql
shell> bin/mysqld_safe --user=mysql &
```

If `mysql_install_db` is located in the `scripts` directory, modify the first command to `scripts/mysql_install_db`.

See How to Protect or Change the MySQL Unix Socket File, and Environment Variables.

There are some alternatives to running the `mysql_install_db` script provided in the MySQL distribution:

- If you want the initial privileges to be different from the standard defaults, you can modify `mysql_install_db` before you run it. However, it is preferable to use `GRANT` and `REVOKE` to change the privileges *after* the grant tables have been set up. In other words, you can run `mysql_install_db`, and then use `mysql -u root mysql` to connect to the server as the MySQL `root` user so that you can issue the necessary `GRANT` and `REVOKE` statements.

  If you want to install MySQL on several machines with the same privileges, you can put the `GRANT` and `REVOKE` statements in a file and execute the file as a script using `mysql` after running `mysql_install_db`. For example:

  ```
  shell> bin/mysql_install_db --user=mysql
  shell> bin/mysql -u root < your_script_file
  ```

  By doing this, you can avoid having to issue the statements manually on each machine.

- It is possible to re-create the grant tables completely after they have previously been created. You might want to do this if you're just learning how to use `GRANT` and `REVOKE` and have made so many modifications after running `mysql_install_db` that you want to wipe out the tables and start over.

  To re-create the grant tables, remove all the `.frm`, `.MYI`, and `.MYD` files in the `mysql` database directory. Then run the `mysql_install_db` script again.

- You can start `mysqld` manually using the `--skip-grant-tables` option and add the privilege information yourself using `mysql`:

  ```
  shell> bin/mysqld_safe --user=mysql --skip-grant-tables &
  shell> bin/mysql mysql
  ```

  From `mysql`, manually execute the SQL commands contained in `mysql_install_db`. Make sure that you run `mysqladmin flush-privileges` or `mysqladmin reload` afterward to tell the server to reload the grant tables.

  Note that by not using `mysql_install_db`, you not only have to populate the grant tables manually, you also have to create them first.

2.2.2. Starting and Stopping MySQL Automatically

Generally, you start the `mysqld` server in one of these ways:

- By invoking `mysqld` directly. This works on any platform.
- By running the MySQL server as a Windows service. The service can be set to start the server automatically when Windows starts, or as a manual service that you start on request. For instructions, see Starting MySQL as a Windows Service.
- By invoking `mysqld_safe`, which tries to determine the proper options for `mysqld` and then runs it with those options. This

script is used on Unix and Unix-like systems. See `mysqld_safe`.

- By invoking `mysql.server`. This script is used primarily at system startup and shutdown on systems that use System V-style run directories, where it usually is installed under the name `mysql`. The `mysql.server` script starts the server by invoking `mysqld_safe`. See `mysql.server`.
- On Mac OS X, you can install a separate MySQL Startup Item package to enable the automatic startup of MySQL on system startup. The Startup Item starts the server by invoking `mysql.server`. See Installing MySQL on Mac OS X, for details.

The `mysqld_safe` and `mysql.server` scripts and the Mac OS X Startup Item can be used to start the server manually, or automatically at system startup time. `mysql.server` and the Startup Item also can be used to stop the server.

To start or stop the server manually using the `mysql.server` script, invoke it with `start` or `stop` arguments:

```
shell> mysql.server start
shell> mysql.server stop
```

Before `mysql.server` starts the server, it changes location to the MySQL installation directory, and then invokes `mysqld_safe`. If you want the server to run as some specific user, add an appropriate `user` option to the `[mysqld]` group of the `/etc/my.cnf` option file, as shown later in this section. (It is possible that you will need to edit `mysql.server` if you've installed a binary distribution of MySQL in a non-standard location. Modify it to `cd` into the proper directory before it runs `mysqld_safe`. If you do this, your modified version of `mysql.server` may be overwritten if you upgrade MySQL in the future, so you should make a copy of your edited version that you can reinstall.)

`mysql.server stop` stops the server by sending a signal to it. You can also stop the server manually by executing `mysqladmin shutdown`.

To start and stop MySQL automatically on your server, you need to add start and stop commands to the appropriate places in your `/etc/rc*` files.

If you use the Linux server RPM package (`MySQL-server-`*`VERSION`*`.rpm`), the `mysql.server` script is installed in the `/etc/init.d` directory with the name `mysql`. You need not install it manually. See Installing MySQL from RPM Packages on Linux, for more information on the Linux RPM packages.

Some vendors provide RPM packages that install a startup script under a different name such as `mysqld`.

If you install MySQL from a source distribution or using a binary distribution format that does not install `mysql.server` automatically, you can install it manually. The script can be found in the `support-files` directory under the MySQL installation directory or in a MySQL source tree.

To install `mysql.server` manually, copy it to the `/etc/init.d` directory with the name `mysql`, and then make it executable. Do this by changing location into the appropriate directory where `mysql.server` is located and executing these commands:

```
shell> cp mysql.server /etc/init.d/mysql
shell> chmod +x /etc/init.d/mysql
```

Older Red Hat systems use the `/etc/rc.d/init.d` directory rather than `/etc/init.d`. Adjust the preceding commands accordingly. Alternatively, first create `/etc/init.d` as a symbolic link that points to `/etc/rc.d/init.d`:

```
shell> cd /etc
shell> ln -s rc.d/init.d .
```

After installing the script, the commands needed to activate it to run at system startup depend on your operating system. On Linux, you can use `chkconfig`:

```
shell> chkconfig --add mysql
```

On some Linux systems, the following command also seems to be necessary to fully enable the `mysql` script:

```
shell> chkconfig --level 345 mysql on
```

On FreeBSD, startup scripts generally should go in `/usr/local/etc/rc.d/`. The `rc(8)` manual page states that scripts in this directory are executed only if their basename matches the `*.sh` shell file name pattern. Any other files or directories present within the directory are silently ignored. In other words, on FreeBSD, you should install the `mysql.server` script as `/usr/local/etc/rc.d/mysql.server.sh` to enable automatic startup.

As an alternative to the preceding setup, some operating systems also use `/etc/rc.local` or `/etc/init.d/boot.local` to start additional services on startup. To start up MySQL using this method, you could append a command like the one following to the appropriate startup file:

```
/bin/sh -c 'cd /usr/local/mysql; ./bin/mysqld_safe --user=mysql &'
```

For other systems, consult your operating system documentation to see how to install startup scripts.

You can add options for `mysql.server` in a global `/etc/my.cnf` file. A typical `/etc/my.cnf` file might look like this:

```
[mysqld]
datadir=/usr/local/mysql/var
socket=/var/tmp/mysql.sock
port=3306
user=mysql
[mysql.server]
basedir=/usr/local/mysql
```

The `mysql.server` script supports the following options: `basedir`, `datadir`, and `pid-file`. If specified, they *must* be placed in an option file, not on the command line. `mysql.server` supports only `start` and `stop` as command-line arguments.

The following table shows which option groups the server and each startup script read from option files.

Script	Option Groups
`mysqld`	`[mysqld]`, `[server]`, `[mysqld-`*`major_version`*`]`
`mysqld_safe`	`[mysqld]`, `[server]`, `[mysqld_safe]`
`mysql.server`	`[mysqld]`, `[mysql.server]`, `[server]`

`[mysqld-`*`major_version`*`]` means that groups with names like `[mysqld-5.1]` and `[mysqld-6.0]` are read by servers having versions 5.1.x, 6.0.x, and so forth. This feature can be used to specify options that can be read only by servers within a given release series.

For backward compatibility, `mysql.server` also reads the `[mysql_server]` group and `mysqld_safe` also reads the `[safe_mysqld]` group. However, you should update your option files to use the `[mysql.server]` and `[mysqld_safe]` groups instead when using MySQL 6.0.

See Using Option Files.

2.2.3. Starting and Troubleshooting the MySQL Server

This section provides troubleshooting suggestions for problems starting the server on Unix. If you are using Windows, see Troubleshooting a MySQL Installation Under Windows.

If you have problems starting the server, here are some things to try:

- Check the error log to see why the server does not start.
- Specify any special options needed by the storage engines you are using.
- Make sure that the server knows where to find the data directory.
- Make sure that the server can access the data directory. The ownership and permissions of the data directory and its contents must be set such that the server can read and modify them.
- Verify that the network interfaces the server wants to use are available.

Some storage engines have options that control their behavior. You can create a `my.cnf` file and specify startup options for the engines that you plan to use. If you are going to use storage engines that support transactional tables (`InnoDB`, `NDB`), be sure that you have them configured the way you want before starting the server:

> **MySQL Enterprise**
> For expert advice on start-up options appropriate to your circumstances, subscribe to The MySQL Enterprise Monitor. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

- If you are using `InnoDB` tables, see `InnoDB` Configuration.

Storage engines will use default option values if you specify none, but it is recommended that you review the available options and specify explicit values for those for which the defaults are not appropriate for your installation.

When the `mysqld` server starts, it changes location to the data directory. This is where it expects to find databases and where it expects to write log files. The server also writes the pid (process ID) file in the data directory.

The data directory location is hardwired in when the server is compiled. This is where the server looks for the data directory by default. If the data directory is located somewhere else on your system, the server will not work properly. You can determine what the default path settings are by invoking `mysqld` with the `--verbose` and `--help` options.

If the default locations don't match the MySQL installation layout on your system, you can override them by specifying options to `mysqld` or `mysqld_safe` on the command line or in an option file.

To specify the location of the data directory explicitly, use the `--datadir` option. However, normally you can tell `mysqld` the location of the base directory under which MySQL is installed and it looks for the data directory there. You can do this with the `--basedir` option.

To check the effect of specifying path options, invoke `mysqld` with those options followed by the `--verbose` and `--help` options. For example, if you change location into the directory where `mysqld` is installed and then run the following command, it shows the effect of starting the server with a base directory of `/usr/local`:

```
shell> ./mysqld --basedir=/usr/local --verbose --help
```

You can specify other options such as `--datadir` as well, but `--verbose` and `--help` must be the last options.

Once you determine the path settings you want, start the server without `--verbose` and `--help`.

If `mysqld` is currently running, you can find out what path settings it is using by executing this command:

```
shell> mysqladmin variables
```

Or:

```
shell> mysqladmin -h host_name variables
```

`host_name` is the name of the MySQL server host.

If you get `Errcode 13` (which means `Permission denied`) when starting `mysqld`, this means that the privileges of the data directory or its contents do not allow the server access. In this case, you change the permissions for the involved files and directories so that the server has the right to use them. You can also start the server as `root`, but this raises security issues and should be avoided.

On Unix, change location into the data directory and check the ownership of the data directory and its contents to make sure the server has access. For example, if the data directory is `/usr/local/mysql/var`, use this command:

```
shell> ls -la /usr/local/mysql/var
```

If the data directory or its files or subdirectories are not owned by the login account that you use for running the server, change their ownership to that account. If the account is named `mysql`, use these commands:

```
shell> chown -R mysql /usr/local/mysql/var
shell> chgrp -R mysql /usr/local/mysql/var
```

If the server fails to start up correctly, check the error log. Log files are located in the data directory (typically `C:\Program Files\MySQL\MySQL Server 6.0\data` on Windows, `/usr/local/mysql/data` for a Unix binary distribution, and `/usr/local/var` for a Unix source distribution). Look in the data directory for files with names of the form *`host_name`*`.err` and *`host_name`*`.log`, where *`host_name`* is the name of your server host. Then examine the last few lines of these files. On Unix, you can use `tail` to display them:

```
shell> tail host_name.err
shell> tail host_name.log
```

The error log should contain information that indicates why the server couldn't start.

If either of the following errors occur, it means that some other program (perhaps another `mysqld` server) is using the TCP/IP port or Unix socket file that `mysqld` is trying to use:

```
Can't start server: Bind on TCP/IP port: Address already in use
Can't start server: Bind on unix socket...
```

Use `ps` to determine whether you have another `mysqld` server running. If so, shut down the server before starting `mysqld` again. (If another server is running, and you really want to run multiple servers, you can find information about how to do so in Running

Multiple MySQL Servers on the Same Machine.)

If no other server is running, try to execute the command `telnet your_host_name tcp_ip_port_number`. (The default MySQL port number is 3306.) Then press Enter a couple of times. If you don't get an error message like `telnet: Unable to connect to remote host: Connection refused`, some other program is using the TCP/IP port that `mysqld` is trying to use. You'll need to track down what program this is and disable it, or else tell `mysqld` to listen to a different port with the `--port` option. In this case, you'll also need to specify the port number for client programs when connecting to the server via TCP/IP.

Another reason the port might be inaccessible is that you have a firewall running that blocks connections to it. If so, modify the firewall settings to allow access to the port.

If the server starts but you can't connect to it, you should make sure that you have an entry in `/etc/hosts` that looks like this:

```
127.0.0.1       localhost
```

This problem occurs only on systems that do not have a working thread library and for which MySQL must be configured to use MIT-pthreads.

If you cannot get `mysqld` to start, you can try to make a trace file to find the problem by using the `--debug` option. See MySQL Internals: Porting.

2.3. Securing the Initial MySQL Accounts

Part of the MySQL installation process is to set up the `mysql` database that contains the grant tables:

- Windows distributions contain preinitialized grant tables that are installed automatically.
- On Unix, the grant tables are populated by the `mysql_install_db` program. Some installation methods run this program for you. Others require that you execute it manually. For details, see Section 2.2, “Unix Post-Installation Procedures”.

The grant tables define the initial MySQL user accounts and their access privileges. These accounts are set up as follows:

- Accounts with the user name `root` are created. These are superuser accounts that can do anything. The initial `root` account passwords are empty, so anyone can connect to the MySQL server as `root` — *without a password* — and be granted all privileges.
  - On Windows, one `root` account is created; this account allows connecting from the local host only. The Windows installer will optionally create an account allowing for connections from any host only if the user selects the **ENABLE ROOT ACCESS FROM REMOTE MACHINES** option during installation.
  - On Unix, both `root` accounts are for connections from the local host. Connections must be made from the local host by specifying a host name of `localhost` for one of the accounts, or the actual host name or IP number for the other.
- Two anonymous-user accounts are created, each with an empty user name. The anonymous accounts have no password, so anyone can use them to connect to the MySQL server.
  - On Windows, one anonymous account is for connections from the local host. It has no global privileges. The other is for connections from any host and has all privileges for the `test` database and for other databases with names that start with `test`.
  - On Unix, both anonymous accounts are for connections from the local host. Connections must be made from the local host by specifying a host name of `localhost` for one of the accounts, or the actual host name or IP number for the other. These accounts have all privileges for the `test` database and for other databases with names that start with `test_`.

As noted, none of the initial accounts have passwords. This means that your MySQL installation is unprotected until you do something about it:

- If you want to prevent clients from connecting as anonymous users without a password, you should either assign a password to each anonymous account or else remove the accounts.
- You should assign a password to each MySQL `root` account.

The following instructions describe how to set up passwords for the initial MySQL accounts, first for the anonymous accounts and then for the `root` accounts. Replace “*newpwd*” in the examples with the actual password that you want to use. The instructions

also cover how to remove the anonymous accounts, should you prefer not to allow anonymous access at all.

You might want to defer setting the passwords until later, so that you don't need to specify them while you perform additional setup or testing. However, be sure to set them before using your installation for production purposes.

Anonymous Account Password Assignment

To assign passwords to the anonymous accounts, connect to the server as `root` and then use either `SET PASSWORD` or `UPDATE`. In either case, be sure to encrypt the password using the `PASSWORD()` function.

To use `SET PASSWORD` on Windows, do this:

```
shell> mysql -u root
mysql> SET PASSWORD FOR ''@'localhost' = PASSWORD('newpwd');
mysql> SET PASSWORD FOR ''@'%' = PASSWORD('newpwd');
```

To use `SET PASSWORD` on Unix, do this:

```
shell> mysql -u root
mysql> SET PASSWORD FOR ''@'localhost' = PASSWORD('newpwd');
mysql> SET PASSWORD FOR ''@'host_name' = PASSWORD('newpwd');
```

In the second `SET PASSWORD` statement, replace *`host_name`* with the name of the server host. This is the name that is specified in the `Host` column of the non-`localhost` record for `root` in the `user` table. If you don't know what host name this is, issue the following statement before using `SET PASSWORD`:

```
mysql> SELECT Host, User FROM mysql.user;
```

Look for the record that has `root` in the `User` column and something other than `localhost` in the `Host` column. Then use that `Host` value in the second `SET PASSWORD` statement.

Anonymous Account Removal

If you prefer to remove the anonymous accounts instead, do so as follows:

```
shell> mysql -u root
mysql> DROP USER '';
```

The `DROP` statement applies both to Windows and to Unix. On Windows, if you want to remove only the anonymous account that has the same privileges as `root`, do this instead:

```
shell> mysql -u root
mysql> DROP USER ''@'localhost';
```

That account allows anonymous access but has full privileges, so removing it improves security.

`root` Account Password Assignment

You can assign passwords to the `root` accounts in several ways. The following discussion demonstrates three methods:

- Use the `SET PASSWORD` statement
- Use the `mysqladmin` command-line client program
- Use the `UPDATE` statement

To assign passwords using `SET PASSWORD`, connect to the server as `root` and issue `SET PASSWORD` statements. Be sure to encrypt the password using the `PASSWORD()` function.

For Windows, do this:

```
shell> mysql -u root
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('newpwd');
mysql> SET PASSWORD FOR 'root'@'%' = PASSWORD('newpwd');
```

For Unix, do this:

```
shell> mysql -u root
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('newpwd');
mysql> SET PASSWORD FOR 'root'@'host_name' = PASSWORD('newpwd');
```

In the second `SET PASSWORD` statement, replace *`host_name`* with the name of the server host. This is the same host name that you used when you assigned the anonymous account passwords.

If the `user` table contains an account with `User` and `Host` values of `'root'` and `'127.0.0.1'`, use an additional `SET PASSWORD` statement to set that account's password:

```
mysql> SET PASSWORD FOR 'root'@'127.0.0.1' = PASSWORD('newpwd');
```

To assign passwords to the `root` accounts using `mysqladmin`, execute the following commands:

```
shell> mysqladmin -u root password "newpwd"
shell> mysqladmin -u root -h host_name password "newpwd"
```

These commands apply both to Windows and to Unix. In the second command, replace *`host_name`* with the name of the server host. The double quotes around the password are not always necessary, but you should use them if the password contains spaces or other characters that are special to your command interpreter.

The `mysqladmin` method of setting the `root` account passwords does not set the password for the `'root'@'127.0.0.1'` account. To do so, use `SET PASSWORD` as shown earlier.

You can also use `UPDATE` to modify the `user` table directly. The following `UPDATE` statement assigns a password to all `root` accounts:

```
shell> mysql -u root
mysql> UPDATE mysql.user SET Password = PASSWORD('newpwd')
    ->     WHERE User = 'root';
mysql> FLUSH PRIVILEGES;
```

The `UPDATE` statement applies both to Windows and to Unix.

After the passwords have been set, you must supply the appropriate password whenever you connect to the server. For example, if you want to use `mysqladmin` to shut down the server, you can do so using this command:

```
shell> mysqladmin -u root -p shutdown
Enter password: (enter root password here)
```

> **Note**
>
> If you forget your `root` password after setting it up, How to Reset the Root Password, covers the procedure for resetting it.

To set up additional accounts, you can use the `GRANT` statement. For instructions, see Section 4.2, “Adding User Accounts”.

Chapter 3. The MySQL Access Privilege System

The primary function of the MySQL privilege system is to authenticate a user who connects from a given host and to associate that user with privileges on a database such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Additional functionality includes the ability to have anonymous users and to grant privileges for MySQL-specific functions such as `LOAD DATA INFILE` and administrative operations.

There are some things that you cannot do with the MySQL privilege system:

- You cannot explicitly specify that a given user should be denied access. That is, you cannot explicitly match a user and then refuse the connection.
- You cannot specify that a user has privileges to create or drop tables in a database but not to create or drop the database itself.
- A password applies globally to an account. You cannot associate a password with a specific object such as a database, table, or routine.

The user interface to the MySQL privilege system consists of SQL statements such as `CREATE USER`, `GRANT`, and `REVOKE`. See Account Management Statements.

Internally, the server stores privilege information in the grant tables of the `mysql` database (that is, in the database named `mysql`). The MySQL server reads the contents of these tables into memory when it starts and bases access-control decisions on the in-memory copies of the grant tables.

The MySQL privilege system ensures that all users may perform only the operations allowed to them. As a user, when you connect to a MySQL server, your identity is determined by *the host from which you connect* and *the user name you specify*. When you issue requests after connecting, the system grants privileges according to your identity and *what you want to do*.

MySQL considers both your host name and user name in identifying you because there is no reason to assume that a given user name belongs to the same person on all hosts. For example, the user `joe` who connects from `office.example.com` need not be the same person as the user `joe` who connects from `home.example.com`. MySQL handles this by allowing you to distinguish users on different hosts that happen to have the same name: You can grant one set of privileges for connections by `joe` from `office.example.com`, and a different set of privileges for connections by `joe` from `home.example.com`. To see what privileges a given account has, use the `SHOW GRANTS` statement. For example:

```
SHOW GRANTS FOR 'joe'@'office.example.com';
SHOW GRANTS FOR 'joe'@'home.example.com';
```

MySQL access control involves two stages when you run a client program that connects to the server:

Stage 1: The server accepts or rejects the connection based on your identity and whether you can verify your identity by supplying the correct password.

Stage 2: Assuming that you can connect, the server checks each statement you issue to determine whether you have sufficient privileges to perform it. For example, if you try to select rows from a table in a database or drop a table from the database, the server verifies that you have the `SELECT` privilege for the table or the `DROP` privilege for the database.

For a more detailed description of what happens during each stage, see Section 3.4, “Access Control, Stage 1: Connection Verification”, and Section 3.5, “Access Control, Stage 2: Request Verification”.

If your privileges are changed (either by yourself or someone else) while you are connected, those changes do not necessarily take effect immediately for the next statement that you issue. For details about the conditions under which the server reloads the grant tables, see Section 3.6, “When Privilege Changes Take Effect”.

For general security-related advice, see Chapter 1, *General Security Issues*. For help in diagnosing privilege-related problems, see Section 3.7, “Causes of Access-Denied Errors”.

3.1. Privileges Provided by MySQL

MySQL provides privileges that apply in different contexts and at different levels of operation:

- Administrative privileges enable users to manage operation of the MySQL server. These privileges are global because they are not specific to a particular database.
- Database privileges apply to a database and to all objects within it. These privileges can be granted for specific databases, or globally so that they apply to all databases.

- Privileges for database objects such as tables, indexes, views, and stored routines can be granted for specific objects within a database, for all objects of a given type within a database (for example, all tables in a database), or globally for all objects of a given type in all databases).

Information about account privileges is stored in the `user`, `db`, `host`, `tables_priv`, `columns_priv`, and `procs_priv` tables in the `mysql` database (see Section 3.2, "Privilege System Grant Tables"). The MySQL server reads the contents of these tables into memory when it starts and reloads them under the circumstances indicated in Section 3.6, "When Privilege Changes Take Effect". Access-control decisions are based on the in-memory copies of the grant tables.

Some releases of MySQL introduce changes to the structure of the grant tables to add new access privileges or features. Whenever you update to a new version of MySQL, you should update your grant tables to make sure that they have the current structure so that you can take advantage of any new capabilities. See `mysql_upgrade`.

The following table shows the privilege names used at the SQL level in the `GRANT` and `REVOKE` statements, along with the column name associated with each privilege in the grant tables and the context in which the privilege applies.

Privilege	Column	Context
`CREATE`	`Create_priv`	databases, tables, or indexes
`DROP`	`Drop_priv`	databases or tables
`GRANT OPTION`	`Grant_priv`	databases, tables, or stored routines
`REFERENCES`	`References_priv`	databases or tables
`EVENT`	`Event_priv`	databases
`ALTER`	`Alter_priv`	tables
`DELETE`	`Delete_priv`	tables
`INDEX`	`Index_priv`	tables
`INSERT`	`Insert_priv`	tables
`SELECT`	`Select_priv`	tables
`UPDATE`	`Update_priv`	tables
`CREATE TEMPORARY TABLES`	`Create_tmp_table_priv`	tables
`LOCK TABLES`	`Lock_tables_priv`	tables
`TRIGGER`	`Trigger_priv`	tables
`CREATE VIEW`	`Create_view_priv`	views
`SHOW VIEW`	`Show_view_priv`	views
`ALTER ROUTINE`	`Alter_routine_priv`	stored routines
`CREATE ROUTINE`	`Create_routine_priv`	stored routines
`EXECUTE`	`Execute_priv`	stored routines
`FILE`	`File_priv`	file access on server host
`CREATE TABLESPACE`	`Create_tablespace_priv`	server administration
`CREATE USER`	`Create_user_priv`	server administration
`PROCESS`	`Process_priv`	server administration
`RELOAD`	`Reload_priv`	server administration
`REPLICATION CLIENT`	`Repl_client_priv`	server administration
`REPLICATION SLAVE`	`Repl_slave_priv`	server administration
`SHOW DATABASES`	`Show_db_priv`	server administration
`SHUTDOWN`	`Shutdown_priv`	server administration
`SUPER`	`Super_priv`	server administration
`ALL [PRIVILEGES]`		server administration
`USAGE`		server administration

The following list provides a general description of each privilege available in MySQL. Particular SQL statements might have more specific privilege requirements than indicated here. If so, the description for the statement in question provides the details.

- The `ALL` or `ALL PRIVILEGES` privilege specifier is shorthand. It stands for "all privileges available at a given privilege

level" (except `GRANT OPTION`). For example, granting `ALL` at the global or table level grants all global privileges or all table-level privileges.

- The `ALTER` privilege enables use of `ALTER TABLE` to change the structure of or rename tables. (`ALTER TABLE` also requires the `INSERT` and `CREATE` privileges.)

  > **MySQL Enterprise**
  > In some circumstances, the `ALTER` privilege is entirely unnecessary — on slaves where there are no non-replicated tables, for instance. The MySQL Enterprise Monitor notifies subscribers when accounts have inappropriate privileges. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

- The `ALTER ROUTINE` privilege is needed to alter or drop stored routines (procedures and functions).
- The `CREATE` privilege enables creation of new databases and tables.
- The `CREATE ROUTINE` privilege is needed to create stored routines (procedures and functions).
- The `CREATE TABLESPACE` privilege is needed to create, alter, or drop tablespaces and log file groups. This privilege was added in MySQL 6.0.7.
- The `CREATE TEMPORARY TABLES` privilege enables the use of the keyword `TEMPORARY` in `CREATE TABLE` statements.
- The `CREATE USER` privilege enables use of `CREATE USER`, `DROP USER`, `RENAME USER`, and `REVOKE ALL PRIVILEGES`.
- The `CREATE VIEW` privilege enables use of `CREATE VIEW`.
- The `DELETE` privilege enables rows to be deleted from tables in a database.
- The `DROP` privilege enables you to drop (remove) existing databases, tables, and views. The `DROP` privilege is required in order to use the statement `ALTER TABLE ... DROP PARTITION` on a partitioned table. The `DROP` privilege is also required for `TRUNCATE TABLE`. *If you grant the `DROP` privilege for the `mysql` database to a user, that user can drop the database in which the MySQL access privileges are stored.*
- The `EVENT` privilege is required to create, alter, or drop events for the Event Scheduler.
- The `EXECUTE` privilege is required to execute stored routines (procedures and functions).
- The `FILE` privilege gives you permission to read and write files on the server host using the `LOAD DATA INFILE` and `SELECT ... INTO OUTFILE` statements. A user who has the `FILE` privilege can read any file on the server host that is either world-readable or readable by the MySQL server. (This implies the user can read any file in any database directory, because the server can access any of those files.) The `FILE` privilege also enables the user to create new files in any directory where the MySQL server has write access. As a security measure, the server will not overwrite existing files.
- The `GRANT OPTION` privilege enables you to give to other users or remove from other users those privileges that you yourself possess.
- The `INDEX` privilege enables you to create or drop (remove) indexes. `INDEX` applies to existing tables. If you have the `CREATE` privilege for a table, you can include index definitions in the `CREATE TABLE` statement.
- The `INSERT` privilege enables rows to be inserted into tables in a database. `INSERT` is also required for the `ANALYZE TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` table-maintenance statements.
- The `LOCK TABLES` privilege enables the use of explicit `LOCK TABLES` statements to lock tables for which you have the `SELECT` privilege. This includes the use of write locks, which prevents other sessions from reading the locked table.
- The `PROCESS` privilege pertains to display of information about the threads executing within the server (that is, information about the statements being executed by sessions). The privilege enables use of `SHOW PROCESSLIST` or `mysqladmin processlist` to see threads belonging to other accounts; you can always see your own threads.
- The `REFERENCES` privilege currently is unused.
- The `RELOAD` privilege enables use of the `FLUSH` statement. It also enables `mysqladmin` commands that are equivalent to `FLUSH` operations: `flush-hosts`, `flush-logs`, `flush-privileges`, `flush-status`, `flush-tables`, `flush-threads`, `refresh`, and `reload`.

  The `reload` command tells the server to reload the grant tables into memory. `flush-privileges` is a synonym for `reload`. The `refresh` command closes and reopens the log files and flushes all tables. The other `flush-`*`xxx`* commands perform functions similar to `refresh`, but are more specific and may be preferable in some instances. For example, if you want to flush just the log files, `flush-logs` is a better choice than `refresh`.

- The `REPLICATION CLIENT` privilege enables the use of `SHOW MASTER STATUS` and `SHOW SLAVE STATUS`.
- The `REPLICATION SLAVE` privilege should be granted to accounts that are used by slave servers to connect to the current server as their master. Without this privilege, the slave cannot request updates that have been made to databases on the master server.
- The `SELECT` privilege enables you to select rows from tables in a database. `SELECT` statements require the `SELECT` privilege only if they actually retrieve rows from a table. Some `SELECT` statements do not access tables and can be executed without permission for any database. For example, you can use `SELECT` as a simple calculator to evaluate expressions that make no reference to tables:

```
SELECT 1+1;
SELECT PI()*2;
```

- The `SHOW DATABASES` privilege enables the account to see database names by issuing the `SHOW DATABASE` statement. Accounts that do not have this privilege see only databases for which they have some privileges, and cannot use the statement at all if the server was started with the `--skip-show-database` option. Note that *any* global privilege is a privilege for the database.

> **MySQL Enterprise**
> The `SHOW DATABASES` privilege should be granted only to users who need to see all the databases on a MySQL server. Subscribers to the MySQL Enterprise Monitor are alerted when servers are started without the `--skip-show-database` option. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

- The `SHOW VIEW` privilege enables use of `SHOW CREATE VIEW`.
- The `SHUTDOWN` privilege enables use of the `mysqladmin shutdown` command. There is no corresponding SQL statement.
- The `SUPER` privilege enables use of `CHANGE MASTER TO`, `KILL` or `mysqladmin kill` to kill threads belonging to other accounts (you can always kill your own threads), `PURGE BINARY LOGS`, and `SET GLOBAL` statements, the `mysqladmin debug` command, and allows you to connect (once) even if the connection limit controlled by the `max_connections` system variable is reached.

  To create or alter stored functions if binary logging is enabled, you may also need the `SUPER` privilege, as described in Binary Logging of Stored Programs.
- The `TRIGGER` privilege enables you to create and drop triggers. You must have this privilege for a table to create or drop triggers for that table.
- The `UPDATE` privilege enables rows to be updated in tables in a database.
- The `USAGE` privilege specifier stands for “no privileges.” It is used at the global level with `GRANT` to modify account attributes such as resource limits or SSL characteristics without affecting existing account privileges.

It is a good idea to grant to an account only those privileges that it needs. You should exercise particular caution in granting the `FILE` and administrative privileges:

- The `FILE` privilege can be abused to read into a database table any files that the MySQL server can read on the server host. This includes all world-readable files and files in the server's data directory. The table can then be accessed using `SELECT` to transfer its contents to the client host.
- The `GRANT OPTION` privilege enables users to give their privileges to other users. Two users that have different privileges and with the `GRANT OPTION` privilege are able to combine privileges.
- The `ALTER` privilege may be used to subvert the privilege system by renaming tables.
- The `SHUTDOWN` privilege can be abused to deny service to other users entirely by terminating the server.
- The `PROCESS` privilege can be used to view the plain text of currently executing statements, including statements that set or change passwords.
- The `SUPER` privilege can be used to terminate other sessions or change how the server operates.
- Privileges granted for the `mysql` database itself can be used to change passwords and other access privilege information. Passwords are stored encrypted, so a malicious user cannot simply read them to know the plain text password. However, a user with write access to the `user` table `Password` column can change an account's password, and then connect to the MySQL server using that account.

MySQL Enterprise
Accounts with unnecessary global privileges constitute a security risk. Subscribers to the MySQL Enterprise Monitor are automatically alerted to the existence of such accounts. For detailed information, see http://www.mysql.com/products/enterprise/advisors.html.

3.2. Privilege System Grant Tables

Normally, you manipulate the contents of the grant tables indirectly by using statements such as `GRANT` and `REVOKE` to set up accounts and control the privileges available to each one. See Account Management Statements. The discussion here describes the underlying structure of the grant tables and how the server uses their contents when interacting with clients.

Each grant table contains scope columns and privilege columns:

- Scope columns determine the scope of each row (entry) in the tables; that is, the context in which the row applies. For example, a `user` table row with `Host` and `User` values of `'thomas.loc.gov'` and `'bob'` would be used for authenticating connections made to the server from the host `thomas.loc.gov` by a client that specifies a user name of `bob`. Similarly, a `db` table row with `Host`, `User`, and `Db` column values of `'thomas.loc.gov'`, `'bob'` and `'reports'` would be used when `bob` connects from the host `thomas.loc.gov` to access the `reports` database. The `tables_priv` and `columns_priv` tables contain scope columns indicating tables or table/column combinations to which each row applies. The `procs_priv` scope columns indicate the stored routine to which each row applies.
- Privilege columns indicate which privileges are granted by a table row; that is, what operations can be performed. The server combines the information in the various grant tables to form a complete description of a user's privileges. Section 3.5, “Access Control, Stage 2: Request Verification”, describes the rules that are used to do this.

The server uses the grant tables in the following manner:

- The `user` table scope columns determine whether to reject or allow incoming connections. For allowed connections, any privileges granted in the `user` table indicate the user's global (superuser) privileges. Any privilege granted in this table applies to *all* databases on the server.

  Note

  Because any global privilege is considered a privilege for all databases, any global privilege enables a user to see all database names with `SHOW DATABASES` or by examining the `SCHEMATA` table of `INFORMATION_SCHEMA`.

- The `db` table scope columns determine which users can access which databases from which hosts. The privilege columns determine which operations are allowed. A privilege granted at the database level applies to the database and to all objects in the database, such as tables and stored programs.
- The `host` table is used in conjunction with the `db` table when you want a given `db` table row to apply to several hosts. For example, if you want a user to be able to use a database from several hosts in your network, leave the `Host` value empty in the user's `db` table row, then populate the `host` table with a row for each of those hosts. This mechanism is described more detail in Section 3.5, “Access Control, Stage 2: Request Verification”.

  Note

  The `host` table must be modified directly with statements such as `INSERT`, `UPDATE`, and `DELETE`. It is not affected by statements such as `GRANT` and `REVOKE` that modify the grant tables indirectly. Most MySQL installations need not use this table at all.

- The `tables_priv` and `columns_priv` tables are similar to the `db` table, but are more fine-grained: They apply at the table and column levels rather than at the database level. A privilege granted at the table level applies to the table and to all its columns. A privilege granted at the column level applies only to a specific column.
- The `procs_priv` table applies to stored routines. A privilege granted at the routine level applies only to a single routine.

The server uses the `user`, `db`, and `host` tables in the `mysql` database at both the first and second stages of access control (see Chapter 3, *The MySQL Access Privilege System*). The columns in the `user` and `db` tables are shown here. The `host` table is similar to the `db` table but has a specialized use as described in Section 3.5, “Access Control, Stage 2: Request Verification”.

Table Name	user	db
Scope columns	`Host`	`Host`

	User	Db
	Password	User
Privilege columns	Select_priv	Select_priv
	Insert_priv	Insert_priv
	Update_priv	Update_priv
	Delete_priv	Delete_priv
	Index_priv	Index_priv
	Alter_priv	Alter_priv
	Create_priv	Create_priv
	Drop_priv	Drop_priv
	Grant_priv	Grant_priv
	Create_view_priv	Create_view_priv
	Show_view_priv	Show_view_priv
	Create_routine_priv	Create_routine_priv
	Alter_routine_priv	Alter_routine_priv
	Execute_priv	Execute_priv
	Trigger_priv	Trigger_priv
	Event_priv	Event_priv
	Create_tmp_table_priv	Create_tmp_table_priv
	Lock_tables_priv	Lock_tables_priv
	References_priv	References_priv
	Reload_priv	
	Shutdown_priv	
	Process_priv	
	File_priv	
	Show_db_priv	
	Super_priv	
	Repl_slave_priv	
	Repl_client_priv	
	Create_user_priv	
	Create_tablespace_priv	
Security columns	ssl_type	
	ssl_cipher	
	x509_issuer	
	x509_subject	
Resource control columns	max_questions	
	max_updates	
	max_connections	
	max_user_connections	

The Create_tablespace_priv column was added in MySQL 6.0.7.

During the second stage of access control, the server performs request verification to make sure that each client has sufficient privileges for each request that it issues. In addition to the user, db, and host grant tables, the server may also consult the tables_priv and columns_priv tables for requests that involve tables. The latter tables provide finer privilege control at the table and column levels. They have the columns shown in the following table.

Table Name	tables_priv	columns_priv
Scope columns	Host	Host
	Db	Db

	`User`	`User`
	`Table_name`	`Table_name`
		`Column_name`
Privilege columns	`Table_priv`	`Column_priv`
	`Column_priv`	
Other columns	`Timestamp`	`Timestamp`
	`Grantor`	

The `Timestamp` and `Grantor` columns currently are unused and are discussed no further here.

For verification of requests that involve stored routines, the server may consult the `procs_priv` table, which has the columns shown in the following table.

Table Name	**procs_priv**
Scope columns	`Host`
	`Db`
	`User`
	`Routine_name`
	`Routine_type`
Privilege columns	`Proc_priv`
Other columns	`Timestamp`
	`Grantor`

The `Routine_type` column is an `ENUM` column with values of `'FUNCTION'` or `'PROCEDURE'` to indicate the type of routine the row refers to. This column enables privileges to be granted separately for a function and a procedure with the same name.

The `Timestamp` and `Grantor` columns currently are unused and are discussed no further here.

Scope columns in the grant tables contain strings. They are declared as shown here; the default value for each is the empty string.

Column Name	**Type**
`Host`	`CHAR(60)`
`User`	`CHAR(16)`
`Password`	`CHAR(41)`
`Db`	`CHAR(64)`
`Table_name`	`CHAR(64)`
`Column_name`	`CHAR(64)`
`Routine_name`	`CHAR(64)`

For access-checking purposes, comparisons of `User`, `Password`, `Db`, and `Table_name` values are case sensitive. Comparisons of `Host`, `Column_name`, and `Routine_name` values are not case sensitive.

In the `user`, `db`, and `host` tables, each privilege is listed in a separate column that is declared as `ENUM('N','Y') DEFAULT 'N'`. In other words, each privilege can be disabled or enabled, with the default being disabled.

In the `tables_priv`, `columns_priv`, and `procs_priv` tables, the privilege columns are declared as `SET` columns. Values in these columns can contain any combination of the privileges controlled by the table. Only those privileges listed in the column value are enabled.

Table Name	**Column Name**	**Possible Set Elements**
`tables_priv`	`Table_priv`	`'Select', 'Insert', 'Update', 'Delete', 'Create', 'Drop', 'Grant', 'References', 'Index', 'Alter', 'Create View', 'Show view', 'Trigger'`
`tables_priv`	`Column_priv`	`'Select', 'Insert', 'Update', 'References'`
`columns_priv`	`Column_priv`	`'Select', 'Insert', 'Update', 'References'`

procs_priv	Proc_priv	'Execute', 'Alter Routine', 'Grant'

Administrative privileges (such as `RELOAD` or `SHUTDOWN`) are specified only in the `user` table. Administrative operations are operations on the server itself and are not database-specific, so there is no reason to list these privileges in the other grant tables. Consequently, to determine whether you can perform an administrative operation, the server need consult only the `user` table.

The `FILE` privilege also is specified only in the `user` table. It is not an administrative privilege as such, but your ability to read or write files on the server host is independent of the database you are accessing.

The `mysqld` server reads the contents of the grant tables into memory when it starts. You can tell it to reload the tables by issuing a `FLUSH PRIVILEGES` statement or executing a `mysqladmin flush-privileges` or `mysqladmin reload` command. Changes to the grant tables take effect as indicated in Section 3.6, “When Privilege Changes Take Effect”.

When you modify an account's privileges, it is a good idea to verify that the changes set up privileges the way you want. To check the privileges for a given account, use the `SHOW GRANTS` statement (see `SHOW GRANTS` Syntax). For example, to determine the privileges that are granted to an account with user name and host name values of `bob` and `pc84.example.com`, use this statement:

```
SHOW GRANTS FOR 'bob'@'pc84.example.com';
```

3.3. Specifying Account Names

MySQL account names consist of a user name and a host name. This enables creation of accounts for users with the same name who can connect from different hosts. This section describes how to write account names, including special values and wildcard rules.

Within SQL statements such as `CREATE USER`, `GRANT`, and `SET PASSWORD`, account names are written using the following rules:

- Syntax for account names is `'user_name'@'host_name'`.
- An account name consisting only of a user name is equivalent to `'user_name'@'%'`. For example, `'me'` is equivalent to `'me'@'%'`.
- The user name and host name need not be quoted if they are legal as unquoted identifiers. Quotes are necessary to specify a `user_name` string containing special characters (such as “`-`”), or a `host_name` string containing special characters or wildcard characters (such as “`%`”); for example, `'test-user'@'%.com'`.
- Quote user names and host names as identifiers or as strings, using either backticks (“`` ` ``”), single quotes (“`'`”), or double quotes (“`"`”).
- The user name and host name parts, if quoted, must be quoted separately. That is, write `'me'@'localhost'`, not `'me@localhost'`; the latter is interpreted as `'me@localhost'@'%'`.

Account names are stored in grant tables using separate columns for the user name and host name parts:

- The `user` table contains one row for each account. The `User` and `Host` columns store the user name and host name. Another column, `Password`, stores the account password. This table also indicates which global privileges the account has.
- Other grant tables indicate privileges an account has for databases and objects within databases. These tables have `User` and `Host` columns to store the account name. Each row in these tables associates with the account in the `user` table that has the same `User` and `Host` values.

For additional detail about grant table structure, see Section 3.2, “Privilege System Grant Tables”.

User names and host names have certain special values or wildcard conventions, as described following.

A user name is either a non-blank value that literally matches the user name for incoming connection attempts, or a blank value (empty string) that matches any user name. An account with a blank user name is an anonymous user. To specify an anonymous user in SQL statements, use a quoted empty user name part, such as `''@'localhost'`.

The host part of an account name can take many forms, and wildcards are allowed:

- A host value can be a host name or an IP number. `'localhost'` indicates the local host. `'127.0.0.1'` indicates the loop-

back interface.

- You can use the wildcard characters "%" and "_" in host values. These have the same meaning as for pattern-matching operations performed with the `LIKE` operator. For example, a host value of `'%'` matches any host name, whereas a value of `'%.mysql.com'` matches any host in the `mysql.com` domain. `'192.168.1.%'` matches any host in the 192.168.1 class C network.

  Because you can use IP wildcard values in host values (for example, `'192.168.1.%'` to match every host on a subnet), someone could try to exploit this capability by naming a host `192.168.1.somewhere.com`. To foil such attempts, MySQL disallows matching on host names that start with digits and a dot. Thus, if you have a host named something like `1.2.example.com`, its name never matches the host part of account names. An IP wildcard value can match only IP numbers, not host names.

  > **MySQL Enterprise**
  > An overly broad host specifier such as "%" constitutes a security risk. The MySQL Enterprise Monitor provides safeguards against this kind of vulnerability. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

- For host values specified as IP numbers, you can specify a netmask indicating how many address bits to use for the network number. The syntax is *`host_ip/netmask`*. For example:

  ```
  CREATE USER 'david'@'192.58.197.0/255.255.255.0';
  ```

  This enables `david` to connect from any client host having an IP number *`client_ip`* for which the following condition is true:

  ```
  client_ip & netmask = host_ip
  ```

  That is, for the `CREATE USER` statement just shown:

  ```
  client_ip & 255.255.255.0 = 192.58.197.0
  ```

  IP numbers that satisfy this condition and can connect to the MySQL server are those in the range from `192.58.197.0` to `192.58.197.255`.

  The netmask can only be used to tell the server to use 8, 16, 24, or 32 bits of the address. Examples:

  - `192.0.0.0/255.0.0.0`: anything on the 192 class A network
  - `192.168.0.0/255.255.0.0`: anything on the 192.168 class B network
  - `192.168.1.0/255.255.255.0`: anything on the 192.168.1 class C network
  - `192.168.1.1`: only this specific IP

  The following netmask (28 bits) will not work:

  ```
  192.168.0.1/255.255.255.240
  ```

3.4. Access Control, Stage 1: Connection Verification

When you attempt to connect to a MySQL server, the server accepts or rejects the connection based on your identity and whether you can verify your identity by supplying the correct password. If not, the server denies access to you completely. Otherwise, the server accepts the connection, and then enters Stage 2 and waits for requests.

Your identity is based on two pieces of information:

- The client host from which you connect
- Your MySQL user name

Identity checking is performed using the three `user` table scope columns (`Host`, `User`, and `Password`). The server accepts the connection only if the `Host` and `User` columns in some `user` table row match the client host name and user name and the client supplies the password specified in that row. The rules for allowable `Host` and `User` values are given in Section 3.3, "Specifying Account Names".

If the `User` column value is non-blank, the user name in an incoming connection must match exactly. If the `User` value is blank, it matches any user name. If the `user` table row that matches an incoming connection has a blank user name, the user is considered to be an anonymous user with no name, not a user with the name that the client actually specified. This means that a blank user name is used for all further access checking for the duration of the connection (that is, during Stage 2).

The `Password` column can be blank. This is not a wildcard and does not mean that any password matches. It means that the user must connect without specifying a password.

Non-blank `Password` values in the `user` table represent encrypted passwords. MySQL does not store passwords in plaintext form for anyone to see. Rather, the password supplied by a user who is attempting to connect is encrypted (using the `PASSWORD()` function). The encrypted password then is used during the connection process when checking whether the password is correct. (This is done without the encrypted password ever traveling over the connection.) See Section 4.1, “User Names and Passwords”.

From MySQL's point of view, the encrypted password is the *real* password, so you should never give anyone access to it. In particular, *do not give non-administrative users read access to tables in the `mysql` database*.

The following table shows how various combinations of `Host` and `User` values in the `user` table apply to incoming connections.

`Host` Value	`User` Value	Allowable Connections
`'thomas.loc.gov'`	`'fred'`	`fred`, connecting from `thomas.loc.gov`
`'thomas.loc.gov'`	`''`	Any user, connecting from `thomas.loc.gov`
`'%'`	`'fred'`	`fred`, connecting from any host
`'%'`	`''`	Any user, connecting from any host
`'%.loc.gov'`	`'fred'`	`fred`, connecting from any host in the `loc.gov` domain
`'x.y.%'`	`'fred'`	`fred`, connecting from `x.y.net`, `x.y.com`, `x.y.edu`, and so on; this is probably not useful
`'144.155.166.177'`	`'fred'`	`fred`, connecting from the host with IP address `144.155.166.177`
`'144.155.166.%'`	`'fred'`	`fred`, connecting from any host in the `144.155.166` class C subnet
`'144.155.166.0/255.255.255.0'`	`'fred'`	Same as previous example

It is possible for the client host name and user name of an incoming connection to match more than one row in the `user` table. The preceding set of examples demonstrates this: Several of the entries shown match a connection from `thomas.loc.gov` by `fred`.

When multiple matches are possible, the server must determine which of them to use. It resolves this issue as follows:

- Whenever the server reads the `user` table into memory, it sorts the rows.
- When a client attempts to connect, the server looks through the rows in sorted order.
- The server uses the first row that matches the client host name and user name.

To see how this works, suppose that the `user` table looks like this:

```
+-----------+----------+-
| Host      | User     | ...
+-----------+----------+-
| %         | root     | ...
| %         | jeffrey  | ...
| localhost | root     | ...
| localhost |          | ...
+-----------+----------+-
```

When the server reads the table into memory, it orders the rows with the most-specific `Host` values first. Literal host names and IP numbers are the most specific. (The specificity if a literal IP number is not affected by whether it has a netmask, so `192.168.1.13` and `192.168.1.0/255.255.255.0` are considered equally specific.) The pattern `'%'` means “any host” and is least specific. Rows with the same `Host` value are ordered with the most-specific `User` values first (a blank `User` value means “any user” and is least specific). For the `user` table just shown, the result after sorting looks like this:

```
+-----------+----------+-
| Host      | User     | ...
+-----------+----------+-
| localhost | root     | ...
| localhost |          | ...
| %         | jeffrey  | ...
```

```
| %         | root     | ...
+-----------+----------+-
```

When a client attempts to connect, the server looks through the sorted rows and uses the first match found. For a connection from `localhost` by `jeffrey`, two of the rows from the table match: the one with `Host` and `User` values of `'localhost'` and `''`, and the one with values of `'%'` and `'jeffrey'`. The `'localhost'` row appears first in sorted order, so that is the one the server uses.

Here is another example. Suppose that the `user` table looks like this:

```
+---------------+----------+-
| Host          | User     | ...
+---------------+----------+-
| %             | jeffrey  | ...
| thomas.loc.gov |          | ...
+---------------+----------+-
```

The sorted table looks like this:

```
+---------------+----------+-
| Host          | User     | ...
+---------------+----------+-
| thomas.loc.gov |          | ...
| %             | jeffrey  | ...
+---------------+----------+-
```

A connection by `jeffrey` from `thomas.loc.gov` is matched by the first row, whereas a connection by `jeffrey` from any host is matched by the second.

> **Note**
>
> It is a common misconception to think that, for a given user name, all rows that explicitly name that user are used first when the server attempts to find a match for the connection. This is not true. The preceding example illustrates this, where a connection from `thomas.loc.gov` by `jeffrey` is first matched not by the row containing `'jeffrey'` as the `User` column value, but by the row with no user name. As a result, `jeffrey` is authenticated as an anonymous user, even though he specified a user name when connecting.

If you are able to connect to the server, but your privileges are not what you expect, you probably are being authenticated as some other account. To find out what account the server used to authenticate you, use the `CURRENT_USER()` function. (See Information Functions.) It returns a value in *`user_name@host_name`* format that indicates the `User` and `Host` values from the matching `user` table row. Suppose that `jeffrey` connects and issues the following query:

```
mysql> SELECT CURRENT_USER();
+----------------+
| CURRENT_USER() |
+----------------+
| @localhost     |
+----------------+
```

The result shown here indicates that the matching `user` table row had a blank `User` column value. In other words, the server is treating `jeffrey` as an anonymous user.

Another way to diagnose authentication problems is to print out the `user` table and sort it by hand to see where the first match is being made.

3.5. Access Control, Stage 2: Request Verification

After you establish a connection, the server enters Stage 2 of access control. For each request that you issue via that connection, the server determines what operation you want to perform, then checks whether you have sufficient privileges to do so. This is where the privilege columns in the grant tables come into play. These privileges can come from any of the `user`, `db`, `host`, `tables_priv`, `columns_priv`, or `procs_priv` tables. (You may find it helpful to refer to Section 3.2, “Privilege System Grant Tables”, which lists the columns present in each of the grant tables.)

The `user` table grants privileges that are assigned to you on a global basis and that apply no matter what the default database is. For example, if the `user` table grants you the `DELETE` privilege, you can delete rows from any table in any database on the server host! In other words, `user` table privileges are superuser privileges. It is wise to grant privileges in the `user` table only to superusers such as database administrators. For other users, you should leave all privileges in the `user` table set to `'N'` and grant privileges at more specific levels only. You can grant privileges for particular databases, tables, columns, or routines.

The `db` and `host` tables grant database-specific privileges. Values in the scope columns of these tables can take the following forms:

- A blank `User` value in the `db` table matches the anonymous user. A non-blank value matches literally; there are no wildcards in user names.
- The wildcard characters "`%`" and "`_`" can be used in the `Host` and `Db` columns of either table. These have the same meaning as for pattern-matching operations performed with the `LIKE` operator. If you want to use either character literally when granting privileges, you must escape it with a backslash. For example, to include the underscore character ("`_`") as part of a database name, specify it as "`\_`" in the `GRANT` statement.
- A `'%' Host` value in the `db` table means "any host." A blank `Host` value in the `db` table means "consult the `host` table for further information" (a process that is described later in this section).
- A `'%'` or blank `Host` value in the `host` table means "any host."
- A `'%'` or blank `Db` value in either table means "any database."

The server reads the `db` and `host` tables into memory and sorts them at the same time that it reads the `user` table. The server sorts the `db` table based on the `Host`, `Db`, and `User` scope columns, and sorts the `host` table based on the `Host` and `Db` scope columns. As with the `user` table, sorting puts the most-specific values first and least-specific values last, and when the server looks for matching entries, it uses the first match that it finds.

The `tables_priv`, `columns_priv`, and `procs_priv` tables grant table-specific, column-specific, and routine-specific privileges. Values in the scope columns of these tables can take the following forms:

- The wildcard characters "`%`" and "`_`" can be used in the `Host` column. These have the same meaning as for pattern-matching operations performed with the `LIKE` operator.
- A `'%'` or blank `Host` value means "any host."
- The `Db`, `Table_name`, `Column_name`, and `Routine_name` columns cannot contain wildcards or be blank.

The server sorts the `tables_priv`, `columns_priv`, and `procs_priv` tables based on the `Host`, `Db`, and `User` columns. This is similar to `db` table sorting, but simpler because only the `Host` column can contain wildcards.

The server uses the sorted tables to verify each request that it receives. For requests that require administrative privileges such as `SHUTDOWN` or `RELOAD`, the server checks only the `user` table row because that is the only table that specifies administrative privileges. The server grants access if the row allows the requested operation and denies access otherwise. For example, if you want to execute `mysqladmin shutdown` but your `user` table row doesn't grant the `SHUTDOWN` privilege to you, the server denies access without even checking the `db` or `host` tables. (They contain no `Shutdown_priv` column, so there is no need to do so.)

For database-related requests (`INSERT`, `UPDATE`, and so on), the server first checks the user's global (superuser) privileges by looking in the `user` table row. If the row allows the requested operation, access is granted. If the global privileges in the `user` table are insufficient, the server determines the user's database-specific privileges by checking the `db` and `host` tables:

1. The server looks in the `db` table for a match on the `Host`, `Db`, and `User` columns. The `Host` and `User` columns are matched to the connecting user's host name and MySQL user name. The `Db` column is matched to the database that the user wants to access. If there is no row for the `Host` and `User`, access is denied.
2. If there is a matching `db` table row and its `Host` column is not blank, that row defines the user's database-specific privileges.
3. If the matching `db` table row's `Host` column is blank, it signifies that the `host` table enumerates which hosts should be allowed access to the database. In this case, a further lookup is done in the `host` table to find a match on the `Host` and `Db` columns. If no `host` table row matches, access is denied. If there is a match, the user's database-specific privileges are computed as the intersection (*not* the union!) of the privileges in the `db` and `host` table entries; that is, the privileges that are `'Y'` in both entries. (This way you can grant general privileges in the `db` table row and then selectively restrict them on a host-by-host basis using the `host` table entries.)

After determining the database-specific privileges granted by the `db` and `host` table entries, the server adds them to the global privileges granted by the `user` table. If the result allows the requested operation, access is granted. Otherwise, the server successively checks the user's table and column privileges in the `tables_priv` and `columns_priv` tables, adds those to the user's privileges, and allows or denies access based on the result. For stored-routine operations, the server uses the `procs_priv` table rather than `tables_priv` and `columns_priv`.

Expressed in boolean terms, the preceding description of how a user's privileges are calculated may be summarized like this:

```
global privileges
OR (database privileges AND host privileges)
OR table privileges
```

```
OR column privileges
OR routine privileges
```

It may not be apparent why, if the global `user` row privileges are initially found to be insufficient for the requested operation, the server adds those privileges to the database, table, and column privileges later. The reason is that a request might require more than one type of privilege. For example, if you execute an `INSERT INTO ... SELECT` statement, you need both the `INSERT` and the `SELECT` privileges. Your privileges might be such that the `user` table row grants one privilege and the `db` table row grants the other. In this case, you have the necessary privileges to perform the request, but the server cannot tell that from either table by itself; the privileges granted by the entries in both tables must be combined.

The `host` table is not affected by the `GRANT` or `REVOKE` statements, so it is unused in most MySQL installations. If you modify it directly, you can use it for some specialized purposes, such as to maintain a list of secure servers on the local network that are granted all privileges.

You can also use the `host` table to indicate hosts that are *not* secure. Suppose that you have a machine `public.your.domain` that is located in a public area that you do not consider secure. You can enable access to all hosts on your network except that machine by using `host` table entries like this:

```
+--------------------+----+-
| Host               | Db | ...
+--------------------+----+-
| public.your.domain | %  | ... (all privileges set to 'N')
| %.your.domain      | %  | ... (all privileges set to 'Y')
+--------------------+----+-
```

3.6. When Privilege Changes Take Effect

When `mysqld` starts, it reads all grant table contents into memory. The in-memory tables become effective for access control at that point.

If you modify the grant tables indirectly using account-management statements such as `GRANT`, `REVOKE`, or `SET PASSWORD`, the server notices these changes and loads the grant tables into memory again immediately.

If you modify the grant tables directly using statements such as `INSERT`, `UPDATE`, or `DELETE`, your changes have no effect on privilege checking until you either restart the server or tell it to reload the tables. If you change the grant tables directly but forget to reload them, your changes have *no effect* until you restart the server. This may leave you wondering why your changes do not seem to make any difference!

To tell the sever to reload the grant tables, perform a flush-privileges operation. This can be done by issuing a `FLUSH PRIVILEGES` statement or by executing a `mysqladmin flush-privileges` or `mysqladmin reload` command.

When the server reloads the grant tables, privileges for each existing client connection are affected as follows:

- Table and column privilege changes take effect with the client's next request.
- Database privilege changes take effect the next time the client executes a `USE` *`db_name`* statement.

  > **Note**
  >
  > Client applications may cache the database name; thus, this effect may not be visible to them without actually changing to a different database or flushing the privileges.

- Global privileges and passwords are unaffected for a connected client. These changes take effect only for subsequent connections.

If the server is started with the `--skip-grant-tables` option, it does not read the grant tables or implement any access control. Anyone can connect and do anything. To cause a server thus started to read the tables and enable access checking, flush the privileges.

3.7. Causes of Access-Denied Errors

If you encounter problems when you try to connect to the MySQL server, the following items describe some courses of action you can take to correct the problem.

- Make sure that the server is running. If it is not, clients cannot connect to it. For example, if an attempt to connect to the server fails with a message such as one of those following, one cause might be that the server is not running:

```
shell> mysql
ERROR 2003: Can't connect to MySQL server on 'host_name' (111)
shell> mysql
ERROR 2002: Can't connect to local MySQL server through socket
'/tmp/mysql.sock' (111)
```

- It might be that the server is running, but you are trying to connect using a TCP/IP port, named pipe, or Unix socket file different from the one on which the server is listening. To correct this when you invoke a client program, specify a `--port` option to indicate the proper port number, or a `--socket` option to indicate the proper named pipe or Unix socket file. To find out where the socket file is, you can use this command:

```
shell> netstat -ln | grep mysql
```

- Make sure that the server has not been configured to ignore network connections or (if you are attempting to connect remotely) that it has not been configured to listen only locally on its network interfaces. If the server was started with `--skip-networking`, it will not accept TCP/IP connections at all. If the server was started with `--bind-address=127.0.0.1`, it will listen for TCP/IP connections only locally on the loopback interface and will not accept remote connections.

- Check to make sure that there is no firewall blocking access to MySQL. Your firewall may be configured on the basis of the application being executed, or the port number used by MySQL for communication (3306 by default). Under Linux or Unix, check your IP tables (or similar) configuration to ensure that the port has not been blocked. Under Windows, applications such as ZoneAlarm or the Windows XP personal firewall may need to be configured not to block the MySQL port.

- The grant tables must be properly set up so that the server can use them for access control. For some distribution types (such as binary distributions on Windows, or RPM distributions on Linux), the installation process initializes the `mysql` database containing the grant tables. For distributions that do not do this, you must initialize the grant tables manually by running the `mysql_install_db` script. For details, see Section 2.2, “Unix Post-Installation Procedures”.

  To determine whether you need to initialize the grant tables, look for a `mysql` directory under the data directory. (The data directory normally is named `data` or `var` and is located under your MySQL installation directory.) Make sure that you have a file named `user.MYD` in the `mysql` database directory. If not, execute the `mysql_install_db` script. After running this script and starting the server, test the initial privileges by executing this command:

```
shell> mysql -u root test
```

  The server should let you connect without error.

- After a fresh installation, you should connect to the server and set up your users and their access permissions:

```
shell> mysql -u root mysql
```

  The server should let you connect because the MySQL `root` user has no password initially. That is also a security risk, so setting the password for the `root` accounts is something you should do while you're setting up your other MySQL accounts. For instructions on setting the initial passwords, see Section 2.3, “Securing the Initial MySQL Accounts”.

  MySQL Enterprise
  The MySQL Enterprise Monitor enforces security-related best practices. For example, subscribers are alerted whenever there is any account without a password. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

- If you have updated an existing MySQL installation to a newer version, did you run the `mysql_upgrade` script? If not, do so. The structure of the grant tables changes occasionally when new capabilities are added, so after an upgrade you should always make sure that your tables have the current structure. For instructions, see `mysql_upgrade`.

- If a client program receives the following error message when it tries to connect, it means that the server expects passwords in a newer format than the client is capable of generating:

```
shell> mysql
Client does not support authentication protocol requested
by server; consider upgrading MySQL client
```

  For information on how to deal with this, see Section 4.6.3, “Password Hashing in MySQL”, and `Client does not support authentication protocol`.

- Remember that client programs use connection parameters specified in option files or environment variables. If a client program seems to be sending incorrect default connection parameters when you have not specified them on the command line, check any applicable option files and your environment. For example, if you get `Access denied` when you run a client without any options, make sure that you have not specified an old password in any of your option files!

You can suppress the use of option files by a client program by invoking it with the `--no-defaults` option. For example:

```
shell> mysqladmin --no-defaults -u root version
```

The option files that clients use are listed in Using Option Files. Environment variables are listed in Environment Variables.

- If you get the following error, it means that you are using an incorrect `root` password:

```
shell> mysqladmin -u root -pxxxx ver
Access denied for user 'root'@'localhost' (using password: YES)
```

If the preceding error occurs even when you have not specified a password, it means that you have an incorrect password listed in some option file. Try the `--no-defaults` option as described in the previous item.

For information on changing passwords, see Section 4.5, “Assigning Account Passwords”.

If you have lost or forgotten the `root` password, see How to Reset the Root Password.

- If you change a password by using `SET PASSWORD`, `INSERT`, or `UPDATE`, you must encrypt the password using the `PASSWORD()` function. If you do not use `PASSWORD()` for these statements, the password will not work. For example, the following statement assigns a password, but fails to encrypt it, so the user is not able to connect afterward:

```
SET PASSWORD FOR 'abe'@'host_name' = 'eagle';
```

Instead, set the password like this:

```
SET PASSWORD FOR 'abe'@'host_name' = PASSWORD('eagle');
```

The `PASSWORD()` function is unnecessary when you specify a password using the `CREATE USER` or `GRANT` statements or the `mysqladmin password` command. Each of those automatically uses `PASSWORD()` to encrypt the password. See Section 4.5, “Assigning Account Passwords”, and `CREATE USER` Syntax.

- `localhost` is a synonym for your local host name, and is also the default host to which clients try to connect if you specify no host explicitly.

To avoid this problem on such systems, you can use a `--host=127.0.0.1` option to name the server host explicitly. This will make a TCP/IP connection to the local `mysqld` server. You can also use TCP/IP by specifying a `--host` option that uses the actual host name of the local host. In this case, the host name must be specified in a `user` table row on the server host, even though you are running the client program on the same host as the server.

- The `Access denied` error message tells you who you are trying to log in as, the client host from which you are trying to connect, and whether you were using a password. Normally, you should have one row in the `user` table that exactly matches the host name and user name that were given in the error message. For example, if you get an error message that contains `using password: NO`, it means that you tried to log in without a password.

- If you get an `Access denied` error when trying to connect to the database with `mysql -u user_name`, you may have a problem with the `user` table. Check this by executing `mysql -u root mysql` and issuing this SQL statement:

```
SELECT * FROM user;
```

The result should include a row with the `Host` and `User` columns matching your client's host name and your MySQL user name.

- If the following error occurs when you try to connect from a host other than the one on which the MySQL server is running, it means that there is no row in the `user` table with a `Host` value that matches the client host:

```
Host ... is not allowed to connect to this MySQL server
```

You can fix this by setting up an account for the combination of client host name and user name that you are using when trying to connect.

If you do not know the IP number or host name of the machine from which you are connecting, you should put a row with `'%'` as the `Host` column value in the `user` table. After trying to connect from the client machine, use a `SELECT USER()` query to see how you really did connect. Then change the `'%'` in the `user` table row to the actual host name that shows up in the log. Otherwise, your system is left insecure because it allows connections from any host for the given user name.

On Linux, another reason that this error might occur is that you are using a binary MySQL version that is compiled with a different version of the `glibc` library than the one you are using. In this case, you should either upgrade your operating system or

`glibc`, or download a source distribution of MySQL version and compile it yourself. A source RPM is normally trivial to compile and install, so this is not a big problem.

- If you specify a host name when trying to connect, but get an error message where the host name is not shown or is an IP number, it means that the MySQL server got an error when trying to resolve the IP number of the client host to a name:

```
shell> mysqladmin -u root -pxxxx -h some_hostname ver
Access denied for user 'root'@'' (using password: YES)
```

  If you try to connect as `root` and get the following error, it means that you do not have a row in the `user` table with a `User` column value of `'root'` and that `mysqld` cannot resolve the host name for your client:

```
Access denied for user ''@'unknown'
```

  These errors indicate a DNS problem. To fix it, execute `mysqladmin flush-hosts` to reset the internal DNS host name cache. See How MySQL Uses DNS.

  Some permanent solutions are:

  - Determine what is wrong with your DNS server and fix it.
  - Specify IP numbers rather than host names in the MySQL grant tables.
  - Put an entry for the client machine name in `/etc/hosts` on Unix or `\windows\hosts` on Windows.
  - Start `mysqld` with the `--skip-name-resolve` option.
  - Start `mysqld` with the `--skip-host-cache` option.
  - On Unix, if you are running the server and the client on the same machine, connect to `localhost`. Unix connections to `localhost` use a Unix socket file rather than TCP/IP.
  - On Windows, if you are running the server and the client on the same machine and the server supports named pipe connections, connect to the host name `.` (period). Connections to `.` use a named pipe rather than TCP/IP.

- If `mysql -u root test` works but `mysql -h your_hostname -u root test` results in `Access denied` (where *`your_hostname`* is the actual host name of the local host), you may not have the correct name for your host in the `user` table. A common problem here is that the `Host` value in the `user` table row specifies an unqualified host name, but your system's name resolution routines return a fully qualified domain name (or vice versa). For example, if you have an entry with host `'pluto'` in the `user` table, but your DNS tells MySQL that your host name is `'pluto.example.com'`, the entry does not work. Try adding an entry to the `user` table that contains the IP number of your host as the `Host` column value. (Alternatively, you could add an entry to the `user` table with a `Host` value that contains a wildcard; for example, `'pluto.%'`. However, use of `Host` values ending with "%" is *insecure* and is *not* recommended!)

- If `mysql -u user_name test` works but `mysql -u user_name other_db` does not, you have not granted access to the given user for the database named *`other_db`*.

- If `mysql -u user_name` works when executed on the server host, but `mysql -h host_name -u user_name` does not work when executed on a remote client host, you have not enabled access to the server for the given user name from the remote host.

- If you cannot figure out why you get `Access denied`, remove from the `user` table all entries that have `Host` values containing wildcards (entries that contain `'%'` or `'_'` characters). A very common error is to insert a new entry with `Host='%'` and `User='some_user'`, thinking that this allows you to specify `localhost` to connect from the same machine. The reason that this does not work is that the default privileges include an entry with `Host='localhost'` and `User=''`. Because that entry has a `Host` value `'localhost'` that is more specific than `'%'`, it is used in preference to the new entry when connecting from `localhost`! The correct procedure is to insert a second entry with `Host='localhost'` and `User='some_user'`, or to delete the entry with `Host='localhost'` and `User=''`. After deleting the entry, remember to issue a `FLUSH PRIVILEGES` statement to reload the grant tables. See also Section 3.4, "Access Control, Stage 1: Connection Verification".

- If you are able to connect to the MySQL server, but get an `Access denied` message whenever you issue a `SELECT ... INTO OUTFILE` or `LOAD DATA INFILE` statement, your account does not have the `FILE` privilege.

- If you change the grant tables directly (for example, by using `INSERT`, `UPDATE`, or `DELETE` statements) and your changes seem to be ignored, remember that you must execute a `FLUSH PRIVILEGES` statement or a `mysqladmin flush-privileges` command to cause the server to reload the privilege tables. Otherwise, your changes have no effect until the next time the server is restarted. Remember that after you change the `root` password with an `UPDATE` command, you will not need to specify the new password until after you flush the privileges, because the server will not know you've changed the password yet!

- If your privileges seem to have changed in the middle of a session, it may be that a MySQL administrator has changed them. Reloading the grant tables affects new client connections, but it also affects existing connections as indicated in Section 3.6, “When Privilege Changes Take Effect”.

- If you have access problems with a Perl, PHP, Python, or ODBC program, try to connect to the server with `mysql -u user_name db_name` or `mysql -u user_name -pyour_pass db_name`. If you are able to connect using the `mysql` client, the problem lies with your program, not with the access privileges. (There is no space between `-p` and the password; you can also use the `--password=your_pass` syntax to specify the password. If you use the `-p` or `--password` option with no password value, MySQL prompts you for the password.)

- For testing purposes, start the `mysqld` server with the `--skip-grant-tables` option. Then you can change the MySQL grant tables and use the `mysqlaccess` script to check whether your modifications have the desired effect. When you are satisfied with your changes, execute `mysqladmin flush-privileges` to tell the `mysqld` server to reload the privileges. This enables you to begin using the new grant table contents without stopping and restarting the server.

- If you get the following error, you may have a problem with the `db` or `host` table:

  ```
  Access to database denied
  ```

  If the entry selected from the `db` table has an empty value in the `Host` column, make sure that there are one or more corresponding entries in the `host` table specifying which hosts the `db` table entry applies to. This problem occurs infrequently because the `host` table is rarely used.

- If everything else fails, start the `mysqld` server with a debugging option (for example, `--debug=d,general,query`). This prints host and user information about attempted connections, as well as information about each command issued. See MySQL Internals: Porting.

- If you have any other problems with the MySQL grant tables and feel you must post the problem to the mailing list, always provide a dump of the MySQL grant tables. You can dump the tables with the `mysqldump mysql` command. To file a bug report, see the instructions at How to Report Bugs or Problems. In some cases, you may need to restart `mysqld` with `--skip-grant-tables` to run `mysqldump`.

Chapter 4. MySQL User Account Management

This section describes how to set up accounts for clients of your MySQL server. It discusses the following topics:

- The meaning of account names and passwords as used in MySQL and how that compares to names and passwords used by your operating system
- How to set up new accounts and remove existing accounts
- How to change passwords
- Guidelines for using passwords securely
- How to use secure connections with SSL

See also Account Management Statements, which describes the syntax and use for all user-management SQL statements.

4.1. User Names and Passwords

A MySQL account is defined in terms of a user name and the client host or hosts from which the user can connect to the server. The account also has a password. There are several distinctions between the way user names and passwords are used by MySQL and the way they are used by your operating system:

- User names, as used by MySQL for authentication purposes, have nothing to do with user names (login names) as used by Windows or Unix. On Unix, most MySQL clients by default try to log in using the current Unix user name as the MySQL user name, but that is for convenience only. The default can be overridden easily, because client programs allow any user name to be specified with a `-u` or `--user` option. Because this means that anyone can attempt to connect to the server using any user name, you cannot make a database secure in any way unless all MySQL accounts have passwords. Anyone who specifies a user name for an account that has no password is able to connect successfully to the server.
- MySQL user names can be up to 16 characters long. Operating system user names, because they are completely unrelated to MySQL user names, may be of a different maximum length. For example, Unix user names typically are limited to eight characters.

  > **Warning**
  >
  > The limit on MySQL user name length is hard-coded in the MySQL servers and clients, and trying to circumvent it by modifying the definitions of the tables in the `mysql` database *does not work*.
  >
  > You should never alter any of the tables in the `mysql` database in any manner whatsoever except by means of the procedure that is described in `mysql_upgrade`. Attempting to redefine MySQL's system tables in any other fashion results in undefined (and unsupported!) behavior.

- MySQL passwords have nothing to do with passwords for logging in to your operating system. There is no necessary connection between the password you use to log in to a Windows or Unix machine and the password you use to access the MySQL server on that machine.
- MySQL encrypts passwords using its own algorithm. This encryption is the same as that implemented by the `PASSWORD()` SQL function but differs from that used during the Unix login process. Unix password encryption is the same as that implemented by the `ENCRYPT()` SQL function. See the descriptions of the `PASSWORD()` and `ENCRYPT()` functions in Encryption and Compression Functions.

  From version 4.1 on, MySQL employs a stronger authentication method that has better password protection during the connection process than in earlier versions. It is secure even if TCP/IP packets are sniffed or the `mysql` database is captured. (In earlier versions, even though passwords are stored in encrypted form in the `user` table, knowledge of the encrypted password value could be used to connect to the MySQL server.) Section 4.6.3, “Password Hashing in MySQL”, discusses password encryption further.

When you install MySQL, the grant tables are populated with an initial set of accounts. These accounts have names and access privileges that are described in Section 2.3, “Securing the Initial MySQL Accounts”, which also discusses how to assign passwords to them. Thereafter, you normally set up, modify, and remove MySQL accounts using statements such as `GRANT` and `REVOKE`. See Account Management Statements.

When you connect to a MySQL server with a command-line client, you should specify the user name and password for the account that you want to use:

```
shell> mysql --user=monty --password=guess db_name
```

If you prefer short options, the command looks like this:

```
shell> mysql -u monty -pguess db_name
```

There must be *no space* between the `-p` option and the following password value. For additional information about specifying user names, passwords, and other connection parameters, see Connecting to the MySQL Server.

4.2. Adding User Accounts

You can create MySQL accounts in two ways:

- By using statements intended for creating accounts, such as `CREATE USER` or `GRANT`. These statements cause the server to make appropriate modifications to the grant tables.
- By manipulating the MySQL grant tables directly with statements such as `INSERT`, `UPDATE`, or `DELETE`.

The preferred method is to use account-creation statements because they are more concise and less error-prone than manipulating the grant tables directly. `CREATE USER` and `GRANT` are described in Account Management Statements.

Another option for creating accounts is to use one of several available third-party programs that offer capabilities for MySQL account administration. `phpMyAdmin` is one such program.

The following examples show how to use the `mysql` client program to set up new accounts. These examples assume that privileges have been set up according to the defaults described in Section 2.3, “Securing the Initial MySQL Accounts”. This means that to make changes, you must connect to the MySQL server as the MySQL `root` user, and the `root` account must have the `INSERT` privilege for the `mysql` database and the `RELOAD` administrative privilege.

As noted in the examples where appropriate, some of the statements will fail if the server's SQL mode has been set to enable certain restrictions. In particular, strict mode (`STRICT_TRANS_TABLES`, `STRICT_ALL_TABLES`) and `NO_AUTO_CREATE_USER` will prevent the server from accepting some of the statements. Workarounds are indicated for these cases. For more information about SQL modes and their effect on grant table manipulation, see Server SQL Modes, and `GRANT` Syntax.

First, use the `mysql` program to connect to the server as the MySQL `root` user:

```
shell> mysql --user=root mysql
```

If you have assigned a password to the `root` account, you'll also need to supply a `--password` or `-p` option, both for this `mysql` command and for those later in this section.

After connecting to the server as `root`, you can add new accounts. The following statements use `GRANT` to set up four new accounts:

```
mysql> CREATE USER 'monty'@'localhost' IDENTIFIED BY 'some_pass';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'monty'@'localhost'
    ->     WITH GRANT OPTION;
mysql> CREATE USER 'monty'@'%' IDENTIFIED BY 'some_pass';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'monty'@'%'
    ->     WITH GRANT OPTION;
mysql> CREATE USER 'admin'@'localhost';
mysql> GRANT RELOAD,PROCESS ON *.* TO 'admin'@'localhost';
mysql> CREATE USER 'dummy'@'localhost';
```

The accounts created by these statements have the following properties:

- Two of the accounts have a user name of `monty` and a password of `some_pass`. Both accounts are superuser accounts with full privileges to do anything. The `'monty'@'localhost'` account can be used only when connecting from the local host. The `'monty'@'%'` account uses the `'%'` wildcard for the host part, so it can be used to connect from any host.

  It is necessary to have both accounts for `monty` to be able to connect from anywhere as `monty`. Without the `localhost` account, the anonymous-user account for `localhost` that is created by `mysql_install_db` would take precedence when `monty` connects from the local host. As a result, `monty` would be treated as an anonymous user. The reason for this is that the anonymous-user account has a more specific `Host` column value than the `'monty'@'%'` account and thus comes earlier in the `user` table sort order. (`user` table sorting is discussed in Section 3.4, “Access Control, Stage 1: Connection Verification”.)

- The `'admin'@'localhost'` account has no password. This account can be used only by `admin` to connect from the local host. It is granted the `RELOAD` and `PROCESS` administrative privileges. These privileges allow the `admin` user to execute the

`mysqladmin reload`, `mysqladmin refresh`, and `mysqladmin flush-xxx` commands, as well as `mysqladmin processlist` . No privileges are granted for accessing any databases. You could add such privileges later by issuing additional `GRANT` statements.

- The `'dummy'@'localhost'` account has no password. This account can be used only to connect from the local host. No privileges are granted. It is assumed that you will grant specific privileges to the account later.

The statements that create accounts with no password will fail if the `NO_AUTO_CREATE_USER` SQL mode is enabled. To deal with this, use an `IDENTIFIED BY` clause that specifies a non-empty password.

To check the privileges for an account, use `SHOW GRANTS`:

```
mysql> SHOW GRANTS FOR 'admin'@'localhost';
+-----------------------------------------------------+
| Grants for admin@localhost                          |
+-----------------------------------------------------+
| GRANT RELOAD, PROCESS ON *.* TO 'admin'@'localhost' |
+-----------------------------------------------------+
```

As an alternative to `CREATE USER` and `GRANT`, you can create the same accounts directly by issuing `INSERT` statements and then telling the server to reload the grant tables using `FLUSH PRIVILEGES`:

```
shell> mysql --user=root mysql
mysql> INSERT INTO user
    ->     VALUES('localhost','monty',PASSWORD('some_pass'),
    ->     'Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y');
mysql> INSERT INTO user
    ->     VALUES('%','monty',PASSWORD('some_pass'),
    ->     'Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y',
    ->     'Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y','Y',
    ->     '','','','',0,0,0,0);
mysql> INSERT INTO user SET Host='localhost',User='admin',
    ->     Reload_priv='Y', Process_priv='Y';
mysql> INSERT INTO user (Host,User,Password)
    ->     VALUES('localhost','dummy','');
mysql> FLUSH PRIVILEGES;
```

When you create accounts with `INSERT`, it is necessary to use `FLUSH PRIVILEGES` to tell the server to reload the grant tables. Otherwise, the changes go unnoticed until you restart the server. With `CREATE USER`, `FLUSH PRIVILEGES` is unnecessary.

The reason for using the `PASSWORD()` function with `INSERT` is to encrypt the password. The `CREATE USER` statement encrypts the password for you, so `PASSWORD()` is unnecessary.

The `'Y'` values enable privileges for the accounts. Depending on your MySQL version, you may have to use a different number of `'Y'` values in the first two `INSERT` statements. The `INSERT` statement for the `admin` account employs the more readable extended `INSERT` syntax using `SET`.

In the `INSERT` statement for the `dummy` account, only the `Host`, `User`, and `Password` columns in the `user` table row are assigned values. None of the privilege columns are set explicitly, so MySQL assigns them all the default value of `'N'`. This is equivalent to what `CREATE USER` does.

If strict SQL mode is enabled, all columns that have no default value must have a value specified. In this case, `INSERT` statements must explicitly specify values for the `ssl_cipher`, `x509_issuer`, and `x509_subject` columns.

To set up a superuser account, it is necessary only to create a `user` table entry with the privilege columns set to `'Y'`. The `user` table privileges are global, so no entries in any of the other grant tables are needed.

The next examples create three accounts and give them access to specific databases. Each of them has a user name of `custom` and password of `obscure`.

To create the accounts with `CREATE USER` and `GRANT`, use the following statements:

```
shell> mysql --user=root mysql
mysql> CREATE USER 'custom'@'localhost' IDENTIFIED BY 'obscure';
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    ->     ON bankaccount.*
    ->     TO 'custom'@'localhost';
mysql> CREATE USER 'custom'@'host47.example.com' IDENTIFIED BY 'obscure';
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    ->     ON expenses.*
    ->     TO 'custom'@'host47.example.com';
mysql> CREATE USER 'custom'@'server.domain' IDENTIFIED BY 'obscure';
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    ->     ON customer.*
    ->     TO 'custom'@'server.domain';
```

The three accounts can be used as follows:

- The first account can access the `bankaccount` database, but only from the local host.
- The second account can access the `expenses` database, but only from the host `host47.example.com`.
- The third account can access the `customer` database, but only from the host `server.domain`.

To set up the `custom` accounts without `GRANT`, use `INSERT` statements as follows to modify the grant tables directly:

```
shell> mysql --user=root mysql
mysql> INSERT INTO user (Host,User,Password)
    ->     VALUES('localhost','custom',PASSWORD('obscure'));
mysql> INSERT INTO user (Host,User,Password)
    ->     VALUES('host47.example.com','custom',PASSWORD('obscure'));
mysql> INSERT INTO user (Host,User,Password)
    ->     VALUES('server.domain','custom',PASSWORD('obscure'));
mysql> INSERT INTO db
    ->     (Host,Db,User,Select_priv,Insert_priv,
    ->     Update_priv,Delete_priv,Create_priv,Drop_priv)
    ->     VALUES('localhost','bankaccount','custom',
    ->     'Y','Y','Y','Y','Y','Y');
mysql> INSERT INTO db
    ->     (Host,Db,User,Select_priv,Insert_priv,
    ->     Update_priv,Delete_priv,Create_priv,Drop_priv)
    ->     VALUES('host47.example.com','expenses','custom',
    ->     'Y','Y','Y','Y','Y','Y');
mysql> INSERT INTO db
    ->     (Host,Db,User,Select_priv,Insert_priv,
    ->     Update_priv,Delete_priv,Create_priv,Drop_priv)
    ->     VALUES('server.domain','customer','custom',
    ->     'Y','Y','Y','Y','Y','Y');
mysql> FLUSH PRIVILEGES;
```

The first three `INSERT` statements add `user` table entries that allow the user `custom` to connect from the various hosts with the given password, but grant no global privileges (all privileges are set to the default value of `'N'`). The next three `INSERT` statements add `db` table entries that grant privileges to `custom` for the `bankaccount`, `expenses`, and `customer` databases, but only when accessed from the proper hosts. As usual when you modify the grant tables directly, you must tell the server to reload them with `FLUSH PRIVILEGES` so that the privilege changes take effect.

To create a user who has access from all machines in a given domain (for example, `mydomain.com`), you can use the "`%`" wildcard character in the host part of the account name:

```
mysql> CREATE USER 'myname'@'%.mydomain.com' IDENTIFIED BY 'mypass';
```

To do the same thing by modifying the grant tables directly, do this:

```
mysql> INSERT INTO user (Host,User,Password,...)
    ->     VALUES('%.mydomain.com','myname',PASSWORD('mypass'),...);
mysql> FLUSH PRIVILEGES;
```

4.3. Removing User Accounts

To remove an account, use the `DROP USER` statement, which is described in `DROP USER` Syntax.

4.4. Limiting Account Resources

One means of limiting use of MySQL server resources is to set the `max_user_connections` system variable to a nonzero value. However, this limits only the number of simultaneous connections made using a single account, and not what a client can do once connected. In addition, this method is strictly global, and does not allow for management of individual accounts. Both types of control are of interest to many MySQL administrators, particularly those working for Internet Service Providers.

In MySQL 6.0, you can limit the following server resources for individual accounts:

- The number of queries that an account can issue per hour
- The number of updates that an account can issue per hour
- The number of times an account can connect to the server per hour
- The number of simultaneous connections to the server an account can have

Any statement that a client can issue counts against the query limit. Only statements that modify databases or tables count against the update limit.

An “account” in this context corresponds to a single row in the `user` table. That is, connections are assessed against the `User` and `Host` values in the `user` table row that applies to the connection. Suppose that there is a row in the `user` table that has `User` and `Host` values of `usera` and `%.example.com`, to allow `usera` to connect from any host in the `example.com` domain. In this case, the server applies resource limits collectively to all connections by `usera` from any host in the `example.com` domain because all such connections use the same account.

Before MySQL 5.0.3, an “account” was assessed against the actual host from which a user connects. This older method accounting may be selected by starting the server with the `--old-style-user-limits` option. In this case, if `usera` connects simultaneously from `host1.example.com` and `host2.example.com`, the server applies the account resource limits separately to each connection. If `usera` connects again from `host1.example.com`, the server applies the limits for that connection together with the existing connection from that host.

The server limits account resources based on the resource-related columns of the `user` table in the `mysql` database: `max_questions`, `max_updates`, `max_connections`, and `max_user_connections`. If your `user` table does not have these columns, it must be upgraded; see `mysql_upgrade`.

To set resource limits, use the `GRANT` statement and provide a `WITH` clause that names each resource to be limited. For example, to create a new account that can access the `customer` database, but only in a limited fashion, issue these statements:

```
mysql> CREATE USER 'francis'@'localhost' IDENTIFIED BY 'frank';
mysql> GRANT ALL ON customer.* TO 'francis'@'localhost'
    ->     WITH MAX_QUERIES_PER_HOUR 20
    ->          MAX_UPDATES_PER_HOUR 10
    ->          MAX_CONNECTIONS_PER_HOUR 5
    ->          MAX_USER_CONNECTIONS 2;
```

The limit types need not all be named in the `WITH` clause, but those named can be present in any order. The value for each per-hour limit should be an integer representing a count per hour. If the `GRANT` statement has no `WITH` clause, the limits are each set to the default value of zero (that is, no limit). For `MAX_USER_CONNECTIONS`, the limit is an integer representing the maximum number of simultaneous connections the account can make at any one time. If the limit is set to the default value of zero, the `max_user_connections` system variable determines the number of simultaneous connections for the account.

To modify limits for an existing account, use a `GRANT USAGE` statement at the global level (`ON *.*`). The following statement changes the query limit for `francis` to 100:

```
mysql> GRANT USAGE ON *.* TO 'francis'@'localhost'
    ->     WITH MAX_QUERIES_PER_HOUR 100;
```

This statement leaves the account's existing privileges unchanged and modifies only the limit values specified.

To remove an existing limit, set its value to zero. For example, to remove the limit on how many times per hour `francis` can connect, use this statement:

```
mysql> GRANT USAGE ON *.* TO 'francis'@'localhost'
    ->     WITH MAX_CONNECTIONS_PER_HOUR 0;
```

Resource-use counting takes place when any account has a nonzero limit placed on its use of any of the resources.

As the server runs, it counts the number of times each account uses resources. If an account reaches its limit on number of connections within the last hour, further connections for the account are rejected until that hour is up. Similarly, if the account reaches its limit on the number of queries or updates, further queries or updates are rejected until the hour is up. In all such cases, an appropriate error message is issued.

Resource counting is done per account, not per client. For example, if your account has a query limit of 50, you cannot increase your limit to 100 by making two simultaneous client connections to the server. Queries issued on both connections are counted together.

Queries for which results are served from the query cache do not count against the `MAX_QUERIES_PER_HOUR` limit.

The current per-hour resource-use counts can be reset globally for all accounts, or individually for a given account:

- To reset the current counts to zero for all accounts, issue a `FLUSH USER_RESOURCES` statement. The counts also can be reset by reloading the grant tables (for example, with a `FLUSH PRIVILEGES` statement or a `mysqladmin reload` command).
- The counts for an individual account can be set to zero by re-granting it any of its limits. To do this, use `GRANT USAGE` as described earlier and specify a limit value equal to the value that the account currently has.

Counter resets do not affect the `MAX_USER_CONNECTIONS` limit.

All counts begin at zero when the server starts; counts are not carried over through a restart.

For the MAX_USER_CONNECTIONS limit, an edge case can occur if the account currently has open the maximum number of connections allowed to it: A disconnect followed quickly by a connect can result in an error (ER_TOO_MANY_USER_CONNECTIONS or ER_USER_LIMIT_REACHED) if the server has not fully processed the disconnect by the time the connect occurs. When the server finishes disconnect processing, another connection will once more be allowed.

4.5. Assigning Account Passwords

To assign a password when you create a new account with CREATE USER, include an IDENTIFIED BY clause:

```
mysql> CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'biscuit';
```

To assign or change a password for an existing account, one way is to issue a SET PASSWORD statement:

```
mysql> SET PASSWORD FOR 'jeffrey'@'localhost' = PASSWORD('biscuit');
```

Only users such as root that have update access to the mysql database can change the password for other users. If you are not connected as an anonymous user, you can change your own password by omitting the FOR clause:

```
mysql> SET PASSWORD = PASSWORD('biscuit');
```

You can also use a GRANT USAGE statement at the global level (ON *.*) to assign a password to an account without affecting the account's current privileges:

```
mysql> GRANT USAGE ON *.* TO 'jeffrey'@'localhost' IDENTIFIED BY 'biscuit';
```

Passwords can be assigned from the command line by using the mysqladmin command:

```
shell> mysqladmin -u user_name -h host_name password "newpwd"
```

The account for which this command resets the password is the one with a user table row that matches *user_name* in the User column and the client host *from which you connect* in the Host column.

Although it is generally preferable to assign passwords using one of the preceding methods, you can also do so by modifying the user table directly:

- To establish a password when creating a new account, provide a value for the Password column:

  ```
  shell> mysql -u root mysql
  mysql> INSERT INTO user (Host,User,Password)
      -> VALUES('localhost','jeffrey',PASSWORD('biscuit'));
  mysql> FLUSH PRIVILEGES;
  ```

- To change the password for an existing account, use UPDATE to set the Password column value:

  ```
  shell> mysql -u root mysql
  mysql> UPDATE user SET Password = PASSWORD('bagel')
      -> WHERE Host = 'localhost' AND User = 'francis';
  mysql> FLUSH PRIVILEGES;
  ```

When you assign passwords using CREATE USER or GRANT with an IDENTIFIED BY clause or with the mysqladmin password command, they take care of encrypting the password for you.

When you assign an account a non-empty password using SET PASSWORD, INSERT, or UPDATE, you must use the PASSWORD() function to encrypt the password. PASSWORD() is necessary because the user table stores passwords in encrypted form, not as plaintext. If you forget that fact, you are likely to set passwords like this:

```
shell> mysql -u root mysql
mysql> INSERT INTO user (Host,User,Password)
    -> VALUES('localhost','jeffrey','biscuit');
mysql> FLUSH PRIVILEGES;
```

The result is that the literal value 'biscuit' is stored as the password in the user table, not the encrypted value. When jeffrey attempts to connect to the server using this password, the value is encrypted and compared to the value stored in the user table. However, the stored value is the literal string 'biscuit', so the comparison fails and the server rejects the connection:

```
shell> mysql -u jeffrey -pbiscuit test
```

```
Access denied
```

> **Note**
>
> `PASSWORD()` encryption differs from Unix password encryption. See Section 4.1, “User Names and Passwords”.

4.6. Password Security in MySQL

Passwords occur in several contexts within MySQL. The following sections provide guidelines that enable administrators and end users to keep these passwords secure and avoid exposing them. There is also a discussion of how MySQL uses password hashing internally.

4.6.1. Administrator Guidelines for Password Security

Database administrators should use the following guidelines to keep passwords secure.

MySQL stores passwords for user accounts in the `mysql.user` table. Access to this table should never be granted to any non-administrative accounts.

Passwords can appear as plain text in SQL statements such as `CREATE USER`, `GRANT`, and `SET PASSWORD`. If these statements are logged by the MySQL server, the passwords become available to anyone with access to the logs. This applies to the general query log, the slow query log, and the binary log (see MySQL Server Logs). To guard against unwarranted exposure to log files, they should be located in a directory that restricts access to only the server and the database administrator. If you log to tables in the `mysql` database, access to the tables should never be granted to any non-administrative accounts.

Database backups that include tables or log files containing passwords should be protected using a restricted access mode.

4.6.2. End-User Guidelines for Password Security

MySQL users should use the following guidelines to keep passwords secure.

When you run a client program to connect to the MySQL server, it is inadvisable to specify your password in a way that exposes it to discovery by other users. The methods you can use to specify your password when you run client programs are listed here, along with an assessment of the risks of each method. In short, the safest methods are to have the client program prompt for the password or to specify the password in a properly protected option file.

- Use a `-p`*`your_pass`* or `--password=`*`your_pass`* option on the command line. For example:

  ```
  shell> mysql -u francis -pfrank db_name
  ```

  This is convenient *but insecure*, because your password becomes visible to system status programs such as `ps` that may be invoked by other users to display command lines. MySQL clients typically overwrite the command-line password argument with zeros during their initialization sequence. However, there is still a brief interval during which the value is visible. Also, on some systems this overwriting strategy is ineffective and the password remains visible to `ps`. (SystemV Unix systems and perhaps others are subject to this problem.)

  If your operating environment is set up to display your current command in the title bar of your terminal window, the password remains visible as long as the command is running, even if the command has scrolled out of view in the window content area.

- Use the `-p` or `--password` option on the command line with no password value specified. In this case, the client program solicits the password interactively:

  ```
  shell> mysql -u francis -p db_name
  Enter password: ********
  ```

  The “`*`” characters indicate where you enter your password. The password is not displayed as you enter it.

  It is more secure to enter your password this way than to specify it on the command line because it is not visible to other users. However, this method of entering a password is suitable only for programs that you run interactively. If you want to invoke a client from a script that runs non-interactively, there is no opportunity to enter the password from the keyboard. On some systems, you may even find that the first line of your script is read and interpreted (incorrectly) as your password.

- Store your password in an option file. For example, on Unix you can list your password in the `[client]` section of the `.my.cnf` file in your home directory:

  ```
  [client]
  password=your_pass
  ```

To keep the password safe, the file should not be accessible to anyone but yourself. To ensure this, set the file access mode to `400` or `600`. For example:

```
shell> chmod 600 .my.cnf
```

Using Option Files, discusses option files in more detail.

- Store your password in the `MYSQL_PWD` environment variable. See Environment Variables.

  This method of specifying your MySQL password must be considered *extremely insecure* and should not be used. Some versions of `ps` include an option to display the environment of running processes. If you set `MYSQL_PWD`, your password is exposed to any other user who runs `ps`. Even on systems without such a version of `ps`, it is unwise to assume that there are no other methods by which users can examine process environments.

On Unix, the `mysql` client writes a record of executed statements to a history file (see `mysql`). By default, this file is named `.mysql_history` and is created in your home directory. Passwords can appear as plain text in SQL statements such as `CREATE USER`, `GRANT`, and `SET PASSWORD`, so if you use these statements, they are logged in the history file. To keep this file safe, use a restrictive access mode, the same way as described earlier for the `.my.cnf` file.

4.6.3. Password Hashing in MySQL

MySQL user accounts are listed in the `user` table of the `mysql` database. Each MySQL account is assigned a password, although what is stored in the `Password` column of the `user` table is not the plaintext version of the password, but a hash value computed from it. Password hash values are computed by the `PASSWORD()` function.

MySQL uses passwords in two phases of client/server communication:

- When a client attempts to connect to the server, there is an initial authentication step in which the client must present a password that has a hash value matching the hash value stored in the `user` table for the account that the client wants to use.
- After the client connects, it can (if it has sufficient privileges) set or change the password hashes for accounts listed in the `user` table. The client can do this by using the `PASSWORD()` function to generate a password hash, or by using the `GRANT` or `SET PASSWORD` statements.

In other words, the server *uses* hash values during authentication when a client first attempts to connect. The server *generates* hash values if a connected client invokes the `PASSWORD()` function or uses a `GRANT` or `SET PASSWORD` statement to set or change a password.

The password hashing mechanism was updated in MySQL 4.1 to provide better security and to reduce the risk of passwords being intercepted. However, this new mechanism is understood only by MySQL 4.1 (and newer) servers and clients, which can result in some compatibility problems. A 4.1 or newer client can connect to a pre-4.1 server, because the client understands both the old and new password hashing mechanisms. However, a pre-4.1 client that attempts to connect to a 4.1 or newer server may run into difficulties. For example, a 3.23 `mysql` client that attempts to connect to a 6.0 server may fail with the following error message:

```
shell> mysql -h localhost -u root
Client does not support authentication protocol requested
by server; consider upgrading MySQL client
```

Another common example of this phenomenon occurs for attempts to use the older PHP `mysql` extension after upgrading to MySQL 4.1 or newer. (See Common Problems with MySQL and PHP.)

The following discussion describes the differences between the old and new password mechanisms, and what you should do if you upgrade your server but need to maintain backward compatibility with pre-4.1 clients. Additional information can be found in `Client does not support authentication protocol`. This information is of particular importance to PHP programmers migrating MySQL databases from version 4.0 or lower to version 4.1 or higher.

> **Note**
>
> This discussion contrasts 4.1 behavior with pre-4.1 behavior, but the 4.1 behavior described here actually begins with 4.1.1. MySQL 4.1.0 is an “odd” release because it has a slightly different mechanism than that implemented in 4.1.1 and up. Differences between 4.1.0 and more recent versions are described further in MySQL 5.1 Reference Manual.

Prior to MySQL 4.1, password hashes computed by the `PASSWORD()` function are 16 bytes long. Such hashes look like this:

```
mysql> SELECT PASSWORD('mypass');
+--------------------+
| PASSWORD('mypass') |
```

```
+--------------------+
| 6f8c114b58f2ce9e   |
+--------------------+
```

The `Password` column of the `user` table (in which these hashes are stored) also is 16 bytes long before MySQL 4.1.

As of MySQL 4.1, the `PASSWORD()` function has been modified to produce a longer 41-byte hash value:

```
mysql> SELECT PASSWORD('mypass');
+-------------------------------------------+
| PASSWORD('mypass')                        |
+-------------------------------------------+
| *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4 |
+-------------------------------------------+
```

Accordingly, the `Password` column in the `user` table also must be 41 bytes long to store these values:

- If you perform a new installation of MySQL 6.0, the `Password` column is made 41 bytes long automatically.
- Upgrading from MySQL 4.1 (4.1.1 or later in the 4.1 series) to MySQL 6.0 should not give rise to any issues in this regard because both versions use the same password hashing mechanism. If you wish to upgrade an older release of MySQL to version 6.0, you should upgrade to version 4.1 first, then upgrade the 4.1 installation to 6.0.

A widened `Password` column can store password hashes in both the old and new formats. The format of any given password hash value can be determined two ways:

- The obvious difference is the length (16 bytes versus 41 bytes).
- A second difference is that password hashes in the new format always begin with a "`*`" character, whereas passwords in the old format never do.

The longer password hash format has better cryptographic properties, and client authentication based on long hashes is more secure than that based on the older short hashes.

The differences between short and long password hashes are relevant both for how the server uses passwords during authentication and for how it generates password hashes for connected clients that perform password-changing operations.

The way in which the server uses password hashes during authentication is affected by the width of the `Password` column:

- If the column is short, only short-hash authentication is used.
- If the column is long, it can hold either short or long hashes, and the server can use either format:
  - Pre-4.1 clients can connect, although because they know only about the old hashing mechanism, they can authenticate only using accounts that have short hashes.
  - 4.1 and later clients can authenticate using accounts that have short or long hashes.

Even for short-hash accounts, the authentication process is actually a bit more secure for 4.1 and later clients than for older clients. In terms of security, the gradient from least to most secure is:

- Pre-4.1 client authenticating with short password hash
- 4.1 or later client authenticating with short password hash
- 4.1 or later client authenticating with long password hash

The way in which the server generates password hashes for connected clients is affected by the width of the `Password` column and by the `--old-passwords` option. A 4.1 or later server generates long hashes only if certain conditions are met: The `Password` column must be wide enough to hold long values and the `--old-passwords` option must not be given. These conditions apply as follows:

- The `Password` column must be wide enough to hold long hashes (41 bytes). If the column has not been updated and still has the pre-4.1 width of 16 bytes, the server notices that long hashes cannot fit into it and generates only short hashes when a client performs password-changing operations using `PASSWORD()`, `GRANT`, or `SET PASSWORD`. This is the behavior that occurs if

you have upgraded to 4.1 but have not yet run the `mysql_upgrade` program to widen the `Password` column.

- If the `Password` column is wide, it can store either short or long password hashes. In this case, `PASSWORD()`, `GRANT`, and `SET PASSWORD` generate long hashes unless the server was started with the `--old-passwords` option. That option forces the server to generate short password hashes instead.

The purpose of the `--old-passwords` option is to enable you to maintain backward compatibility with pre-4.1 clients under circumstances where the server would otherwise generate long password hashes. The option doesn't affect authentication (4.1 and later clients can still use accounts that have long password hashes), but it does prevent creation of a long password hash in the `user` table as the result of a password-changing operation. Were that to occur, the account no longer could be used by pre-4.1 clients. Without the `--old-passwords` option, the following undesirable scenario is possible:

- An old client connects to an account that has a short password hash.
- The client changes its own password. Without `--old-passwords`, this results in the account having a long password hash.
- The next time the old client attempts to connect to the account, it cannot, because the account has a long password hash that requires the new hashing mechanism during authentication. (Once an account has a long password hash in the user table, only 4.1 and later clients can authenticate for it, because pre-4.1 clients do not understand long hashes.)

This scenario illustrates that, if you must support older pre-4.1 clients, it is dangerous to run a 4.1 or newer server without using the `--old-passwords` option. By running the server with `--old-passwords`, password-changing operations do not generate long password hashes and thus do not cause accounts to become inaccessible to older clients. (Those clients cannot inadvertently lock themselves out by changing their password and ending up with a long password hash.)

The downside of the `--old-passwords` option is that any passwords you create or change use short hashes, even for 4.1 clients. Thus, you lose the additional security provided by long password hashes. If you want to create an account that has a long hash (for example, for use by 4.1 clients), you must do so while running the server without `--old-passwords`.

> **MySQL Enterprise**
> Subscribers to the MySQL Enterprise Monitor are automatically alerted whenever a server is running with the `--old-passwords` option. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

The following scenarios are possible for running a 4.1 or later server:

Scenario 1: Short `Password` column in user table:

- Only short hashes can be stored in the `Password` column.
- The server uses only short hashes during client authentication.
- For connected clients, password hash-generating operations involving `PASSWORD()`, `GRANT`, or `SET PASSWORD` use short hashes exclusively. Any change to an account's password results in that account having a short password hash.
- The `--old-passwords` option can be used but is superfluous because with a short `Password` column, the server generates only short password hashes anyway.

Scenario 2: Long `Password` column; server not started with `--old-passwords` option:

- Short or long hashes can be stored in the `Password` column.
- 4.1 and later clients can authenticate using accounts that have short or long hashes.
- Pre-4.1 clients can authenticate only using accounts that have short hashes.
- For connected clients, password hash-generating operations involving `PASSWORD()`, `GRANT`, or `SET PASSWORD` use long hashes exclusively. A change to an account's password results in that account having a long password hash.

As indicated earlier, a danger in this scenario is that it is possible for accounts that have a short password hash to become inaccessible to pre-4.1 clients. A change to such an account's password made via `GRANT`, `PASSWORD()`, or `SET PASSWORD` results in the account being given a long password hash. From that point on, no pre-4.1 client can authenticate to that account until the client upgrades to 4.1.

To deal with this problem, you can change a password in a special way. For example, normally you use SET PASSWORD as follows to change an account password:

```
SET PASSWORD FOR 'some_user'@'some_host' = PASSWORD('mypass');
```

To change the password but create a short hash, use the OLD_PASSWORD() function instead:

```
SET PASSWORD FOR 'some_user'@'some_host' = OLD_PASSWORD('mypass');
```

OLD_PASSWORD() is useful for situations in which you explicitly want to generate a short hash.

Scenario 3: Long Password column; 4.1 or newer server started with --old-passwords option:

- Short or long hashes can be stored in the Password column.
- 4.1 and later clients can authenticate for accounts that have short or long hashes (but note that it is possible to create long hashes only when the server is started without --old-passwords).
- Pre-4.1 clients can authenticate only for accounts that have short hashes.
- For connected clients, password hash-generating operations involving PASSWORD(), GRANT, or SET PASSWORD use short hashes exclusively. Any change to an account's password results in that account having a short password hash.

In this scenario, you cannot create accounts that have long password hashes, because the --old-passwords option prevents generation of long hashes. Also, if you create an account with a long hash before using the --old-passwords option, changing the account's password while --old-passwords is in effect results in the account being given a short password, causing it to lose the security benefits of a longer hash.

The disadvantages for these scenarios may be summarized as follows:

In scenario 1, you cannot take advantage of longer hashes that provide more secure authentication.

In scenario 2, accounts with short hashes become inaccessible to pre-4.1 clients if you change their passwords without explicitly using OLD_PASSWORD().

In scenario 3, --old-passwords prevents accounts with short hashes from becoming inaccessible, but password-changing operations cause accounts with long hashes to revert to short hashes, and you cannot change them back to long hashes while --old-passwords is in effect.

4.6.4. Implications of Password Hashing Changes in MySQL 4.1 for Application Programs

An upgrade to MySQL version 4.1 or later can cause compatibility issues for applications that use PASSWORD() to generate passwords for their own purposes. Applications really should not do this, because PASSWORD() should be used only to manage passwords for MySQL accounts. But some applications use PASSWORD() for their own purposes anyway.

If you upgrade to 4.1 or later from a pre-4.1 version of MySQL and run the server under conditions where it generates long password hashes, an application using PASSWORD() for its own passwords breaks. The recommended course of action in such cases is to modify the application to use another function, such as SHA1() or MD5(), to produce hashed values. If that is not possible, you can use the OLD_PASSWORD() function, which is provided for generate short hashes in the old format. However, you should note that OLD_PASSWORD() may one day no longer be supported.

If the server is running under circumstances where it generates short hashes, OLD_PASSWORD() is available but is equivalent to PASSWORD().

PHP programmers migrating their MySQL databases from version 4.0 or lower to version 4.1 or higher should see MySQL PHP API.

4.7. Using SSL for Secure Connections

MySQL supports secure (encrypted) connections between MySQL clients and the server using the Secure Sockets Layer (SSL) protocol. This section discusses how to use SSL connections. For information on how to require users to use SSL connections, see the discussion of the REQUIRE clause of the GRANT statement in GRANT Syntax.

The standard configuration of MySQL is intended to be as fast as possible, so encrypted connections are not used by default. Doing so would make the client/server protocol much slower. Encrypting data is a CPU-intensive operation that requires the computer to

do additional work and can delay other MySQL tasks. For applications that require the security provided by encrypted connections, the extra computation is warranted.

MySQL allows encryption to be enabled on a per-connection basis. You can choose a normal unencrypted connection or a secure encrypted SSL connection according the requirements of individual applications.

Secure connections are based on the OpenSSL API and are available through the MySQL C API. Replication uses the C API, so secure connections can be used between master and slave servers.

Another way to connect securely is from within an SSH connection to the MySQL server host. For an example, see Section 4.8, “Connecting to MySQL Remotely from Windows with SSH”.

4.7.1. Basic SSL Concepts

To understand how MySQL uses SSL, it is necessary to explain some basic SSL and X509 concepts. People who are familiar with these can skip this part of the discussion.

By default, MySQL uses unencrypted connections between the client and the server. This means that someone with access to the network could watch all your traffic and look at the data being sent or received. They could even change the data while it is in transit between client and server. To improve security a little, you can compress client/server traffic by using the `--compress` option when invoking client programs. However, this does not foil a determined attacker.

When you need to move information over a network in a secure fashion, an unencrypted connection is unacceptable. Encryption is the way to make any kind of data unreadable. In fact, today's practice requires many additional security elements from encryption algorithms. They should resist many kind of known attacks such as changing the order of encrypted messages or replaying data twice.

SSL is a protocol that uses different encryption algorithms to ensure that data received over a public network can be trusted. It has mechanisms to detect any data change, loss, or replay. SSL also incorporates algorithms that provide identity verification using the X509 standard.

X509 makes it possible to identify someone on the Internet. It is most commonly used in e-commerce applications. In basic terms, there should be some company called a “Certificate Authority” (or CA) that assigns electronic certificates to anyone who needs them. Certificates rely on asymmetric encryption algorithms that have two encryption keys (a public key and a secret key). A certificate owner can show the certificate to another party as proof of identity. A certificate consists of its owner's public key. Any data encrypted with this public key can be decrypted only using the corresponding secret key, which is held by the owner of the certificate.

If you need more information about SSL, X509, or encryption, use your favorite Internet search engine to search for the keywords in which you are interested.

4.7.2. Using SSL Connections

To use SSL connections between the MySQL server and client programs, your system must support either OpenSSL or yaSSL and your version of MySQL must be built with SSL support.

To make it easier to use secure connections, MySQL is bundled with yaSSL. (MySQL and yaSSL employ the same licensing model, whereas OpenSSL uses an Apache-style license.) yaSSL support initially was available only for a few platforms, but now it is available on all MySQL platforms supported by Sun Microsystems, Inc.

To get secure connections to work with MySQL and SSL, you must do the following:

1. If you are not using a binary (precompiled) version of MySQL that has been built with SSL support, and you are going to use OpenSSL rather than the bundled yaSSL library, install OpenSSL if it has not already been installed. We have tested MySQL with OpenSSL 0.9.6. To obtain OpenSSL, visit http://www.openssl.org.

2. If you are not using a binary (precompiled) version of MySQL that has been built with SSL support, configure a MySQL source distribution to use SSL. When you configure MySQL, invoke the `configure` script like this:

   ```
   shell> ./configure --with-ssl
   ```

   That configures the distribution to use the bundled yaSSL library. To use OpenSSL instead, specify the `--with-ssl` option with the path to the directory where the OpenSSL header files and libraries are located:

   ```
   shell> ./configure --with-ssl=path
   ```

   Note that yaSSL support on Unix platforms requires that either `/dev/urandom` or `/dev/random` be available to retrieve true random numbers. For additional information (especially regarding yaSSL on Solaris versions prior to 2.8 and HP-UX),

see Bug#13164.

3. Make sure that the `user` in the `mysql` database includes the SSL-related columns (beginning with `ssl_` and `x509_`). If your `user` table does not have these columns, it must be upgraded; see `mysql_upgrade`.

4. To check whether a server binary is compiled with SSL support, invoke it with the `--ssl` option. An error will occur if the server does not support SSL:

```
shell> mysqld --ssl --help
060525 14:18:52 [ERROR] mysqld: unknown option '--ssl'
```

   To check whether a running `mysqld` server supports SSL, examine the value of the `have_ssl` system variable:

```
mysql> SHOW VARIABLES LIKE 'have_ssl';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| have_ssl      | YES   |
+---------------+-------+
```

   If the value is `YES`, the server supports SSL connections. If the value is `DISABLED`, the server supports SSL connections but was not started with the appropriate `--ssl-xxx` options (described later in this section).

To enable SSL connections, the proper SSL-related options must be used (see Section 4.7.3, “SSL Command Options”).

To start the MySQL server so that it allows clients to connect via SSL, use the options that identify the key and certificate files the server needs when establishing a secure connection:

```
shell> mysqld --ssl-ca=cacert.pem \
       --ssl-cert=server-cert.pem \
       --ssl-key=server-key.pem
```

- `--ssl-ca` identifies the Certificate Authority (CA) certificate.
- `--ssl-cert` identifies the server public key. This can be sent to the client and authenticated against the CA certificate that it has.
- `--ssl-key` identifies the server private key.

To establish a secure connection to a MySQL server with SSL support, the options that a client must specify depend on the SSL requirements of the user account that the client uses. (See the discussion of the `REQUIRE` clause in `GRANT` Syntax.)

If the account has no special SSL requirements or was created using a `GRANT` statement that includes the `REQUIRE SSL` option, a client can connect securely by using just the `--ssl-ca` option:

```
shell> mysql --ssl-ca=cacert.pem
```

To require that a client certificate also be specified, create the account using the `REQUIRE X509` option. Then the client must also specify the proper client key and certificate files or the server will reject the connection:

```
shell> mysql --ssl-ca=cacert.pem \
       --ssl-cert=client-cert.pem \
       --ssl-key=client-key.pem
```

In other words, the options are similar to those used for the server. Note that the Certificate Authority certificate has to be the same.

A client can determine whether the current connection with the server uses SSL by checking the value of the `Ssl_cipher` status variable. The value of `Ssl_cipher` is non-empty if SSL is used, and empty otherwise. For example:

```
mysql> SHOW STATUS LIKE 'Ssl_cipher';
+---------------+--------------------+
| Variable_name | Value              |
+---------------+--------------------+
| Ssl_cipher    | DHE-RSA-AES256-SHA |
+---------------+--------------------+
```

For the `mysql` client, you can use the `STATUS` or `\s` command and check the `SSL` line:

```
mysql> \s
...
SSL:                    Not in use
```

```
...
```

Or:

```
mysql> \s
...
SSL:                    Cipher in use is DHE-RSA-AES256-SHA
...
```

To establish a secure connection from within an application program, use the `mysql_ssl_set()` C API function to set the appropriate certificate options before calling `mysql_real_connect()`. See `mysql_ssl_set()`. After the connection is established, you can use `mysql_get_ssl_cipher()` to determine whether SSL is in use. A non-`NULL` return value indicates a secure connection and names the SSL cipher used for encryption. A `NULL` return value indicates that SSL is not being used. See `mysql_get_ssl_cipher()`.

4.7.3. SSL Command Options

The following list describes options that are used for specifying the use of SSL, certificate files, and key files. They can be given on the command line or in an option file. These options are not available unless MySQL has been built with SSL support. See Section 4.7.2, “Using SSL Connections”.

Table 4.1. `mysqld` SSL Option/Variable Summary

Name	Cmd-Line	Option file	System Var	Status Var	Var Scope	Dynamic
have_openssl			Yes		Global	No
have_ssl			Yes		Global	No
skip-ssl	Yes	Yes				
ssl	Yes	Yes				
ssl-ca	Yes	Yes			Global	No
- *Variable*: ssl_ca			Yes		Global	No
ssl-capath	Yes	Yes			Global	No
- *Variable*: ssl_capath			Yes		Global	No
ssl-cert	Yes	Yes			Global	No
- *Variable*: ssl_cert			Yes		Global	No
ssl-cipher	Yes	Yes			Global	No
- *Variable*: ssl_cipher			Yes		Global	No
ssl-key	Yes	Yes			Global	No
- *Variable*: ssl_key			Yes		Global	No
ssl-verify-server-cert	Yes	Yes				

- `--ssl`

  For the server, this option specifies that the server allows SSL connections. For a client program, it allows the client to connect to the server using SSL. This option is not sufficient in itself to cause an SSL connection to be used. You must also specify the `--ssl-ca` option, and possibly the `--ssl-cert` and `--ssl-key` options.

  This option is more often used in its opposite form to override any other SSL options and indicate that SSL should *not* be used. To do this, specify the option as `--skip-ssl` or `--ssl=0`.

  Note that use of `--ssl` does not *require* an SSL connection. For example, if the server or client is compiled without SSL support, a normal unencrypted connection is used.

  The secure way to require use of an SSL connection is to create an account on the server that includes a `REQUIRE SSL` clause in the `GRANT` statement. Then use that account to connect to the server, where both the server and the client have SSL support enabled.

  The `REQUIRE` clause allows other SSL-related restrictions as well. The description of `REQUIRE` in `GRANT` Syntax, provides additional detail about which SSL command options may or must be specified by clients that connect using accounts that are created using the various `REQUIRE` options.

- `--ssl-ca=file_name`

  The path to a file that contains a list of trusted SSL CAs.

- `--ssl-capath=directory_name`

  The path to a directory that contains trusted SSL CA certificates in PEM format.

- `--ssl-cert=file_name`

  The name of the SSL certificate file to use for establishing a secure connection.

- `--ssl-cipher=cipher_list`

  A list of allowable ciphers to use for SSL encryption. For greatest portability, *`cipher_list`* should be a list of one or more cipher names, separated by colons. Examples:

  ```
  --ssl-cipher=AES128-SHA
  --ssl-cipher=DHE-RSA-AES256-SHA:AES128-SHA
  ```

  This format is understood both by OpenSSL and yaSSL. OpenSSL supports a more flexible syntax for specifying ciphers, as described in the OpenSSL documentation at http://www.openssl.org/docs/apps/ciphers.html. However, this extended syntax will fail if used with a MySQL installation compiled against yaSSL.

  If no cipher in the list is supported, SSL connections will not work.

- `--ssl-key=file_name`

  The name of the SSL key file to use for establishing a secure connection.

- `--ssl-verify-server-cert`

  This option is available for client programs only, not the server. It causes the server's Common Name value in the certificate that the server sends to the client to be verified against the host name that the client uses for connecting to the server, and the connection is rejected if there is a mismatch. This feature can be used to prevent man-in-the-middle attacks. Verification is disabled by default.

If you use SSL when establishing a client connection, you can tell the client not to authenticate the server certificate by specifying neither `--ssl-ca` nor `--ssl-capath`. The server still verifies the client according to any applicable requirements established via `GRANT` statements for the client, and it still uses any `--ssl-ca`/`--ssl-capath` values that were passed to server at startup time.

4.7.4. Setting Up SSL Certificates for MySQL

This section demonstrates how to set up SSL certificate and key files for use by MySQL servers and clients. The first example shows a simplified procedure such as you might use from the command line. The second shows a script that contains more detail. The first two examples are intended for use on Unix and both use the `openssl` command that is part of OpenSSL. The third example describes how to set up SSL files on Windows.

Following the third example, instructions are given for using the files to test SSL connections. You can also use the files as described in Section 4.7.2, “Using SSL Connections”.

Example 1: Creating SSL files from the command line on Unix

The following example shows a set of commands to create MySQL server and client certificate and key files. You will need to respond to several prompts by the `openssl` commands. For testing, you can press Enter to all prompts. For production use, you should provide non-empty responses.

```
# Create clean environment
shell> rm -rf newcerts
shell> mkdir newcerts && cd newcerts
# Create CA certificate
shell> openssl genrsa 2048 > ca-key.pem
shell> openssl req -new -x509 -nodes -days 1000 \
         -key ca-key.pem > ca-cert.pem
# Create server certificate
shell> openssl req -newkey rsa:2048 -days 1000 \
         -nodes -keyout server-key.pem > server-req.pem
shell> openssl x509 -req -in server-req.pem -days 1000 \
         -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 > server-cert.pem
# Create client certificate
shell> openssl req -newkey rsa:2048 -days 1000 \
         -nodes -keyout client-key.pem > client-req.pem
shell> openssl x509 -req -in client-req.pem -days 1000 \
```

```
          -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 > client-cert.pem
```

Example 2: Creating SSL files using a script on Unix

Here is an example script that shows how to set up SSL certificates for MySQL:

```
DIR=`pwd`/openssl
PRIV=$DIR/private
mkdir $DIR $PRIV $DIR/newcerts
cp /usr/share/ssl/openssl.cnf $DIR
replace ./demoCA $DIR -- $DIR/openssl.cnf
# Create necessary files: $database, $serial and $new_certs_dir
# directory (optional)
touch $DIR/index.txt
echo "01" > $DIR/serial
#
# Generation of Certificate Authority(CA)
#
openssl req -new -x509 -keyout $PRIV/cakey.pem -out $DIR/cacert.pem \
    -days 3600 -config $DIR/openssl.cnf
# Sample output:
# Using configuration from /home/monty/openssl/openssl.cnf
# Generating a 1024 bit RSA private key
# ................++++++
# .........++++++
# writing new private key to '/home/monty/openssl/private/cakey.pem'
# Enter PEM pass phrase:
# Verifying password - Enter PEM pass phrase:
# -----
# You are about to be asked to enter information that will be
# incorporated into your certificate request.
# What you are about to enter is what is called a Distinguished Name
# or a DN.
# There are quite a few fields but you can leave some blank
# For some fields there will be a default value,
# If you enter '.', the field will be left blank.
# -----
# Country Name (2 letter code) [AU]:FI
# State or Province Name (full name) [Some-State]:.
# Locality Name (eg, city) []:
# Organization Name (eg, company) [Internet Widgits Pty Ltd]:MySQL AB
# Organizational Unit Name (eg, section) []:
# Common Name (eg, YOUR name) []:MySQL admin
# Email Address []:
#
# Create server request and key
#
openssl req -new -keyout $DIR/server-key.pem -out \
   $DIR/server-req.pem -days 3600 -config $DIR/openssl.cnf
# Sample output:
# Using configuration from /home/monty/openssl/openssl.cnf
# Generating a 1024 bit RSA private key
# ..++++++
# ..........++++++
# writing new private key to '/home/monty/openssl/server-key.pem'
# Enter PEM pass phrase:
# Verifying password - Enter PEM pass phrase:
# -----
# You are about to be asked to enter information that will be
# incorporated into your certificate request.
# What you are about to enter is what is called a Distinguished Name
# or a DN.
# There are quite a few fields but you can leave some blank
# For some fields there will be a default value,
# If you enter '.', the field will be left blank.
# -----
# Country Name (2 letter code) [AU]:FI
# State or Province Name (full name) [Some-State]:.
# Locality Name (eg, city) []:
# Organization Name (eg, company) [Internet Widgits Pty Ltd]:MySQL AB
# Organizational Unit Name (eg, section) []:
# Common Name (eg, YOUR name) []:MySQL server
# Email Address []:
#
# Please enter the following 'extra' attributes
# to be sent with your certificate request
# A challenge password []:
# An optional company name []:
#
# Remove the passphrase from the key
#
openssl rsa -in $DIR/server-key.pem -out $DIR/server-key.pem
#
# Sign server cert
#
openssl ca  -policy policy_anything -out $DIR/server-cert.pem \
    -config $DIR/openssl.cnf -infiles $DIR/server-req.pem
# Sample output:
# Using configuration from /home/monty/openssl/openssl.cnf
# Enter PEM pass phrase:
# Check that the request matches the signature
# Signature ok
# The Subjects Distinguished Name is as follows
# countryName           :PRINTABLE:'FI'
# organizationName      :PRINTABLE:'MySQL AB'
```

```
# commonName            :PRINTABLE:'MySQL admin'
# Certificate is to be certified until Sep 13 14:22:46 2003 GMT
# (365 days)
# Sign the certificate? [y/n]:y
#
#
# 1 out of 1 certificate requests certified, commit? [y/n]y
# Write out database with 1 new entries
# Data Base Updated
#
# Create client request and key
#
openssl req -new -keyout $DIR/client-key.pem -out \
    $DIR/client-req.pem -days 3600 -config $DIR/openssl.cnf
# Sample output:
# Using configuration from /home/monty/openssl/openssl.cnf
# Generating a 1024 bit RSA private key
# ....................................++++++
# ..........................................++++++
# writing new private key to '/home/monty/openssl/client-key.pem'
# Enter PEM pass phrase:
# Verifying password - Enter PEM pass phrase:
# -----
# You are about to be asked to enter information that will be
# incorporated into your certificate request.
# What you are about to enter is what is called a Distinguished Name
# or a DN.
# There are quite a few fields but you can leave some blank
# For some fields there will be a default value,
# If you enter '.', the field will be left blank.
# -----
# Country Name (2 letter code) [AU]:FI
# State or Province Name (full name) [Some-State]:.
# Locality Name (eg, city) []:
# Organization Name (eg, company) [Internet Widgits Pty Ltd]:MySQL AB
# Organizational Unit Name (eg, section) []:
# Common Name (eg, YOUR name) []:MySQL user
# Email Address []:
#
# Please enter the following 'extra' attributes
# to be sent with your certificate request
# A challenge password []:
# An optional company name []:
#
# Remove the passphrase from the key
#
openssl rsa -in $DIR/client-key.pem -out $DIR/client-key.pem
#
# Sign client cert
#
openssl ca  -policy policy_anything -out $DIR/client-cert.pem \
    -config $DIR/openssl.cnf -infiles $DIR/client-req.pem
# Sample output:
# Using configuration from /home/monty/openssl/openssl.cnf
# Enter PEM pass phrase:
# Check that the request matches the signature
# Signature ok
# The Subjects Distinguished Name is as follows
# countryName           :PRINTABLE:'FI'
# organizationName      :PRINTABLE:'MySQL AB'
# commonName            :PRINTABLE:'MySQL user'
# Certificate is to be certified until Sep 13 16:45:17 2003 GMT
# (365 days)
# Sign the certificate? [y/n]:y
#
#
# 1 out of 1 certificate requests certified, commit? [y/n]y
# Write out database with 1 new entries
# Data Base Updated
#
# Create a my.cnf file that you can use to test the certificates
#
cnf=""
cnf="$cnf [client]"
cnf="$cnf ssl-ca=$DIR/cacert.pem"
cnf="$cnf ssl-cert=$DIR/client-cert.pem"
cnf="$cnf ssl-key=$DIR/client-key.pem"
cnf="$cnf [mysqld]"
cnf="$cnf ssl-ca=$DIR/cacert.pem"
cnf="$cnf ssl-cert=$DIR/server-cert.pem"
cnf="$cnf ssl-key=$DIR/server-key.pem"
echo $cnf | replace " " '
' > $DIR/my.cnf
```

Example 3: Creating SSL files on Windows

Download OpenSSL for Windows. An overview of available packages can be seen here: http://www.slproweb.com/products/Win32OpenSSL.html

Choose of the following packages, depending on your architecture (32-bit or 64-bit):

- Win32 OpenSSL v0.9.8j Light, available at: http://www.slproweb.com/download/Win32OpenSSL_Light-0_9_8j.exe
- Win64 OpenSSL v0.9.8j Light, available at: http://www.slproweb.com/download/Win64OpenSSL_Light-0_9_8j.exe

if a message occurs during setup indicating `'...critical component is missing: Microsoft Visual C++ 2008 Redistributables'`, cancel the setup and download one of the following packages as well, again depending on your architecture (32-bit or 64-bit):

- Visual C++ 2008 Redistributables (x86), available at: http://www.microsoft.com/downloads/details.aspx?familyid=9B2DA534-3E03-4391-8A4D-074B9F2BC1BF“isplaylang=en
- Visual C++ 2008 Redistributables (x64), available at: http://www.microsoft.com/downloads/details.aspx?familyid=bd2a6171-e2d6-4230-b809-9a8d7548c1b6“isplaylang=en

After installing the additional package, restart the OpenSSL setup.

During installation, leave the default `C:\OpenSSL` as the install path, and also leave the default option `'Copy OpenSSL DLL files to the Windows system directory'` selected.

When the installation has finished, add `C:\OpenSSL\bin` to the Windows System Path variable of your server:

- On the Windows desktop, right-click on the My Computer icon, and select Properties.
- Next select the Advanced tab from the SYSTEM PROPERTIES menu that appears, and click the ENVIRONMENT VARIABLES button.
- Under **SYSTEM VARIABLES**, select Path, and then click the EDIT button. The EDIT SYSTEM VARIABLE dialogue should appear.
- Add `';C:\OpenSSL\bin'` to the end (notice the semicolon).
- Press OK 3 times.
- Check that OpenSSL was correctly integrated into the Path variable by opening a new command console (`Start>Run>cmd.exe`) and verifying that OpenSSL is available:

```
Microsoft Windows [Version ...]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.
C:\Windows\system32>cd \
C:\>openssl
OpenSSL> exit <<< If you see the OpenSSL prompt, installation was successful.
C:\>
```

Depending on your version of Windows, the preceding instructions might be slightly different.

After OpenSSL has been installed, use the instructions from Example 1 (shown earlier in this section), with the following changes:

- Change the follow Unix commands:

```
# Create clean environment
shell> rm -rf newcerts
shell> mkdir newcerts && cd newcerts
```

  On Windows, use these commands instead:

```
# Create clean environment
shell> md c:\newcerts
shell> cd c:\newcerts
```

- When a `'\'` character is shown at the end of a command line, this `'\'` character must be removed and the command lines entered all on a single line.
- For references to `my.cnf` option files, substitute `my.ini` instead.

Testing SSL connections

To test SSL connections, start the server as follows, where `$DIR` is the path name to the directory where the sample `my.cnf` option file is located:

```
shell> mysqld --defaults-file=$DIR/my.cnf &
```

Then invoke a client program using the same option file:

```
shell> mysql --defaults-file=$DIR/my.cnf
```

If you have a MySQL source distribution, you can also test your setup by modifying the preceding `my.cnf` file to refer to the demonstration certificate and key files in the `mysql-test/std_data` directory of the distribution.

4.8. Connecting to MySQL Remotely from Windows with SSH

This section describes how to get a secure connection to a remote MySQL server with SSH. The information was provided by David Carlson <dcarlson@mplcomm.com>.

1. Install an SSH client on your Windows machine. As a user, the best non-free one I have found is from `SecureCRT` from http://www.vandyke.com/. Another option is `f-secure` from http://www.f-secure.com/. You can also find some free ones on `Google` at http://directory.google.com/Top/Computers/Internet/Protocols/SSH/Clients/Windows/.

2. Start your Windows SSH client. Set `Host_Name = yourmysqlserver_URL_or_IP`. Set `userid=your_userid` to log in to your server. This `userid` value might not be the same as the user name of your MySQL account.

3. Set up port forwarding. Either do a remote forward (Set `local_port: 3306`, `remote_host: yourmysqlservername_or_ip`, `remote_port: 3306`) or a local forward (Set `port: 3306`, `host: localhost`, `remote port: 3306`).

4. Save everything, otherwise you will have to redo it the next time.

5. Log in to your server with the SSH session you just created.

6. On your Windows machine, start some ODBC application (such as Access).

7. Create a new file in Windows and link to MySQL using the ODBC driver the same way you normally do, except type in `localhost` for the MySQL host server, not `yourmysqlservername`.

At this point, you should have an ODBC connection to MySQL, encrypted using SSH.

4.9. Auditing MySQL Account Activity

Applications can use the following guidelines to perform auditing that ties database activity to MySQL accounts.

MySQL accounts correspond to rows in the `mysql.user` table. When a client connects successfully, the server authenticates the client to a particular row in this table. The `User` and `Host` column values in this row uniquely identify the account and correspond to the `'user_name'@'host_name'` format in which account names are written in SQL statements.

The account used to authenticate a client determines which privileges the client has. Normally, the `CURRENT_USER()` function can be invoked to determine which account this is for the client user. Its value is constructed from the `User` and `Host` columns of the `user` table row for the account.

However, there are circumstances under which the `CURRENT_USER()` value corresponds not to the client user but to a different account. This occurs in contexts when privilege checking is not based the client's account:

- Stored routines (procedures and functions) defined with the `SQL SECURITY DEFINER` characteristic
- Views defined with the `SQL SECURITY DEFINER` characteristic
- Triggers and events

In those contexts, privilege checking is done against the `DEFINER` account and `CURRENT_USER()` refers to that account, not to the account for the client who invoked the stored routine or view or who caused the trigger to activate. To determine the invoking user, you can call the `USER()` function, which returns a value indicating the actual user name provided by the client and the host from which the client connected. However, this value does not necessarily correspond directly to an account in the `user` table, because the `USER()` value never contains wildcards, whereas account values (as returned by `CURRENT_USER()`) may contain user name and host name wildcards.

For example, a blank user name matches any user, so an account of `''@'localhost'` enables clients to connect as an anonym-

ous user from the local host with any user name. If this case, if a client connects as `user1` from the local host, `USER()` and `CURRENT_USER()` return different values:

```
mysql> SELECT USER(), CURRENT_USER();
+----------------+----------------+
| USER()         | CURRENT_USER() |
+----------------+----------------+
| user1@localhost | @localhost     |
+----------------+----------------+
```

The host name part of an account can contain wildcards, too. If the host name contains a `'%'` or `'_'` pattern character or uses netmask notation, the account can be used for clients connecting from multiple hosts and the `CURRENT_USER()` value will not indicate which one. For example, the account `'user2'@'%.example.com'` can be used by `user2` to connect from any host in the `example.com` domain. If `user2` connects from `remote.example.com`, `USER()` and `CURRENT_USER()` return different values:

```
mysql> SELECT USER(), CURRENT_USER();
+--------------------------+--------------------+
| USER()                   | CURRENT_USER()     |
+--------------------------+--------------------+
| user2@remote.example.com | user2@%.example.com |
+--------------------------+--------------------+
```

If an application must invoke `USER()` for user auditing (for example, if it does auditing from within triggers) but must also be able to associate the `USER()` value with an account in the `user` table, it is necessary to avoid accounts that contain wildcards in the `User` or `Host` column. Specifically, do not allow `User` to be empty (which creates an anonymous-user account), and do not allow pattern characters or netmask notation in `Host` values. All accounts must have a non-empty `User` value and literal `Host` value.

With respect to the previous examples, the `''@'localhost'` and `'user2'@'%.example.com'` accounts should be changed not to use wildcards:

```
RENAME USER ''@'localhost' TO 'user1'@'localhost';
RENAME USER 'user2'@'%.example.com' TO 'user2'@'remote.example.com';
```

If `user2` must be able to connect from several hosts in the `example.com` domain, there should be a separate account for each host.

To extract the user name or host name part from a `CURRENT_USER()` or `USER()` value, use the `SUBSTRING()` function:

```
mysql> SELECT SUBSTRING_INDEX(CURRENT_USER(),'@',1);
+---------------------------------------+
| SUBSTRING_INDEX(CURRENT_USER(),'@',1) |
+---------------------------------------+
| user1                                 |
+---------------------------------------+
mysql> SELECT SUBSTRING_INDEX(CURRENT_USER(),'@',-1);
+----------------------------------------+
| SUBSTRING_INDEX(CURRENT_USER(),'@',-1) |
+----------------------------------------+
| localhost                              |
+----------------------------------------+
```

Chapter 5. Backup and Recovery

It is important to back up your databases in case problems occur so that you can recover your data and be up and running again. MySQL offers a variety of backup strategies from which you can choose to select whatever methods best suit the requirements for your installation.

Briefly summarized, backup concepts with which you should be familiar include the following:

- Logical versus physical backups
- Online versus offline backups
- Local versus remote backups
- Snapshot backups
- Full versus incremental backups
- Point-in-time recovery
- Backup scheduling, compression, and encryption
- Table maintenance

More generally, the following discussion amplifies on the properties of different backup methods.

- **Logical versus physical (raw) backups.** Logical backups save information represented as logical database structure (`CREATE DATABASE`, `CREATE TABLE` statements) and content (`INSERT` statements or delimited-text files). Physical backups consist of raw copies of the directories and files that store database contents.

  Logical backup methods have these characteristics:

  - The backup is done by going through the MySQL server to obtain database structure and content information.
  - Backup is slower than physical methods because the server must access database information, convert it to logical format, and send it to the backup program.
  - Output is larger than for physical backup, particularly when saved in text format.
  - Backup and restore granularity is available at the server level (all databases), database level (all tables in a particular database), or table level. This is true regardless of storage engine.
  - The backup does not include log or configuration files, or other database-related files that are not part of databases.
  - Backups stored in logical format are machine independent and highly portable.
  - Logical backups are performed with the MySQL server running (the server is not taken offline).
  - Logical backup tools include the `mysqldump` program and the `SELECT ... INTO OUTFILE` statement. These work for any storage engine, even `MEMORY`.

    For restore, SQL-format dump files can be processed using the `mysql` client. To load delimited-text files, use the `LOAD DATA INFILE` statement or the `mysqlimport` client.

  Physical backup methods have these characteristics:

  - The backup consists of exact copies of database directories and files. Typically this is a copy of all or part of the MySQL data directory. Data from `MEMORY` tables cannot be backed up this way because their contents are not stored on disk.
  - Physical backup methods are faster than logical because they involve only file copying without conversion.
  - Output is more compact than for logical backup.
  - Backup and restore granularity extends from the level of the entire data directory down to the level of individual files. This may or may not provide for table-level granularity, depending on storage engine. (Each `MyISAM` table corresponds uniquely to a set of files, but an `InnoDB` table shares file storage with other `InnoDB` tables.)
  - In addition to databases, the backup can include any related files such as log or configuration files.

  - Backups are portable only to other machines that have identical or similar hardware characteristics.
  - Backups can be performed while the MySQL server is not running. If the server is running, it is necessary to perform appropriate locking so that the server does not change database contents during the backup.
  - Physical backup tools include file system-level commands (such as `cp`, `scp`, `tar`, `rsync`), `mysqlhotcopy` for `MyISAM` tables, `ibbackup` for `InnoDB` tables, or `START BACKUP` for `NDB` tables.

    For restore, files copied at the file system level or with `mysqlhotcopy` can be copied back to their original locations with file system commands; `ibback` restores `InnoDB` tables, and `ndb_restore` restores `NDB` tables.

- **Online versus offline backups.** Online backups take place while the MySQL server is running so that the database information can be obtained from the server. Offline backups take place while the server is stopped. (This distinction can also be described as “hot” versus “cold” backups; a “warm” backup is one where the server remains running but locked against modifying data while you access database files externally.)

  Online backup methods have these characteristics:

  - Less intrusive to other clients, which can connect to the MySQL server during the backup and may be able to access data depending on what operations they need to perform.
  - Care must be taken to impose appropriate locking so that data modifications do not take place that compromise backup integrity.

  Offline backup methods have these characteristics:

  - Affects clients adversely because the server is unavailable during backup.
  - Simpler backup procedure because there is no possibility of interference from client activity.

- **Local versus remote backups.** A local backup is performed on the same host where the MySQL server runs, whereas a remote backup is initiated from a different host.

  - `mysqldump` can connect to local or remote servers. For SQL output (`CREATE` and `INSERT` statements), local or remote dumps can be done and generate output on the client. For delimited-text output (with the `--tab` option), data files are created on the server host.
  - `mysqlhotcopy` performs only local backups: It connects to the server to lock it against data modifications and then copies local table files.
  - `SELECT ... INTO OUTFILE` can be initiated from a remote client host, but the output file is created on the server host.
  - Physical backup methods typically are initiated locally on the MySQL server host so that the server can be taken offline, although the destination for file copies might be remote.

- **Snapshot backups.** Some file system implementations enable “snapshots” to be taken. These provide logical copies of the file system at a given point in time, without having to physically copy the entire file system. (For example, the implementation may use copy-on-write techniques so that only parts of the file system modified after the snapshot time need be copied.) MySQL itself does not provide the capability for taking file system snapshots. It is available through third-party solutions such as Veritas or LVM.

- **Full versus incremental backups.** A full backup includes all data managed by a MySQL server at a given point in time. An incremental backup consists of the changes made to the data since the full backup. MySQL has different ways to perform full backups, such as those described in previous items. Incremental backups are made possible by enabling the server's binary log, which the server uses to record data changes.

- **Point-in-time recovery.** One use for the binary log is to achieve point-in-time recovery. This is done by recovering first from the backup files to restore the server to its state when the backup was made, and then by re-executing changes in subsequently written binary log files to redo data modifications up to the desired point in time.

- **Backup scheduling, compression, and encryption.** Backup scheduling is valuable for automating backup procedures. Compression of backup output reduces space requirements, and encryption of the output provides better security against unauthorized access of backed-up data. MySQL itself does not provide these capabilities. `ibbackup` can compress `InnoDB` backups, and compression or encryption of backup output can be achieved using file system utilities. Other third-party solutions may be available.

- **Table maintenance.** Data integrity can be compromised if tables become corrupt. MySQL provides programs for checking tables and repairing them should problems be found. These programs apply primarily to `MyISAM` tables. See Section 5.5, “Table Maintenance and Crash Recovery”.

Additional resources

Resources related to backup or to maintaining data availability include the following:

- A forum dedicated to backup issues is available at http://forums.mysql.com/list.php?93.
- The syntax of the SQL statements described here is given in SQL Statement Syntax.
- Details for `mysqldump`, `mysqlhotcopy`, and other MySQL backup programs can be found in MySQL Programs.
- For additional information about `InnoDB` backup procedures, see Backing Up and Recovering an `InnoDB` Database.
- Replication enables you to maintain identical data on multiple servers. This has several benefits, such as allowing client load to be distributed over servers, availability of data even if a given server is taken offline or fails, and the ability to make backups using a slave server without affecting the master. See Replication.
- MySQL Cluster provides a high-availability, high-redundancy version of MySQL adapted for the distributed computing environment. See MySQL Cluster NDB 6.X/7.X, which provides information about MySQL Cluster NDB 6.2 and 6.3 (based on MySQL 5.1 but containing the latest improvements and fixes for the `NDBCLUSTER` storage engine).

  Note

  The `NDBCLUSTER` storage engine is currently not supported in MySQL 6.0.
- Distributed Replicated Block Device (DRBD) is another high-availability solution. It works by replicating a block device from a primary server to a secondary server at the block level. See High Availability and Scalability

5.1. Database Backups

This section summarizes some general methods for making backups.

Making Backups by Copying Files

`MyISAM` tables are stored as files, so it is easy to do a backup by copying files. To get a consistent backup, do a `LOCK TABLES` on the relevant tables, followed by `FLUSH TABLES` for the tables. See `LOCK TABLES` and `UNLOCK TABLES` Syntax, and `FLUSH` Syntax. You need only a read lock; this allows other clients to continue to query the tables while you are making a copy of the files in the database directory. The `FLUSH TABLES` statement is needed to ensure that the all active index pages are written to disk before you start the backup.

Making Delimited-Text File Backups

To create a text file containing a table's data, you can use `SELECT * INTO OUTFILE 'file_name' FROM tbl_name`. The file is created on the MySQL server host, not the client host. For this statement, the output file cannot already exist because allowing files to be overwritten would constitute a security risk. See `SELECT` Syntax. This method works for any kind of data file, but saves only table data, not the table structure.

To reload the output file, use `LOAD DATA INFILE` or `mysqlimport`.

Making Backups with `mysqldump` or `mysqlhotcopy`

Another technique for backing up a database is to use the `mysqldump` program or the `mysqlhotcopy` script. `mysqldump` is more general because it can back up all kinds of tables. `mysqlhotcopy` works only with some storage engines. (See `mysqldump`, and `mysqlhotcopy`.)

Create a full backup of your database:

```
shell> mysqldump --tab=/path/to/some/dir --opt db_name
```

Or:

```
shell> mysqlhotcopy db_name /path/to/some/dir
```

You can also create a binary backup simply by copying all table files (`*.frm`, `*.MYD`, and `*.MYI` files), as long as the server isn't updating anything. The `mysqlhotcopy` script uses this method. (But note that these methods do not work if your database contains `InnoDB` tables. `InnoDB` does not necessarily store table contents in database directories, and `mysqlhotcopy` works only for `MyISAM` and `ISAM` tables.)

For `InnoDB` tables, it is possible to perform an online backup that takes no locks on tables; see `mysqldump`.

Using the Binary Log to Enable Incremental Backups

MySQL supports incremental backups: You must start the server with the `--log-bin` option to enable binary logging; see The Binary Log. The binary log files provide you with the information you need to replicate changes to the database that are made subsequent to the point at which you performed a backup. At the moment you want to make an incremental backup (containing all changes that happened since the last full or incremental backup), you should rotate the binary log by using `FLUSH LOGS`. This done, you need to copy to the backup location all binary logs which range from the one of the moment of the last full or incremental backup to the last but one. These binary logs are the incremental backup; at restore time, you apply them as explained in Section 5.4, "Point-in-Time Recovery". The next time you do a full backup, you should also rotate the binary log using `FLUSH LOGS`, `mysqldump --flush-logs`, or `mysqlhotcopy --flushlog`. See `mysqldump`, and `mysqlhotcopy`.

Backing Up Replication Slaves

If your MySQL server is a slave replication server, then regardless of the backup method you choose, you should also back up the `master.info` and `relay-log.info` files when you back up your slave's data. These files are always needed to resume replication after you restore the slave's data. If your slave is subject to replicating `LOAD DATA INFILE` commands, you should also back up any `SQL_LOAD-*` files that may exist in the directory specified by the `--slave-load-tmpdir` option. (This location defaults to the value of the `tmpdir` system variable if not specified.) The slave needs these files to resume replication of any interrupted `LOAD DATA INFILE` operations.

> **MySQL Enterprise**
> The MySQL Enterprise Monitor provides numerous advisors that issue immediate warnings should replication issues arise. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

If you have performance problems with your master server while making backups, one strategy that can help is to set up replication and perform backups on the slave rather than on the master. See Replication.

Recovering Corrupt Tables

If you have to restore `MyISAM` tables that have become corrupt, try to recover them using `REPAIR TABLE` or `myisamchk -r` first. That should work in 99.9% of all cases. If `myisamchk` fails, try the following procedure. It is assumed that you have enabled binary logging by starting MySQL with the `--log-bin` option.

1. Restore the original `mysqldump` backup, or binary backup.
2. Execute the following command to re-run the updates in the binary logs:

   ```
   shell> mysqlbinlog binlog.[0-9]* | mysql
   ```

   In some cases, you may want to re-run only certain binary logs, from certain positions (usually you want to re-run all binary logs from the date of the restored backup, excepting possibly some incorrect statements). See Section 5.4, "Point-in-Time Recovery".

Making Backups Using a File System Snapshot

If you are using a Veritas file system, you can make a backup like this:

1. From a client program, execute `FLUSH TABLES WITH READ LOCK`.
2. From another shell, execute `mount vxfs snapshot`.
3. From the first client, execute `UNLOCK TABLES`.
4. Copy files from the snapshot.
5. Unmount the snapshot.

5.2. Example Backup and Recovery Strategy

This section discusses a procedure for performing backups that allows you to recover data after several types of crashes:

- Operating system crash
- Power failure

- File system crash
- Hardware problem (hard drive, motherboard, and so forth)

The example commands do not include options such as `--user` and `--password` for the `mysqldump` and `mysql` programs. You should include such options as necessary so that the MySQL server allows you to connect to it.

We assume that data is stored in the `InnoDB` storage engine, which has support for transactions and automatic crash recovery. We also assume that the MySQL server is under load at the time of the crash. If it were not, no recovery would ever be needed.

For cases of operating system crashes or power failures, we can assume that MySQL's disk data is available after a restart. The `InnoDB` data files might not contain consistent data due to the crash, but `InnoDB` reads its logs and finds in them the list of pending committed and non-committed transactions that have not been flushed to the data files. `InnoDB` automatically rolls back those transactions that were not committed, and flushes to its data files those that were committed. Information about this recovery process is conveyed to the user through the MySQL error log. The following is an example log excerpt:

```
InnoDB: Database was not shut down normally.
InnoDB: Starting recovery from log files...
InnoDB: Starting log scan based on checkpoint at
InnoDB: log sequence number 0 13674004
InnoDB: Doing recovery: scanned up to log sequence number 0 13739520
InnoDB: Doing recovery: scanned up to log sequence number 0 13805056
InnoDB: Doing recovery: scanned up to log sequence number 0 13870592
InnoDB: Doing recovery: scanned up to log sequence number 0 13936128
...
InnoDB: Doing recovery: scanned up to log sequence number 0 20555264
InnoDB: Doing recovery: scanned up to log sequence number 0 20620800
InnoDB: Doing recovery: scanned up to log sequence number 0 20664692
InnoDB: 1 uncommitted transaction(s) which must be rolled back
InnoDB: Starting rollback of uncommitted transactions
InnoDB: Rolling back trx no 16745
InnoDB: Rolling back of trx no 16745 completed
InnoDB: Rollback of uncommitted transactions completed
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Apply batch completed
InnoDB: Started
mysqld: ready for connections
```

For the cases of file system crashes or hardware problems, we can assume that the MySQL disk data is *not* available after a restart. This means that MySQL fails to start successfully because some blocks of disk data are no longer readable. In this case, it is necessary to reformat the disk, install a new one, or otherwise correct the underlying problem. Then it is necessary to recover our MySQL data from backups, which means that we must already have made backups. To make sure that is the case, we should design a backup policy.

5.2.1. Backup Policy

We all know that backups must be scheduled periodically. A full backup (a snapshot of the data at a point in time) can be done in MySQL with several tools. For example, `InnoDB Hot Backup` provides online non-blocking physical backup of the `InnoDB` data files, and `mysqldump` provides online logical backup. This discussion uses `mysqldump`.

> **MySQL Enterprise**
> For expert advice on backups and replication, subscribe to the MySQL Enterprise Monitor. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

Assume that we make a backup on Sunday at 1 p.m., when load is low. The following command makes a full backup of all our `InnoDB` tables in all databases:

```
shell> mysqldump --single-transaction --all-databases > backup_sunday_1_PM.sql
```

This backup acquires a global read lock on all tables (using `FLUSH TABLES WITH READ LOCK`) at the beginning of the dump. As soon as this lock has been acquired, the binary log coordinates are read and the lock is released. If long updating statements are running when the `FLUSH` statement is issued, the MySQL server may get stalled until those statements finish. After that, the dump becomes lock-free and does not disturb reads and writes on the tables.

We assumed earlier that our tables are `InnoDB` tables, so `--single-transaction` uses a consistent read and guarantees that data seen by `mysqldump` does not change. (Changes made by other clients to `InnoDB` tables are not seen by the `mysqldump` process.) If we do also have other types of tables, we must assume that they are not changed during the backup. For example, for the `MyISAM` tables in the `mysql` database, we must assume that no administrative changes are being made to MySQL accounts during the backup.

The resulting `.sql` file produced by `mysqldump` contains a set of SQL `INSERT` statements that can be used to reload the dumped tables at a later time.

Full backups are necessary, but they are not always convenient. They produce large backup files and take time to generate. They are not optimal in the sense that each successive full backup includes all data, even that part that has not changed since the previous full backup. After we have made the initial full backup, it is more efficient to make incremental backups. They are smaller and take less time to produce. The tradeoff is that, at recovery time, you cannot restore your data just by reloading the full backup. You must also process the incremental backups to recover the incremental changes.

To make incremental backups, we need to save the incremental changes. The MySQL server should always be started with the `--log-bin` option so that it stores these changes in a file while it updates data. This option enables binary logging, so that the server writes each SQL statement that updates data into a file called a MySQL binary log. Looking at the data directory of a MySQL server that was started with the `--log-bin` option and that has been running for some days, we find these MySQL binary log files:

```
-rw-rw---- 1 guilhem  guilhem   1277324 Nov 10 23:59 gbichot2-bin.000001
-rw-rw---- 1 guilhem  guilhem         4 Nov 10 23:59 gbichot2-bin.000002
-rw-rw---- 1 guilhem  guilhem        79 Nov 11 11:06 gbichot2-bin.000003
-rw-rw---- 1 guilhem  guilhem       508 Nov 11 11:08 gbichot2-bin.000004
-rw-rw---- 1 guilhem  guilhem 220047446 Nov 12 16:47 gbichot2-bin.000005
-rw-rw---- 1 guilhem  guilhem    998412 Nov 14 10:08 gbichot2-bin.000006
-rw-rw---- 1 guilhem  guilhem       361 Nov 14 10:07 gbichot2-bin.index
```

Each time it restarts, the MySQL server creates a new binary log file using the next number in the sequence. While the server is running, you can also tell it to close the current binary log file and begin a new one manually by issuing a `FLUSH LOGS` SQL statement or with a `mysqladmin flush-logs` command. `mysqldump` also has an option to flush the logs. The `.index` file in the data directory contains the list of all MySQL binary logs in the directory. This file is used for replication.

The MySQL binary logs are important for recovery because they form the set of incremental backups. If you make sure to flush the logs when you make your full backup, then any binary log files created afterward contain all the data changes made since the backup. Let's modify the previous `mysqldump` command a bit so that it flushes the MySQL binary logs at the moment of the full backup, and so that the dump file contains the name of the new current binary log:

```
shell> mysqldump --single-transaction --flush-logs --master-data=2 \
         --all-databases > backup_sunday_1_PM.sql
```

After executing this command, the data directory contains a new binary log file, `gbichot2-bin.000007`. The resulting `.sql` file includes these lines:

```
-- Position to start replication or point-in-time recovery from
-- CHANGE MASTER TO MASTER_LOG_FILE='gbichot2-bin.000007',MASTER_LOG_POS=4;
```

Because the `mysqldump` command made a full backup, those lines mean two things:

- The `.sql` file contains all changes made before any changes written to the `gbichot2-bin.000007` binary log file or newer.
- All data changes logged after the backup are not present in the `.sql`, but are present in the `gbichot2-bin.000007` binary log file or newer.

On Monday at 1 p.m., we can create an incremental backup by flushing the logs to begin a new binary log file. For example, executing a `mysqladmin flush-logs` command creates `gbichot2-bin.000008`. All changes between the Sunday 1 p.m. full backup and Monday 1 p.m. will be in the `gbichot2-bin.000007` file. This incremental backup is important, so it is a good idea to copy it to a safe place. (For example, back it up on tape or DVD, or copy it to another machine.) On Tuesday at 1 p.m., execute another `mysqladmin flush-logs` command. All changes between Monday 1 p.m. and Tuesday 1 p.m. will be in the `gbichot2-bin.000008` file (which also should be copied somewhere safe).

The MySQL binary logs take up disk space. To free up space, purge them from time to time. One way to do this is by deleting the binary logs that are no longer needed, such as when we make a full backup:

```
shell> mysqldump --single-transaction --flush-logs --master-data=2 \
         --all-databases --delete-master-logs > backup_sunday_1_PM.sql
```

> **Note**
>
> Deleting the MySQL binary logs with `mysqldump --delete-master-logs` can be dangerous if your server is a replication master server, because slave servers might not yet fully have processed the contents of the binary log. The description for the `PURGE BINARY LOGS` statement explains what should be verified before deleting the MySQL binary logs. See `PURGE BINARY LOGS` Syntax.

5.2.2. Using Backups for Recovery

Now, suppose that we have a catastrophic crash on Wednesday at 8 a.m. that requires recovery from backups. To recover, first we restore the last full backup we have (the one from Sunday 1 p.m.). The full backup file is just a set of SQL statements, so restoring it is very easy:

```
shell> mysql < backup_sunday_1_PM.sql
```

At this point, the data is restored to its state as of Sunday 1 p.m.. To restore the changes made since then, we must use the incremental backups; that is, the `gbichot2-bin.000007` and `gbichot2-bin.000008` binary log files. Fetch the files if necessary from where they were backed up, and then process their contents like this:

```
shell> mysqlbinlog gbichot2-bin.000007 gbichot2-bin.000008 | mysql
```

We now have recovered the data to its state as of Tuesday 1 p.m., but still are missing the changes from that date to the date of the crash. To not lose them, we would have needed to have the MySQL server store its MySQL binary logs into a safe location (RAID disks, SAN, ...) different from the place where it stores its data files, so that these logs were not on the destroyed disk. (That is, we can start the server with a `--log-bin` option that specifies a location on a different physical device from the one on which the data directory resides. That way, the logs are safe even if the device containing the directory is lost.) If we had done this, we would have the `gbichot2-bin.000009` file at hand, and we could apply it using `mysqlbinlog` and `mysql` to restore the most recent data changes with no loss up to the moment of the crash.

5.2.3. Backup Strategy Summary

In case of an operating system crash or power failure, `InnoDB` itself does all the job of recovering data. But to make sure that you can sleep well, observe the following guidelines:

- Always run the MySQL server with the `--log-bin` option, or even `--log-bin=`*`log_name`*, where the log file name is located on some safe media different from the drive on which the data directory is located. If you have such safe media, this technique can also be good for disk load balancing (which results in a performance improvement).
- Make periodic full backups, using the `mysqldump` command shown earlier in Section 5.2.1, “Backup Policy”, that makes an online, non-blocking backup.
- Make periodic incremental backups by flushing the logs with `FLUSH LOGS` or `mysqladmin flush-logs`.

5.3. Using MySQL Backup

MySQL Backup is available as of MySQL 6.0.5. This feature comprises the `BACKUP DATABASE` and `RESTORE` statements. They provide a way to make a copy of a database or set of databases at a given point in time, and a way to restore each database to its state as of that time.

A backup operation can include tables for different storage engines and the backup image will still be consistent. That is, you need not care which storage engines you're using. `BACKUP DATABASE` saves the data in a consistent backup image with respect to its “validity point.”

The validity point ties the backup to the binary log. Restoring a backup can be combined with use of the binary log to accomplish point-in-time recovery: If the restore operation is done because data loss has occurred after the backup was made (that is, after the validity point), restored databases can be brought up to the time of data loss by executing the data changes in the binary log between the times when the backup was made and when the data loss occurred.

A goal of the `BACKUP DATABASE` and `RESTORE` statements is to enable other database operations to proceed concurrently, to make it unnecessary to take databases offline or prevent clients from accessing them. `BACKUP DATABASE` must block some operations from occurring (such as dropping tables from a database while it is being backed up), but the attempt is made to keep blocking to a minimum. Generally, blocked operations are those involving Data Definition Language (DDL) statements. `RESTORE` must do more blocking because it writes database contents rather than just reading them.

The following discussion covers these aspects of `BACKUP DATABASE` and `RESTORE`:

- Quick guide to making backups and restoring them
- How `BACKUP DATABASE` and `RESTORE` work
- Status reporting and monitoring for backup and restore operations

For additional information about the `BACKUP DATABASE` and `RESTORE` statements, see these sections of the manual:

- BACKUP DATABASE Syntax, and RESTORE Syntax, describes the syntax for these statements.
- Limitations on the use of these statements are discussed in Restrictions on BACKUP DATABASE and RESTORE.

5.3.1. Quick Guide to MySQL Backup

Use the BACKUP DATABASE and RESTORE statements like this:

- BACKUP DATABASE backs up one or more databases to a named file:

```
BACKUP DATABASE world TO '/tmp/mybackupfile';
```

  To back up more than one database, separate the names by commas:

```
BACKUP DATABASE world, sakila TO '/tmp/mybackupfile';
```

  To select all databases for backup, use the * selector as a shortcut:

```
BACKUP DATABASE * TO '/tmp/mybackupfile';
```

- RESTORE restores databases using the contents of the backup file:

```
RESTORE FROM '/tmp/mybackupfile';
```

BACKUP DATABASE backs up database and table definitions, table data, stored routines, triggers, events, and views. TEMPORARY tables are not included. Tablespace backup support is limited to the Falcon storage engine.

Prior to MySQL 6.0.7, BACKUP DATABASE did not save any privileges in the backup image file and RESTORE did not restore privileges. As of MySQL 6.0.7, privileges are saved and restored according to these rules:

- BACKUP DATABASE saves privileges for the backed-up databases in the backup image file. The privileges are stored in the form of GRANT statements.
- Only privileges are the database level or below (table, column, routine) are saved. Global privileges are not saved because they are not specific to the databases included in the backup.
- Privileges that specify the database name using a pattern (containing the '%' or '_' wildcard character) are not saved because they might apply to databases not included in the backup.
- For restore operations, only those privileges are restore that pertain to accounts that exist on the MySQL server performing the restore. Other privileges are ignored with a warning. (These warnings can be displayed with SHOW WARNINGS.) Suppose that a backup contains this GRANT statement:

```
GRANT SELECT, INSERT ON db1.* to 'someuser'@'localhost'
```

  The privileges specified by this statement will be restored if the 'someuser'@'localhost' account exists, and ignored with a warning otherwise.

  Restoration of privileges for accounts that do not exist is not done because that would implicitly create accounts that have no password, which is a security risk.

Storage of GRANT statement in backup image files has a security implication: Backup images should be stored in a secure location so that unauthorized users cannot modify the GRANT statements contained therein to change the privileges granted by restore operations.

For anything else not explicitly listed, assume that it is not backed up. This includes but is not limited to items such as UDF definitions and files, logs, and option files.

BACKUP DATABASE currently does not back up the contents of the mysql database. This database contains the grant tables that define user accounts and their privileges, as well as other system information. To make a full server instance backup that includes account information in addition to data, use the BACKUP DATABASE statement together with the mysqldump program. In the following instructions, *path* represents the full path name to the directory where you store your backup files.

1. Use `mysqldump` to back up the `mysql` database. This is a blocking operation that prevents changes to the database during the dump, but the `mysql` database normally is relatively small and can be dumped quickly:

   ```
   shell> mysqldump --databases mysql > path/mysql-db.sql
   ```

2. Use `BACKUP DATABASE` to back up the data from other databases. This is a non-blocking operation:

   ```
   mysql> BACKUP DATABASE * TO 'path/other-dbs.bak';
   ```

Restore the server instance later like this:

1. To restore the user accounts, reload the `mysql` database dump file using the `mysql` client:

   ```
   shell> mysql -u root -p < path/mysql-db.sql
   ```

2. To restore the data for other databases, use `RESTORE` with the image file produced by `BACKUP DATABASE`:

   ```
   mysql> RESTORE FROM 'path/other-dbs.bak';
   ```

For more information about the operation of the `BACKUP DATABASE` and `RESTORE` statements, see `BACKUP DATABASE` Syntax, and `RESTORE` Syntax.

5.3.2. How MySQL Backup Works

A backup operation creates a backup of one or more databases at a given point in time and saves it as a backup image, a file that contains the backup data (table contents) and metadata (definitions for databases, tables, and other objects, and server information).

The backup is intended to provide a consistent snapshot of the backed-up data as of the point at which the operation began, and it is intended to provide online operation as much as possible that allows other server activity to proceed without blocking.

A backup operation begins at time $t1$ and ends at time $t2$, producing a backup image that contains the backup state (database state) at time t, where $t1 < t < t2$. The time t is called the validity point of the backup image. It represents the time when all storage engines are synchronized for the backup. Restoring this image restores the state to be the same as it was at time t.

Consistency of the backup means that these constraints must be true:

- Data from transactional tables is included only for committed transactions.
- Data from non-transactional tables is included only for completed statements.
- Referential integrity is maintained between all backed-up tables within a given backup image.

The referential-integrity constraint does not necessarily hold if two tables are related but only one of them is included in a backup. Restoring the backup then would restore only the backed-up table, which can produce tables for which referential integrity no longer holds.

For a backup to proceed properly, certain types of server activity must be blocked, so the backup system incorporates a commit blocker and a Backup Metadata Lock.

The commit blocker has these properties:

- Changes for non-transactional tables must be blocked.
- Changes for transactional tables are not blocked, but only changes that have been committed when the backup occurs appear in the backup. Changes that occur during the backup operation are not included in the backup image.

When a backup or restore operation is in progress, it is not allowable to modify the structure of database objects. Consequently, during the operation, the Backup Metadata Lock blocks statements that change database metadata from executing. A backup image stores metadata for the following types of objects:

- Databases

- Tablespaces
- Privileges
- Tables
- Views
- Stored programs (functions, procedures, events, triggers)

This requires that the following metadata changes be frozen during backup operation:

- Databases being backed up should not disappear or be changed.
- `BACKUP DATABASE * ...`, new databases should not appear.
- The list of objects inside each database should not change.
- Metadata for objects in the databases should not change.
- The set of privileges for each database should not change.
- Users for which privileges are stored should not disappear or change.
- Tablespaces used by tables being backed up should not disappear or change.

To achieve these requirements, the Backup Metadata Lock blocks the following statements:

```
DROP   DATABASE/TABLE/VIEW/FUNCTION/PROCEDURE/EVENT/TRIGGER/INDEX/
       USER/TABLESPACE
CREATE DATABASE/TABLE/VIEW/FUNCTION/PROCEDURE/EVENT/TRIGGER/INDEX
ALTER  DATABASE/TABLE/VIEW/FUNCTION/PROCEDURE/EVENT/TABLESPACE
RENAME TABLE/USER
GRANT/REVOKE
TRUNCATE/OPTIMIZE/REPAIR TABLE
```

Currently, all instances of statements that change metadata are blocked, even for database or table objects that are not included in the backup. Eventually, the goal is to block only metadata-changing statements for objects in the backup.

Blocking works in both directions. A backup or restore blocks DDL statements, but if a backup or restore operation is initiated while DDL statements are in progress, the operation waits until the statements have finished.

Implementation of `BACKUP DATABASE` and `RESTORE` uses an architecture with the following design:

- The MySQL server communicates with the backup kernel.
- The backup kernel is responsible for communicating with backup engines and for handling metadata (definitions for databases, tables, and other objects, as well as server information).
- Each backup engine provides backup and restore drivers for the backup kernel to use.
- An engine's backup and restore drivers perform actual transfer of data (table contents).

The backup system chooses from among the backup engines available to it:

- There is a default backup engine to be used if a better one is not found. This engine provides default backup and restore drivers that use a blocking algorithm. For example, the backup driver locks all tables at the start of the backup and unlocks them after the last one is processed (which may occur before the operation is complete).
- A consistent-snapshot engine implements the same kind of backup as that made by `mysqldump --single-transaction`.

  The backup driver for the snapshot engine works with only those storage engines that support consistent read via the handler interface, which currently includes only `InnoDB` and `Falcon`. The backup driver creates a logical backup because it reads rows one at a time and returns them to the backup kernel to be stored in the backup image.

A backup image must have contents that are consistent with the binary log coordinates taken from the time of the backup. Otherwise, point-in-time recovery using the backup image plus the binary log contents will not work correctly. `BACKUP DATABASE` synchronizes with binary logging to make sure that the backup image and binary log are consistent with each other. This way, if data loss occurs later, use of the backup image combined with the binary log makes point-in-time recovery possible:

1. Restore the backup image
2. Re-execute binary log contents beginning from the coordinates of the backup's validity point up to the desired point of recovery

5.3.3. MySQL Backup Status Reporting and Monitoring

MySQL provides information about the status or progress of `BACKUP DATABASE` or `RESTORE` operations in the following ways:

- `SHOW PROCESSLIST` displays information while a thread performing a backup or restore is executing.
- Upon successful completion, the `BACKUP DATABASE` and `RESTORE` statements return a result set with the backup number. (This number is the ID for the corresponding row or rows in the metadata tables described later.) Warnings produced during the operation can be displayed with `SHOW WARNINGS`.

  If errors occur during a backup or restore operation, they are written to the error log, recorded in the progress tables, and are available via the `SHOW ERRORS` and `SHOW WARNINGS` statements.

  If a fatal error occurs, the `BACKUP DATABASE` or `RESTORE` statement reports it to the user.
- The server maintains `backup_history` and `backup_progress` tables in the `mysql` database that contain metadata indicating backup status and progress. It is also possible to write log information to files. For information about selecting log destinations, see Section 5.3.3.1, “MySQL Backup Log Control”. For a description of what is logged, see Section 5.3.3.2, “MySQL Backup Log Contents”.

  If you upgrade to MySQL 6.0.5 or later from an older version, be sure to run `mysql_upgrade` to ensure that the backup log tables exist. From MySQL 6.0.5 through 6.0.7, these tables were named `online_backup` and `online_backup_progress`.

  Currently, there are no `INFORMATION_SCHEMA` tables corresponding to the `backup_history` and `backup_progress` tables.

5.3.3.1. MySQL Backup Log Control

MySQL Backup provides status and progress logging. This capability can be enabled or disabled. If logging is enabled, tables in the `mysql` database or log files can be used as the destinations for log output. These features are similar to those provided for the general query log and slow query log (see Selecting General Query and Slow Query Log Output Destinations), although the options and variables are different.

This section describes how to control MySQL Backup logging. For a description of what is logged, see Section 5.3.3.2, “MySQL Backup Log Contents”.

> **Note**
>
> The features described here are available as of MySQL 6.0.8. Before 6.0.8, MySQL Backup logs to the `online_backup` and `online_backup_progress` tables in the `mysql` database. Logging to files is not supported and logging cannot be disabled.

Log control at server startup. The `--log-backup-output` option specifies the destination for log output, if logging is enabled, but the option does not in itself enable the logs. The syntax for this option is `--log-backup-output[=value,...]`:

- If `--log-backup-output` is given with a value, the value can be a comma-separated list of one or more of the words `TABLE` (log to tables), `FILE` (log to files), or `NONE` (do not log to tables or files). `NONE`, if present, takes precedence over any other specifiers.
- If `--log-backup-output` is omitted or given without a value, the default is `TABLE`.

The `--backup_history_log` and `--backup_progress_log` options, if given, enable logging to the history and progress logs for the selected log destinations. (By default, both logs are enabled.) These options take an optional argument of 1 or 0 to en-

able or disable the log. If either log is enabled, the server opens the corresponding log file and writes startup messages to it. However, logging to the file does not occur unless the `FILE` log destination is selected.

Examples:

- With no logging arguments, MySQL Backup logs to the log tables by default.
- To write log entries to the log tables and log files, use `--log-backup-output=TABLE,FILE` to select both log destinations.

Log control at runtime. Several system variables are associated with log tables and files and enable runtime control over logging:

- The global `log_backup_output` system variable indicates the current logging destination. It can be modified at runtime to change the destination.
- The global `backup_history_log` and `backup_progress_log` variables indicate whether the history and progress logs are enabled (`ON`) or disabled (`OFF`). You can set these variables at runtime to control whether the logs are enabled.
- The global `backup_history_log_file` and `backup_progress_log_file` variables indicate the names of the history and progress log files. You can set these variables at server startup or at runtime to change the names of the log files.

5.3.3.2. MySQL Backup Log Contents

If you enable backup logging to tables, MySQL Backup uses the `backup_history` and `backup_progress` tables in the `mysql` database. For logging to files, MySQL Backup uses the `backup_history.log` and `backup_progress.log` files in the MySQL data directory by default. The log file names can be changed by setting the global `backup_history_log_file` and `backup_progress_log_file` system variables. For information about selecting log destinations, see Section 5.3.3.1, “MySQL Backup Log Control”.

The contents of the log tables are discussed following. For logging to files, the server writes lines with a field for each column in the corresponding log table. The server also writes an initial line to the file at startup to indicate the names of the fields. Backup log contents can be culled with the `PURGE BACKUP LOGS` statement. See `PURGE BACKUP LOGS` Syntax.

If the table destination is selected for backup logging, the server uses these tables:

- The `backup_history` table contains a row for each backup and restore operation. A row is created when an operation completes. The rows in this table serve as a history of all backup and restore operations performed on the server. The table can be queried to obtained detailed information about the operations or as a means to create a summary of the operations. The rows are not removed from the table by the server. Any table maintenance, such as removing old rows, is intended to be performed by the database administrator.
- The `backup_progress` table contains progress data describing the steps in the most recent backup or restore operation. There may be multiple rows for the operation. Rows are added to this table over the course of the operation and are not updated. This enables the table to be used to track the current progress of the operation. Each row in the table represents a step in the operation and may contain informational statements, errors, and other pertinent information. The data in this table has a limited lifetime. At the start of each operation, the table is truncated and new data is added. The database administrator should not need to perform maintenance for this data.

The `backup_history` table has this structure:

```
CREATE TABLE backup_history (
    backup_id           BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    process_id          INT UNSIGNED NOT NULL,
    binlog_pos          INT UNSIGNED DEFAULT 0,
    binlog_file         CHAR(64),
    backup_state        ENUM('complete', 'starting', 'validity point',
                             'running', 'error', 'cancel') NOT NULL,
    operation           ENUM('backup', 'restore') NOT NULL,
    error_num           INT NOT NULL DEFAULT 0,
    num_objects         INT UNSIGNED NOT NULL DEFAULT 0,
    total_bytes         BIGINT UNSIGNED,
    validity_point_time DATETIME,
    start_time          DATETIME,
    stop_time           DATETIME,
    host_or_server_name CHAR (30),
    username            CHAR (30),
    backup_file         CHAR (100),
    backup_file_path    VARCHAR (512),
    user_comment        VARCHAR (200),
    command             VARCHAR (512),
    engines             VARCHAR (100),
) ENGINE=CSV CHARSET=utf8;
```

The `backup_history` columns are used as follows:

- `backup_id`

  The ID for the table row. `BACKUP DATABASE` and `RESTORE` return a result set containing a backup ID, which is the value that tells you which row in the `backup_history` table corresponds to the backup or restore operation.

- `process_id`

  The process ID that the operation ran as.

- `binlog_pos`, `binlog_file`

  For a backup, the binary log position and file name at the time the validity point is generated (the time when all storage engines are synchronized). Before that time, the values are 0 and `NULL`.

- `backup_state`

  The status of the operation.

- `operation`

  The type of operation.

- `error_num`

  The error from this operation (0 = no error).

- `num_objects`

  The number of objects in the backup.

- `total_bytes`

  The size of the backup image file in bytes.

- `validity_point_time`

  For a backup, this is the time that the validity point was generated. Before that time, the value is `NULL`. For a restore, the value currently is always `NULL`.

- `start_time`, `stop_time`

  The date and time when the operation started and stopped.

- `host_or_server_name`

  The server name where the operation ran.

- `username`

  The name of the user who ran the operation.

- `backup_file`

  The name of the backup image file. As of MySQL 6.0.8, this column contains the file basename.

- `backup_file_path`

  The directory containing the image file. This column was added in MySQL 6.0.8.

- `user_comment`

  The comment from the user entered at the command line.

- `command`

  The statement used to perform the operation.

- `drivers`

  The names of the drivers used in the operation. Before MySQL 6.0.7, this column was named `engines`.

The `backup_progress` table has this structure:

```
CREATE TABLE backup_progress (
    backup_id   BIGINT UNSIGNED NOT NULL
    object      CHAR (30) NOT NULL
    start_time  DATATIME
    stop_time   DATATIME
    total_bytes BIGINT
    progress    BIGINT UNSIGNED
    error_num   INT NOT NULL DEFAULT 0
    notes       CHAR(100)
) ENGINE=CSV CHARSET=utf8;
```

The `backup_progress` columns are used as follows:

- `backup_id`

  The `backup_id` value of the `backup_history` table row with which the rows in the `backup_progress` table are associated.

- `object`

  The object being operated on.

- `start_time`, `stop_time`

  The date and time when the operation started and stopped.

- `total_bytes`

  The size of the object in bytes.

- `progress`

  The number of bytes processed.

- `error_num`

  The error from this operation (0 = no error).

- `notes`

  Commentary from the backup engine.

5.4. Point-in-Time Recovery

If a MySQL server was started with the `--log-bin` option to enable binary logging, you can use the `mysqlbinlog` utility to recover data from the binary log files, starting from a specified point in time (for example, since your last backup) until the present or another specified point in time. For information on enabling the binary log and using `mysqlbinlog`, see The Binary Log, and `mysqlbinlog`.

> **MySQL Enterprise**
> For maximum data recovery, the MySQL Enterprise Monitor advises subscribers to synchronize to disk at each write. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

To restore data from a binary log, you must know the location and name of the current binary log file. By default, the server creates binary log files in the data directory, but a path name can be specified with the `--log-bin` option to place the files in a different location. Typically the option is given in an option file (that is, `my.cnf` or `my.ini`, depending on your system). It can also be given on the command line when the server is started. To determine the name of the current binary log file, issue the following statement:

```
mysql> SHOW MASTER STATUS
```

If you prefer, you can execute the following command from the command line instead:

```
shell> mysql -u root -p -E -e "SHOW MASTER STATUS"
```

Enter the `root` password for your server when `mysql` prompts you for it.

To view the contents of a binary log, use `mysqlbinlog`. See `mysqlbinlog`.

5.4.1. Specifying Times for Recovery

To indicate the start and end times for recovery, specify the `--start-datetime` and `--stop-datetime` options for `mysqlbinlog`, in `DATETIME` format. As an example, suppose that exactly at 10:00 a.m. on April 20, 2005 an SQL statement was executed that deleted a large table. To restore the table and data, you could restore the previous night's backup, and then execute the following command:

```
shell> mysqlbinlog --stop-datetime="2005-04-20 9:59:59" \
         /var/log/mysql/bin.123456 | mysql -u root -p
```

This command recovers all of the data up until the date and time given by the `--stop-datetime` option. If you did not detect the erroneous SQL statement that was entered until hours later, you will probably also want to recover the activity that occurred afterward. Based on this, you could run `mysqlbinlog` again with a start date and time, like so:

```
shell> mysqlbinlog --start-datetime="2005-04-20 10:01:00" \
         /var/log/mysql/bin.123456 | mysql -u root -p
```

In this command, the SQL statements logged from 10:01 a.m. on will be re-executed. The combination of restoring of the previous night's dump file and the two `mysqlbinlog` commands restores everything up until one second before 10:00 a.m. and everything from 10:01 a.m. on. You should examine the log to be sure of the exact times to specify for the commands. To display the log file contents without executing them, use this command:

```
shell> mysqlbinlog /var/log/mysql/bin.123456 > /tmp/mysql_restore.sql
```

Then open the file with a text editor to examine it.

5.4.2. Specifying Positions for Recovery

Instead of specifying dates and times, the `--start-position` and `--stop-position` options for `mysqlbinlog` can be used for specifying log positions. They work the same as the start and stop date options, except that you specify log position numbers rather than dates. Using positions may enable you to be more precise about which part of the log to recover, especially if many transactions occurred around the same time as a damaging SQL statement. To determine the position numbers, run `mysqlbinlog` for a range of times near the time when the unwanted transaction was executed, but redirect the results to a text file for examination. This can be done like so:

```
shell> mysqlbinlog --start-datetime="2005-04-20 9:55:00" \
         --stop-datetime="2005-04-20 10:05:00" \
         /var/log/mysql/bin.123456 > /tmp/mysql_restore.sql
```

This command creates a small text file in the `/tmp` directory that contains the SQL statements around the time that the deleterious SQL statement was executed. Open this file with a text editor and look for the statement that you don't want to repeat. Determine the positions in the binary log for stopping and resuming the recovery and make note of them. Positions are labeled as `log_pos` followed by a number. After restoring the previous backup file, use the position numbers to process the binary log file. For example, you would use commands something like these:

```
shell> mysqlbinlog --stop-position="368312" /var/log/mysql/bin.123456 \
         | mysql -u root -p
shell> mysqlbinlog --start-position="368315" /var/log/mysql/bin.123456 \
         | mysql -u root -p
```

The first command recovers all the transactions up until the stop position given. The second command recovers all transactions from the starting position given until the end of the binary log. Because the output of `mysqlbinlog` includes `SET TIMESTAMP` statements before each SQL statement recorded, the recovered data and related MySQL logs will reflect the original times at which the transactions were executed.

5.5. Table Maintenance and Crash Recovery

This section discusses how to use `myisamchk` to check or repair `MyISAM` tables (tables that have `.MYD` and `.MYI` files for storing data and indexes). For general `myisamchk` background, see `myisamchk`.

You can use `myisamchk` to get information about your database tables or to check, repair, or optimize them. The following sections describe how to perform these operations and how to set up a table maintenance schedule.

Even though table repair with myisamchk is quite secure, it is always a good idea to make a backup *before* doing a repair or any maintenance operation that could make a lot of changes to a table.

myisamchk operations that affect indexes can cause FULLTEXT indexes to be rebuilt with full-text parameters that are incompatible with the values used by the MySQL server. To avoid this problem, follow the guidelines in myisamchk General Options.

In many cases, you may find it simpler to do MyISAM table maintenance using the SQL statements that perform operations that myisamchk can do:

- To check or repair MyISAM tables, use CHECK TABLE or REPAIR TABLE.
- To optimize MyISAM tables, use OPTIMIZE TABLE.
- To analyze MyISAM tables, use ANALYZE TABLE.

These statements can be used directly or by means of the mysqlcheck client program. One advantage of these statements over myisamchk is that the server does all the work. With myisamchk, you must make sure that the server does not use the tables at the same time so that there is no unwanted interaction between myisamchk and the server. See ANALYZE TABLE Syntax, CHECK TABLE Syntax, OPTIMIZE TABLE Syntax, and REPAIR TABLE Syntax.

5.5.1. Using myisamchk for Crash Recovery

This section describes how to check for and deal with data corruption in MySQL databases. If your tables become corrupted frequently, you should try to find the reason why. See What to Do If MySQL Keeps Crashing.

For an explanation of how MyISAM tables can become corrupted, see MyISAM Table Problems.

If you run mysqld with external locking disabled (which is the default as of MySQL 4.0), you cannot reliably use myisamchk to check a table when mysqld is using the same table. If you can be certain that no one will access the tables through mysqld while you run myisamchk, you only have to execute mysqladmin flush-tables before you start checking the tables. If you cannot guarantee this, you must stop mysqld while you check the tables. If you run myisamchk to check tables that mysqld is updating at the same time, you may get a warning that a table is corrupt even when it is not.

If the server is run with external locking enabled, you can use myisamchk to check tables at any time. In this case, if the server tries to update a table that myisamchk is using, the server will wait for myisamchk to finish before it continues.

If you use myisamchk to repair or optimize tables, you *must* always ensure that the mysqld server is not using the table (this also applies if external locking is disabled). If you do not stop mysqld, you should at least do a mysqladmin flush-tables before you run myisamchk. Your tables *may become corrupted* if the server and myisamchk access the tables simultaneously.

When performing crash recovery, it is important to understand that each MyISAM table *tbl_name* in a database corresponds to the three files in the database directory shown in the following table.

File	Purpose
tbl_name.frm	Definition (format) file
tbl_name.MYD	Data file
tbl_name.MYI	Index file

Each of these three file types is subject to corruption in various ways, but problems occur most often in data files and index files.

myisamchk works by creating a copy of the .MYD data file row by row. It ends the repair stage by removing the old .MYD file and renaming the new file to the original file name. If you use --quick, myisamchk does not create a temporary .MYD file, but instead assumes that the .MYD file is correct and generates only a new index file without touching the .MYD file. This is safe, because myisamchk automatically detects whether the .MYD file is corrupt and aborts the repair if it is. You can also specify the --quick option twice to myisamchk. In this case, myisamchk does not abort on some errors (such as duplicate-key errors) but instead tries to resolve them by modifying the .MYD file. Normally the use of two --quick options is useful only if you have too little free disk space to perform a normal repair. In this case, you should at least make a backup of the table before running myisamchk.

5.5.2. How to Check MyISAM Tables for Errors

To check a MyISAM table, use the following commands:

- myisamchk *tbl_name*

This finds 99.99% of all errors. What it cannot find is corruption that involves *only* the data file (which is very unusual). If you want to check a table, you should normally run `myisamchk` without options or with the `-s` (silent) option.

- `myisamchk -m tbl_name`

  This finds 99.999% of all errors. It first checks all index entries for errors and then reads through all rows. It calculates a checksum for all key values in the rows and verifies that the checksum matches the checksum for the keys in the index tree.

- `myisamchk -e tbl_name`

  This does a complete and thorough check of all data (`-e` means “extended check”). It does a check-read of every key for each row to verify that they indeed point to the correct row. This may take a long time for a large table that has many indexes. Normally, `myisamchk` stops after the first error it finds. If you want to obtain more information, you can add the `-v` (verbose) option. This causes `myisamchk` to keep going, up through a maximum of 20 errors.

- `myisamchk -e -i tbl_name`

  This is like the previous command, but the `-i` option tells `myisamchk` to print additional statistical information.

In most cases, a simple `myisamchk` command with no arguments other than the table name is sufficient to check a table.

5.5.3. How to Repair Tables

The discussion in this section describes how to use `myisamchk` on `MyISAM` tables (extensions `.MYI` and `.MYD`).

You can also (and should, if possible) use the `CHECK TABLE` and `REPAIR TABLE` statements to check and repair `MyISAM` tables. See `CHECK TABLE` Syntax, and `REPAIR TABLE` Syntax.

Symptoms of corrupted tables include queries that abort unexpectedly and observable errors such as these:

- `tbl_name.frm` is locked against change
- Can't find file `tbl_name.MYI` (Errcode: *nnn*)
- Unexpected end of file
- Record file is crashed
- Got error *nnn* from table handler

To get more information about the error, run `perror nnn`, where *nnn* is the error number. The following example shows how to use `perror` to find the meanings for the most common error numbers that indicate a problem with a table:

```
shell> perror 126 127 132 134 135 136 141 144 145
MySQL error code 126 = Index file is crashed
MySQL error code 127 = Record-file is crashed
MySQL error code 132 = Old database file
MySQL error code 134 = Record was already deleted (or record file crashed)
MySQL error code 135 = No more room in record file
MySQL error code 136 = No more room in index file
MySQL error code 141 = Duplicate unique key or constraint on write or update
MySQL error code 144 = Table is crashed and last repair failed
MySQL error code 145 = Table was marked as crashed and should be repaired
```

Note that error 135 (no more room in record file) and error 136 (no more room in index file) are not errors that can be fixed by a simple repair. In this case, you must use `ALTER TABLE` to increase the `MAX_ROWS` and `AVG_ROW_LENGTH` table option values:

```
ALTER TABLE tbl_name MAX_ROWS=xxx AVG_ROW_LENGTH=yyy;
```

If you do not know the current table option values, use `SHOW CREATE TABLE`.

For the other errors, you must repair your tables. `myisamchk` can usually detect and fix most problems that occur.

The repair process involves up to four stages, described here. Before you begin, you should change location to the database directory and check the permissions of the table files. On Unix, make sure that they are readable by the user that `mysqld` runs as (and to you, because you need to access the files you are checking). If it turns out you need to modify files, they must also be writable by you.

This section is for the cases where a table check fails (such as those described in Section 5.5.2, “How to Check `MyISAM` Tables for

Errors"), or you want to use the extended features that `myisamchk` provides.

The options that you can use for table maintenance with `myisamchk` are described in `myisamchk`.

If you are going to repair a table from the command line, you must first stop the `mysqld` server. Note that when you do `mysqladmin shutdown` on a remote server, the `mysqld` server is still alive for a while after `mysqladmin` returns, until all statement-processing has stopped and all index changes have been flushed to disk.

Stage 1: Checking your tables

Run `myisamchk *.MYI` or `myisamchk -e *.MYI` if you have more time. Use the `-s` (silent) option to suppress unnecessary information.

If the `mysqld` server is stopped, you should use the `--update-state` option to tell `myisamchk` to mark the table as "checked."

You have to repair only those tables for which `myisamchk` announces an error. For such tables, proceed to Stage 2.

If you get unexpected errors when checking (such as `out of memory` errors), or if `myisamchk` crashes, go to Stage 3.

Stage 2: Easy safe repair

First, try `myisamchk -r -q tbl_name` (`-r -q` means "quick recovery mode"). This attempts to repair the index file without touching the data file. If the data file contains everything that it should and the delete links point at the correct locations within the data file, this should work, and the table is fixed. Start repairing the next table. Otherwise, use the following procedure:

1. Make a backup of the data file before continuing.
2. Use `myisamchk -r tbl_name` (`-r` means "recovery mode"). This removes incorrect rows and deleted rows from the data file and reconstructs the index file.
3. If the preceding step fails, use `myisamchk --safe-recover tbl_name`. Safe recovery mode uses an old recovery method that handles a few cases that regular recovery mode does not (but is slower).

> **Note**
>
> If you want a repair operation to go much faster, you should set the values of the `sort_buffer_size` and `key_buffer_size` variables each to about 25% of your available memory when running `myisamchk`.

If you get unexpected errors when repairing (such as `out of memory` errors), or if `myisamchk` crashes, go to Stage 3.

Stage 3: Difficult repair

You should reach this stage only if the first 16KB block in the index file is destroyed or contains incorrect information, or if the index file is missing. In this case, it is necessary to create a new index file. Do so as follows:

1. Move the data file to a safe place.
2. Use the table description file to create new (empty) data and index files:

```
shell> mysql db_name
mysql> SET autocommit=1;
mysql> TRUNCATE TABLE tbl_name;
mysql> quit
```

3. Copy the old data file back onto the newly created data file. (Do not just move the old file back onto the new file. You want to retain a copy in case something goes wrong.)

> **Important**
>
> If you are using replication, you should stop it prior to performing the above procedure, since it involves file system operations, and these are not logged by MySQL.

Go back to Stage 2. `myisamchk -r -q` should work. (This should not be an endless loop.)

You can also use the `REPAIR TABLE tbl_name USE_FRM` SQL statement, which performs the whole procedure automatically. There is also no possibility of unwanted interaction between a utility and the server, because the server does all the work when you use `REPAIR TABLE`. See `REPAIR TABLE` Syntax.

Stage 4: Very difficult repair

You should reach this stage only if the `.frm` description file has also crashed. That should never happen, because the description file is not changed after the table is created:

1. Restore the description file from a backup and go back to Stage 3. You can also restore the index file and go back to Stage 2. In the latter case, you should start with `myisamchk -r`.
2. If you do not have a backup but know exactly how the table was created, create a copy of the table in another database. Remove the new data file, and then move the `.frm` description and `.MYI` index files from the other database to your crashed database. This gives you new description and index files, but leaves the `.MYD` data file alone. Go back to Stage 2 and attempt to reconstruct the index file.

5.5.4. Table Optimization

To coalesce fragmented rows and eliminate wasted space that results from deleting or updating rows, run `myisamchk` in recovery mode:

```
shell> myisamchk -r tbl_name
```

You can optimize a table in the same way by using the `OPTIMIZE TABLE` SQL statement. `OPTIMIZE TABLE` does a table repair and a key analysis, and also sorts the index tree so that key lookups are faster. There is also no possibility of unwanted interaction between a utility and the server, because the server does all the work when you use `OPTIMIZE TABLE`. See `OPTIMIZE TABLE` Syntax.

`myisamchk` has a number of other options that you can use to improve the performance of a table:

- `--analyze`, `-a`
- `--sort-index`, `-S`
- `--sort-records=index_num`, `-R index_num`

For a full description of all available options, see `myisamchk`.

5.5.5. Getting Information About a Table

To obtain a description of a table or statistics about it, use the commands shown here. We explain some of the information in more detail later.

- `myisamchk -d tbl_name`

  Runs `myisamchk` in “describe mode” to produce a description of your table. If you start the MySQL server with external locking disabled, `myisamchk` may report an error for a table that is updated while it runs. However, because `myisamchk` does not change the table in describe mode, there is no risk of destroying data.

- `myisamchk -d -v tbl_name`

  Adding `-v` runs `myisamchk` in verbose mode so that it produces more information about what it is doing.

- `myisamchk -eis tbl_name`

  Shows only the most important information from a table. This operation is slow because it must read the entire table.

- `myisamchk -eiv tbl_name`

  This is like `-eis`, but tells you what is being done.

The *`tbl_name`* argument can be either the name of a `MyISAM` table or the name of its index file, as described in `myisamchk`. Multiple *`tbl_name`* arguments can be given.

Sample output for some of these commands follows. They are based on a table with these data and index file sizes:

```
-rw-rw-r--   1 monty    tcx     317235748 Jan 12 17:30 company.MYD
-rw-rw-r--   1 davida   tcx      96482304 Jan 12 18:35 company.MYI
```

Example of `myisamchk -d` output:

```
MyISAM file:     company.MYI
Record format:   Fixed length
Data records:    1403698  Deleted blocks:         0
Recordlength:    226
table description:
Key Start Len Index   Type
1   2     8   unique  double
2   15    10  multip. text packed stripped
3   219   8   multip. double
4   63    10  multip. text packed stripped
5   167   2   multip. unsigned short
6   177   4   multip. unsigned long
7   155   4   multip. text
8   138   4   multip. unsigned long
9   177   4   multip. unsigned long
    193   1           text
```

Example of `myisamchk -d -v` output:

```
MyISAM file:         company
Record format:       Fixed length
File-version:        1
Creation time:       1999-10-30 12:12:51
Recover time:        1999-10-31 19:13:01
Status:              checked
Data records:            1403698  Deleted blocks:              0
Datafile parts:          1403698  Deleted data:                0
Datafile pointer (bytes):      3  Keyfile pointer (bytes):     3
Max datafile length:  3791650815  Max keyfile length: 4294967294
Recordlength:                226
table description:
Key Start Len Index   Type                   Rec/key     Root Blocksize
1   2     8   unique  double                       1 15845376      1024
2   15    10  multip. text packed stripped         2 25062400      1024
3   219   8   multip. double                      73 40907776      1024
4   63    10  multip. text packed stripped         5 48097280      1024
5   167   2   multip. unsigned short            4840 55200768      1024
6   177   4   multip. unsigned long             1346 65145856      1024
7   155   4   multip. text                      4995 75090944      1024
8   138   4   multip. unsigned long               87 85036032      1024
9   177   4   multip. unsigned long              178 96481280      1024
    193   1           text
```

Example of `myisamchk -eis` output:

```
Checking MyISAM file: company
Key:  1:  Keyblocks used:  97%  Packed:    0%  Max levels:  4
Key:  2:  Keyblocks used:  98%  Packed:   50%  Max levels:  4
Key:  3:  Keyblocks used:  97%  Packed:    0%  Max levels:  4
Key:  4:  Keyblocks used:  99%  Packed:   60%  Max levels:  3
Key:  5:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
Key:  6:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
Key:  7:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
Key:  8:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
Key:  9:  Keyblocks used:  98%  Packed:    0%  Max levels:  4
Total:    Keyblocks used:  98%  Packed:   17%
Records:          1403698    M.recordlength:     226
Packed:             0%
Recordspace used:     100%   Empty space:          0%
Blocks/Record:   1.00
Record blocks:    1403698    Delete blocks:        0
Recorddata:     317235748    Deleted data:         0
Lost space:             0    Linkdata:             0
User time 1626.51, System time 232.36
Maximum resident set size 0, Integral resident set size 0
Non physical pagefaults 0, Physical pagefaults 627, Swaps 0
Blocks in 0 out 0, Messages in 0 out 0, Signals 0
Voluntary context switches 639, Involuntary context switches 28966
```

Example of `myisamchk -eiv` output:

```
Checking MyISAM file: company
Data records: 1403698   Deleted blocks:       0
- check file-size
- check delete-chain
block_size 1024:
index  1:
index  2:
index  3:
index  4:
index  5:
index  6:
index  7:
index  8:
index  9:
No recordlinks
```

```
- check index reference
- check data record references index: 1
Key:  1:  Keyblocks used:  97%  Packed:    0%  Max levels:  4
- check data record references index: 2
Key:  2:  Keyblocks used:  98%  Packed:   50%  Max levels:  4
- check data record references index: 3
Key:  3:  Keyblocks used:  97%  Packed:    0%  Max levels:  4
- check data record references index: 4
Key:  4:  Keyblocks used:  99%  Packed:   60%  Max levels:  3
- check data record references index: 5
Key:  5:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
- check data record references index: 6
Key:  6:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
- check data record references index: 7
Key:  7:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
- check data record references index: 8
Key:  8:  Keyblocks used:  99%  Packed:    0%  Max levels:  3
- check data record references index: 9
Key:  9:  Keyblocks used:  98%  Packed:    0%  Max levels:  4
Total:    Keyblocks used:   9%  Packed:   17%
- check records and index references
*** LOTS OF ROW NUMBERS DELETED ***
Records:         1403698   M.recordlength:   226   Packed:           0%
Recordspace used:     100%  Empty space:        0%  Blocks/Record: 1.00
Record blocks:   1403698   Delete blocks:      0
Recorddata:    317235748   Deleted data:       0
Lost space:            0   Linkdata:           0
User time 1639.63, System time 251.61
Maximum resident set size 0, Integral resident set size 0
Non physical pagefaults 0, Physical pagefaults 10580, Swaps 0
Blocks in 4 out 0, Messages in 0 out 0, Signals 0
Voluntary context switches 10604, Involuntary context switches 122798
```

Explanations for the types of information `myisamchk` produces are given here. “Keyfile” refers to the index file. “Record” and “row” are synonymous.

- `MyISAM file`

  Name of the `MyISAM` (index) file.

- `File-version`

  Version of `MyISAM` format. Currently always 2.

- `Creation time`

  When the data file was created.

- `Recover time`

  When the index/data file was last reconstructed.

- `Data records`

  How many rows are in the table.

- `Deleted blocks`

  How many deleted blocks still have reserved space. You can optimize your table to minimize this space. See Section 5.5.4, “Table Optimization”.

- `Datafile parts`

  For dynamic-row format, this indicates how many data blocks there are. For an optimized table without fragmented rows, this is the same as `Data records`.

- `Deleted data`

  How many bytes of unreclaimed deleted data there are. You can optimize your table to minimize this space. See Section 5.5.4, “Table Optimization”.

- `Datafile pointer`

  The size of the data file pointer, in bytes. It is usually 2, 3, 4, or 5 bytes. Most tables manage with 2 bytes, but this cannot be controlled from MySQL yet. For fixed tables, this is a row address. For dynamic tables, this is a byte address.

- `Keyfile pointer`

The size of the index file pointer, in bytes. It is usually 1, 2, or 3 bytes. Most tables manage with 2 bytes, but this is calculated automatically by MySQL. It is always a block address.

- `Max datafile length`

  How long the table data file can become, in bytes.

- `Max keyfile length`

  How long the table index file can become, in bytes.

- `Recordlength`

  How much space each row takes, in bytes.

- `Record format`

  The format used to store table rows. The preceding examples use `Fixed length`. Other possible values are `Compressed` and `Packed`.

- `table description`

  A list of all keys in the table. For each key, `myisamchk` displays some low-level information:

  - `Key`

    This key's number.

  - `Start`

    Where in the row this portion of the index starts.

  - `Len`

    How long this portion of the index is. For packed numbers, this should always be the full length of the column. For strings, it may be shorter than the full length of the indexed column, because you can index a prefix of a string column.

  - `Index`

    Whether a key value can exist multiple times in the index. Possible values are `unique` or `multip.` (multiple).

  - `Type`

    What data type this portion of the index has. This is a `MyISAM` data type with the possible values `packed`, `stripped`, or `empty`.

  - `Root`

    Address of the root index block.

  - `Blocksize`

    The size of each index block. By default this is 1024, but the value may be changed at compile time when MySQL is built from source.

  - `Rec/key`

    This is a statistical value used by the optimizer. It tells how many rows there are per value for this index. A unique index always has a value of 1. This may be updated after a table is loaded (or greatly changed) with `myisamchk -a`. If this is not updated at all, a default value of 30 is given.

  For the table shown in the examples, there are two `table description` lines for the ninth index. This indicates that it is a multiple-part index with two parts.

- `Keyblocks used`

  What percentage of the keyblocks are used. When a table has just been reorganized with `myisamchk`, as for the table in the examples, the values are very high (very near theoretical maximum).

- `Packed`

MySQL tries to pack key values that have a common suffix. This can only be used for indexes on `CHAR` and `VARCHAR` columns. For long indexed strings that have similar leftmost parts, this can significantly reduce the space used. In the third of the preceding examples, the fourth key is 10 characters long and a 60% reduction in space is achieved.

- `Max levels`

  How deep the B-tree for this key is. Large tables with long key values get high values.

- `Records`

  How many rows are in the table.

- `M.recordlength`

  The average row length. This is the exact row length for tables with fixed-length rows, because all rows have the same length.

- `Packed`

  MySQL strips spaces from the end of strings. The `Packed` value indicates the percentage of savings achieved by doing this.

- `Recordspace used`

  What percentage of the data file is used.

- `Empty space`

  What percentage of the data file is unused.

- `Blocks/Record`

  Average number of blocks per row (that is, how many links a fragmented row is composed of). This is always 1.0 for fixed-format tables. This value should stay as close to 1.0 as possible. If it gets too large, you can reorganize the table. See Section 5.5.4, “Table Optimization”.

- `Recordblocks`

  How many blocks (links) are used. For fixed-format tables, this is the same as the number of rows.

- `Deleteblocks`

  How many blocks (links) are deleted.

- `Recorddata`

  How many bytes in the data file are used.

- `Deleted data`

  How many bytes in the data file are deleted (unused).

- `Lost space`

  If a row is updated to a shorter length, some space is lost. This is the sum of all such losses, in bytes.

- `Linkdata`

  When the dynamic table format is used, row fragments are linked with pointers (4 to 7 bytes each). `Linkdata` is the sum of the amount of storage used by all such pointers.

If a table has been compressed with `myisampack`, `myisamchk -d` prints additional information about each table column. See `myisampack`, for an example of this information and a description of what it means.

5.5.6. Setting Up a Table Maintenance Schedule

It is a good idea to perform table checks on a regular basis rather than waiting for problems to occur. One way to check and repair `MyISAM` tables is with the `CHECK TABLE` and `REPAIR TABLE` statements. See `CHECK TABLE` Syntax, and `REPAIR TABLE` Syntax.

Another way to check tables is to use `myisamchk`. For maintenance purposes, you can use `myisamchk -s`. The `-s` option

(short for `--silent`) causes `myisamchk` to run in silent mode, printing messages only when errors occur.

It is also a good idea to enable automatic `MyISAM` table checking. For example, whenever the machine has done a restart in the middle of an update, you usually need to check each table that could have been affected before it is used further. (These are "expected crashed tables.") To check `MyISAM` tables automatically, start the server with the `--myisam-recover` option. See Server Command Options.

You should also check your tables regularly during normal system operation. For example, you can run a `cron` job to check important tables once a week, using a line like this in a `crontab` file:

```
35 0 * * 0 /path/to/myisamchk --fast --silent /path/to/datadir/*/*.MYI
```

This prints out information about crashed tables so that you can examine and repair them as necessary.

We recommend that to start with, you execute `myisamchk -s` each night on all tables that have been updated during the last 24 hours. As you see that problems occur infrequently, you can back off the checking frequency to once a week or so.

Normally, MySQL tables need little maintenance. If you are performing many updates to `MyISAM` tables with dynamic-sized rows (tables with `VARCHAR`, `BLOB`, or `TEXT` columns) or have tables with many deleted rows you may want to defragment/reclaim space from the tables from time to time. You can do this by using `OPTIMIZE TABLE` on the tables in question. Alternatively, if you can stop the `mysqld` server for a while, change location into the data directory and use this command while the server is stopped:

```
shell> myisamchk -r -s --sort-index --sort_buffer_size=16M */*.MYI
```

Chapter 6. Frequently Asked Questions about Security

Questions

Questions and Answers

6.1: Does MySQL 6.0 have built-in authentication against LDAP directories?

No. Support for external authentication methods is on the MySQL roadmap as a “rolling feature”, which means that we plan to implement it in the future, but we have not yet determined when this will be done.

6.2: Does MySQL 6.0 include support for Roles Based Access Control (RBAC)?

No. Support for roles is on the MySQL roadmap as a “rolling feature”, which means that we plan to implement it in the future, but we have not yet determined when this will be done.

6.3: Is SSL support be built into MySQL binaries, or must I recompile the binary myself to enable it?

Most 6.0 binaries have SSL enabled for client-server connections that are secured, authenticated, or both. However, the YaSSL library currently does not compile on all platforms. See Section 4.7, “Using SSL for Secure Connections”, for a complete listing of supported and unsupported platforms.

6.4: Does MySQL 6.0 have native support for SSL?

Most 6.0 binaries have support for SSL connections between the client and server. We can't currently build with the new YaSSL library everywhere, as it's still quite new and does not compile on all platforms yet. See Section 4.7, “Using SSL for Secure Connections”.

You can also tunnel a connection via SSH, if (for instance) if the client application doesn't support SSL connections. For an example, see Section 4.8, “Connecting to MySQL Remotely from Windows with SSH”.

6.5: Where can I find documentation that addresses security issues for MySQL?

The best place to start is Chapter 1, *General Security Issues*.

Other portions of the MySQL Documentation which you may find useful with regard to specific security concerns include the following:

- Section 1.1, “General Security Guidelines”.
- Section 1.2, “Making MySQL Secure Against Attackers”.
- How to Reset the Root Password.
- Section 1.5, “How to Run MySQL as a Normal User”.
- User-Defined Function Security Precautions.
- Section 1.3, “Security-Related `mysqld` Options”.
- Section 1.4, “Security Issues with `LOAD DATA LOCAL`”.
- Chapter 2, *Post-Installation Setup and Testing*.
- SELinux Notes.
- Section 4.7.1, “Basic SSL Concepts”.

MySQL Enterprise
The MySQL Enterprise Monitor enforces best practices for maximizing the security of your servers. For more information see http://www.mysql.com/products/enterprise/advisors.html.