DBA 2: Administering MySQL (PDF)

Introduction
DBA 2: Administering MySQL Lesson 1

How to Learn using O'Reilly Learning Courses

Welcome to the second course in the O'Reilly DBA Series!

In this course you will learn how to setup MySQL, create and maintain databases, add users, and manage permissions. We'll start by installing MySQL, setup a database and add users, then move on to more advanced topics such as backups and indexes.

This course assumes you have worked through the first course in the series, or are familiar with the fundamentals of MySQL. If you need a refresher, feel free to go back over the first course.

To learn a new skill or technology, you must sit down and experiment with the technology. The more experimentation you do, the more you learn. Our learning system is designed to maximize your experimentation and help you **learn how to learn** the technology. Here are some tips for learning:

- **Learn in your own voice** - Work through your own ideas and listen to yourself in order to learn the new skill. We want you to facilitate your own learning, so we avoid lengthy video or audio streaming, and keep spurious animated demonstrations to a minimum.
- **Take your time** - Learning takes time. Rushing through can have negative effects on your progress. By taking your time, you'll try new things and learn more.
- **Create your own examples and demonstrations** - In order to understand a complex concept, you need to understand its various parts. We'll help you do that in this course, offering guidance as you create a demonstration, piece by piece.
- **Experiment with your ideas and questions** - You're encouraged to wander from the path often to explore possibilities! We can't possibly anticipate all of your questions and ideas, so it's up to you to experiment and create on your own.
- **Accept guidance, but don't depend on it** - Try to overcome difficulties on your own. Going from misunderstanding to understanding on your own is the best way to learn any new skill. Our goal is for you to use the technology independent of us. Of course, you can always contact your mentor when necessary.
- **Create REAL projects** - Real projects are more meaningful and rewarding to complete than simulated projects. They'll help you to understand what's involved in real world situations. After each lesson you'll be given objectives or quizzes so you can test your new knowledge.
- **Have fun!** - Relax, keep practicing, and don't be afraid to make mistakes! There are no deadlines in this course, and your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied *I'm so cool! I did it!* feeling. And when you're finished, you'll have some *really cool* projects to show off when you're done.

Understanding the Learning Sandbox Environment

We'll be doing lots of work in the **Learning Sandbox** learning environment, so let's go over how it works. Below is the **CodeRunner editor**, which will serve as your workspace for experimenting with code we give you, as well as your own ideas. CodeRunner is a multi-purpose editor that allows you to create applications in many technologies.

As you can see, all of the communication tools and learning content go in the upper part of the window. Frequently we will ask you to type code into the CodeRunner Editor below and experiment by making changes. When we want you to type code on your own, you will see a white box like the one below, with code in it.

Type the following into CodeRunner:

```
We'll ask you to type code that occurs in white boxes like this into CodeRunner below.

Every time you see a white box, it's your cue to experiment.
```

Similarly, when we want you to simply observe some code or result, we will put it in a gray box like this:

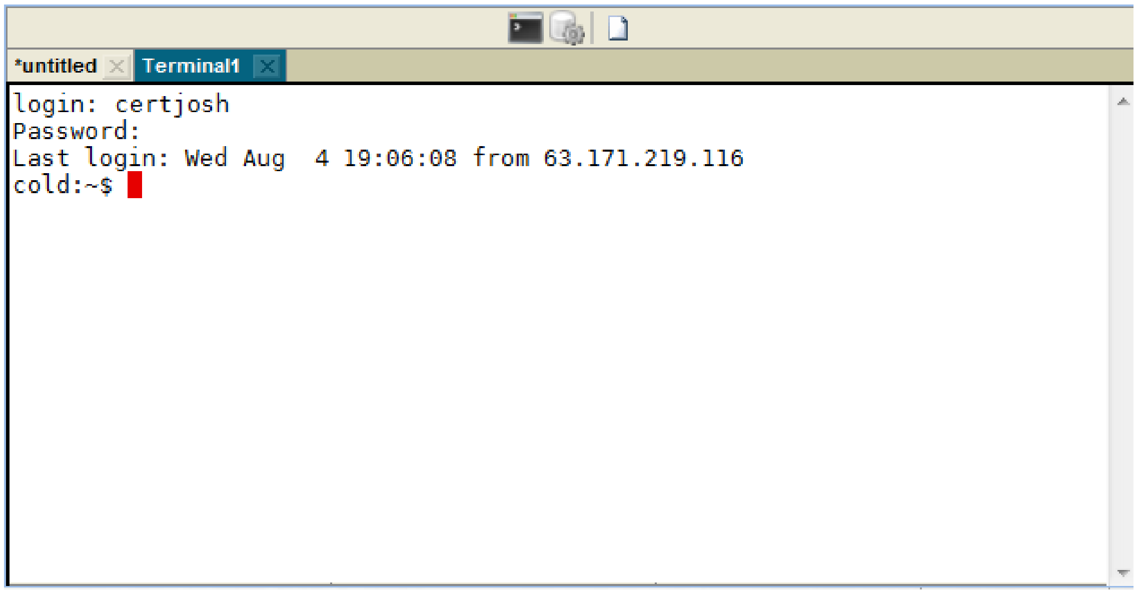
Type the following into the editor:

```
Gray boxes will be used for observing code.

You are not expected to type anything from these "example" boxes.
```

Since CodeRunner is a multi-purpose editor, you'll need to have it set up for the language or environment you want to use. In this course we'll be using the Unix Terminal, working both in Unix and to access the MySQL server that we will be setting up in this lesson. (You may notice that there is also a **Connect to MySQL** button but that doesn't go to the MySQL server we will be setting up so will not be used during this course.)

Try clicking the **New Terminal** button to connect to the Unix Terminal now. You will see prompts for the login and password you were given when you registered with OST, although the system may fill one or both of these in for you. If the system doesn't automatically log you in, type your username and password when prompted. (You may need to click inside the Unix Terminal window to be able to type. When typing your password, you will not see any characters reflected back to you as you type.) You will then be logged in to one of the OST servers.

Throughout this course, whenever you see a reference to certjosh, replace it with your own username.



```
login: certjosh
Password:
Last login: Wed Aug  4 19:06:08 from 63.171.219.116
cold:~$
```

Once you are logged in you'll see the UNIX prompt **cold:~$**. Now we can get ready to connect to MySQL!

Getting Started with MySQL

MySQL is one of the most popular open source SQL databases available. It is typically used in a *LAMP* solution stack of technologies. A *solution stack* is a set of software that is needed to deliver a completely functional solution. The "stack" in *LAMP* is:

- **L**inux
- **A**pache
- **M**ySQL
- **P**HP

It is a fast, reliable, and easy to use relation database management system (RDBMS). It is free to use, provides good performance, and can be used for many types of development projects.

Since you are the database administrator for your company, you don't have to worry about the application development being done. You're job is to provide a reliable, working, and well performing to developers and end users. As such we won't concern ourselves with application level details in this course; instead we'll stick to the workings of the database itself.

Note If you wish to know more about application development, check out the other course offerings at oreillyschool.com.

For this course we are not concerned with **what** we put into our database, only **how** we create and maintain databases. To save us the trouble of creating a whole database structure with data for this course, we'll be using the sample database called *sakila* that is provided by MySQL. Sakila was created to demonstrate many of MySQL's features and functionality.

This course uses version 0.8 of the sakila database, available at MySQL's web site.

Planning for a MySQL Install

Today's computers are very fast, and disk space is plenty cheap. For companies with ample IT budgets, it is tempting to purchase a new server for each new application under development. While this certainly works, this actually has several drawbacks:

- Power: electricity and cooling are large expenses, and are probably won't become cheaper in the future
- Space: servers and disk arrays take up space, and space can be very valuable
- Manageability: maintaining several servers and networks can be difficult, especially when important security updates must be applied quickly
- Distance: keeping data on separate servers isn't always a good idea - it can make cross-database analysis difficult

On the other hand, there are some very good reasons for keeping data on separate servers:

- Security: maintaining high levels of security may be easier with separate physical servers
- Cost savings: if you have old servers that are working fine, you may not see cost savings if you combine applications on one server (in fact you might break something!)
- Locality: data connections can be very fast, but they won't be as fast as a server on the local network

Sizing a database server is something of a mystical art. Nobody can accurately predict the future, so who can tell if the hardware you purchase today will be adequately sized next year?

Instead of guessing how big your server should be, or purchasing the largest server you can possibly purchase, take an inventory of your application. The easiest way to accomplish this is to have MySQL tell you the size of your tables and database!

Installing MySQL

You'll need access to your own private copy of MySQL for this course. Your learning account does includes access to a MySQL database on a shared server, however you do not have administrative rights on that MySQL instance or even shell access to the database server. This means you cannot create your own databases, start and stop the database server, or reinstall the server if you wish.

In practice it is a great idea to have your own copy of MySQL where you can try new things and practice your skills without interrupting production systems.

Tip Your development doesn't necessarily have to be the same exact version of MySQL, or even the same operating system, but it is good practice for these to be almost exactly the same.

Remember - if you have any questions or problems, don't hesitate to contact your mentor!

Since we're logged into the Unix Terminal, let's grab MySQL! You can download the source code to MySQL and compile it yourself if you wish. More than likely you'll simply grab a compiled package or binary from MySQL's web site. For this course we'll use a special version available from O'Reilly's servers.

Type the RED text at the Unix prompt:

```
cold login: certjosh
Password:
Last login: Wed Aug  4 19:06:08 from 63.171.219.116
cold1:~$ wget "http://courses.oreillyschool.com/dba2/downloads/mysql.tar.gz"
```

Note This course is focused on database administration. If you are also interested in Unix/Linux systems administration, check out O'Reilly's excellent Linux administration series!

If you typed in everything correctly you'll see the following:

OBSERVE:

```
cold1:~$ wget "http://courses.oreillyschool.com/dba2/downloads/mysql.tar.gz"
--2012-01-03 16:49:41--  http://courses.oreillyschool.com/dba2/downloads/mysql.tar.gz
Resolving courses.oreillyschool.com... 199.27.144.89
Connecting to courses.oreillyschool.com|199.27.144.89|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 670358 (655K) [application/x-gzip]
Saving to: mysql.tar.gz

100%[==========================================================================>] 670,358     -

2012-01-03 16:49:41 (55.4 MB/s) - mysql.tar.gz

cold1:~$
```

Now, unpack the tar archive.

Type the RED text at the Unix prompt:

```
cold:~$ tar xfz mysql.tar.gz
```

Unless there was an error, you won't see anything except the Unix prompt.

OBSERVE:

```
cold:~$ tar xfz mysql.tar.gz
cold:~$
```

Switch your current directory by using the **cd** command to **mysql** and do a directory listing using the **ls** command.

Type the RED AND GREEN text at the Unix prompt:

```
cold:~$ cd mysql
cold:~/mysql$ ls
```

You'll see the following files:

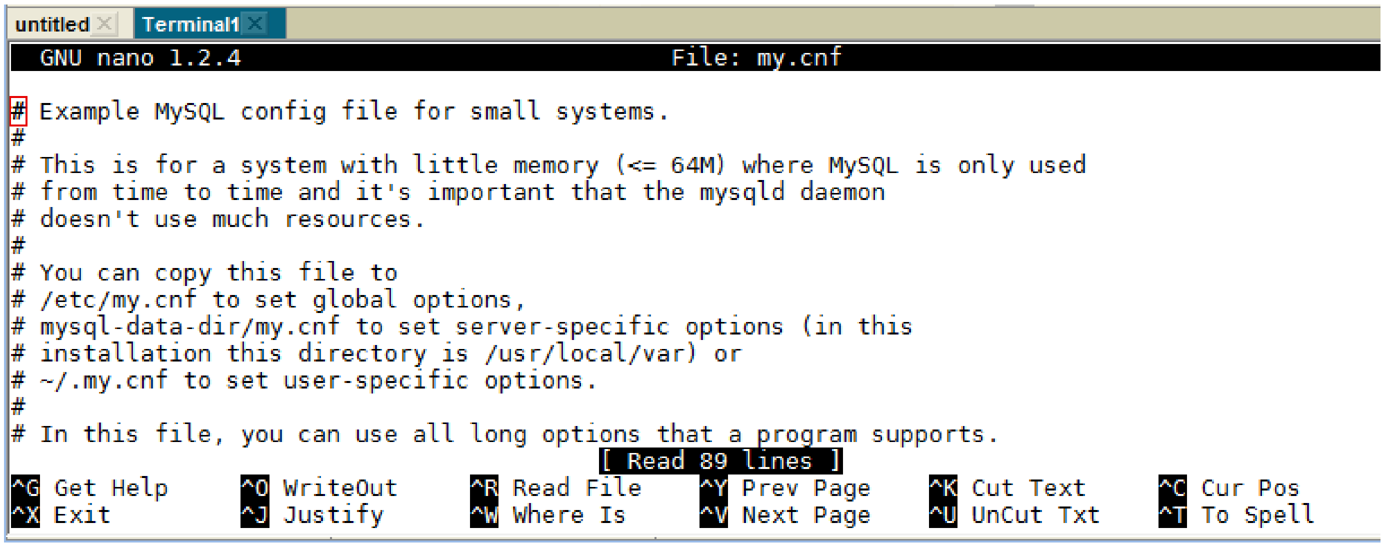
OBSERVE:

```
cold:~$ cd mysql
cold:~/mysql$ ls
bin@             lib@             sakila-data.sql    var/
data/            libexec@         sakila-schema.sql
include@         my.cnf           share@
cold:~/mysql$
```

Most MySQL programs can read configuration settings from the **my.cnf** options file. There are many different settings stored in this file - some only apply to client programs, some only apply to the database server itself.

We need to make some small changes that file to reflect our specific installation. We need to change the location of the **socket** that is used to connect to the MySQL server. The server "listens" on this socket file for incoming connections from programs. You can think of this as a sort of postal worker, watching a mailbox for incoming letters.

Feel free to use whatever editor you are comfortable with - the Sandbox, Pico, vi or Emacs. (If you are not familiar with any of these editors, please contact your mentor for assistance.) If you use Pico the file will look something like the following:



```
  GNU nano 1.2.4                          File: my.cnf

# Example MySQL config file for small systems.
#
# This is for a system with little memory (<= 64M) where MySQL is only used
# from time to time and it's important that the mysqld daemon
# doesn't use much resources.
#
# You can copy this file to
# /etc/my.cnf to set global options,
# mysql-data-dir/my.cnf to set server-specific options (in this
# installation this directory is /usr/local/var) or
# ~/.my.cnf to set user-specific options.
#
# In this file, you can use all long options that a program supports.
                                          [ Read 89 lines ]
^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Txt     ^T To Spell
```

Find the line that looks like this:

OBSERVE:

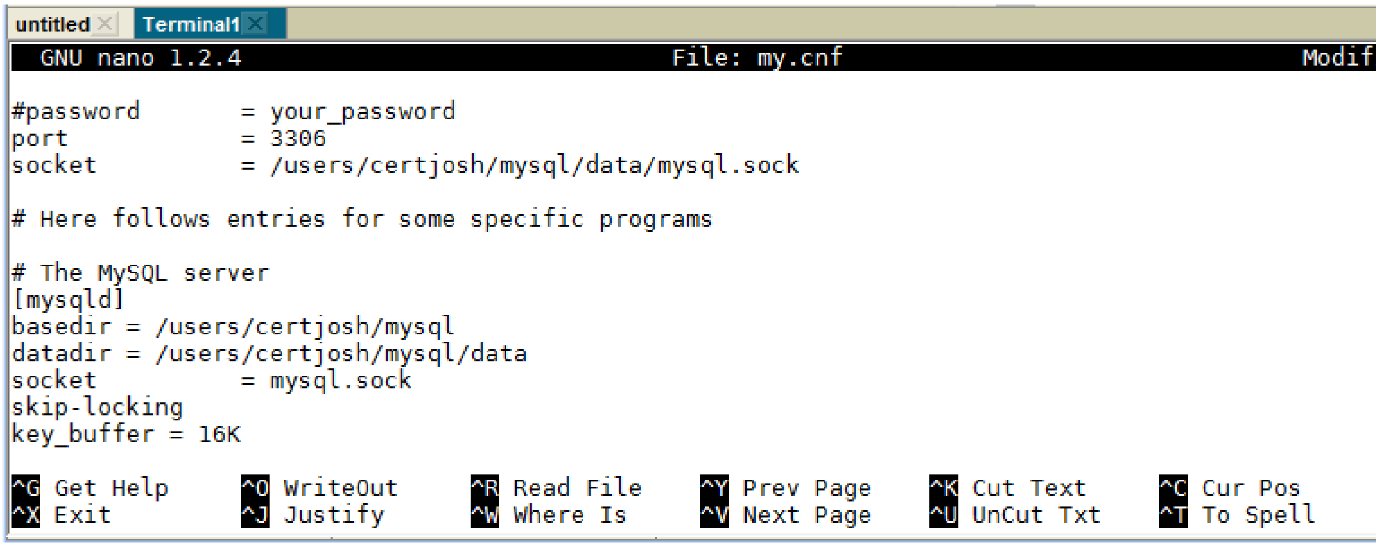
```
socket          = mysql.sock
```

And change it so that it contains the specific location of your file. Remember to replace the **red** text with your username:

OBSERVE:

```
socket          = /users/certjosh/mysql/data/mysql.sock
```

When you are done your file will look something like this:



```
  GNU nano 1.2.4                        File: my.cnf                        Modif

#password       = your_password
port            = 3306
socket          = /users/certjosh/mysql/data/mysql.sock

# Here follows entries for some specific programs

# The MySQL server
[mysqld]
basedir = /users/certjosh/mysql
datadir = /users/certjosh/mysql/data
socket          = mysql.sock
skip-locking
key_buffer = 16K

^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Txt  ^T To Spell
```

Be sure to save your changes!

Next, Find the line that looks like this:

OBSERVE:

```
[mysqld]
socket          = mysql.sock
```

Add two new lines, replacing the **red** text with your username:

OBSERVE:

```
basedir = /users/username/mysql
datadir = /users/username/mysql/data
```

When you are done, that part of the file will look like:

OBSERVE:

```
[mysqld]
basedir = /users/certjosh/mysql
datadir = /users/certjosh/mysql/data
socket          = mysql.sock
```

Be sure to save those changes as well!

Next, we'll link your account's default **.my.cnf** preference file with the my.cnf file you just modified. We do this because your account's **.my.cnf** file is the first file read by the `mysql` program. We'll back up your .my.cnf file first using the Linux move command, just in case.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ mv ~/.my.cnf ~/.my.cnf.saved
```

If you don't have a .my.cnf file, you'll get a message like `mv: /users/certjosh/.my.cnf: No such file or directory`. It is safe to ignore this message.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ ln -s ~/mysql/my.cnf ~/.my.cnf
```

You won't see any response unless you mistyped this command.

We are almost done! The next step is to setup MySQL's system tables. Make sure you replace the **GREEN certjosh** with your username.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ bin/mysql_install_db --defaults-file=/users/certjosh/mysql/my.cnf
```

If everything goes OK you'll see the following messages:

OBSERVE:

```
Installing MySQL system tables...
OK
Filling help tables...
OK

To start mysqld at boot time you have to copy
support-files/mysql.server to the right place for your system

PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:
/users/certjosh/mysql/bin/mysqladmin -u root password 'new-password'
/users/certjosh/mysql/bin/mysqladmin -u root -h cold password 'new-password'
See the manual for more instructions.
You can start the MySQL daemon with:
cd /usr/local ; /users/certjosh/mysql/bin/mysqld_safe &

You can test the MySQL daemon with mysql-test-run.pl
cd mysql-test ; perl mysql-test-run.pl

Please report any problems with the /usr/local/bin/mysqlbug script!

The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at http://shop.mysql.com
```

The message given by the mysql_install_db program is a little misleading. Because of the way the OST servers are set up, our actual command to start MySQL is different. Let's start it now! Make sure you replace the **GREEN certjosh** with your username.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ cd ~/mysql ; bin/mysqld_safe --defaults-file=/users/certjosh/mysql/my.cnf &
```

This time you'll see some output. The message will look something similar to:

OBSERVE:

```
cold:~/mysql$ cd ~/mysql ; bin/mysqld_safe --defaults-file=/users/certjosh/mysql/my.cnf &
[1] 5095
cold:~/mysql$ Starting mysqld daemon with databases from /users/certjosh/mysql/data
cold:~/mysql$
```

Congratulations! You've installed and started MySQL!

Note If you don't see the `cold:~/mysql$` prompt, press enter a few times.

Let's make sure it is working. We can do so by changing the "root" password from nothing to something useful. **Change it to be the same password you use to log into your learning account.** Type in the following command, **but enter your OST password instead of the words *your_ost_password*.**

Type the RED text at the Unix prompt:

```
cold:~/mysql$ mysqladmin -u root password your_ost_password
```

Note If you see a message like `ERROR 2002: Can't connect to local MySQL server through socket '/users/certjosh/mysql/data/mysql.sock' (2)` then your MySQL server is not running.

Again, you won't see any messages unless there was an error. Now try logging in.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ mysql -u root -p
```

Type in your password carefully when prompted. If everything is OK you'll be connected to MySQL!

OBSERVE:

```
cold:~/mysql$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 5.0.41-OREILLY

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Shutdown

Now that we have verified everything is working correctly, let's shutdown MySQL. You don't turn off your computer by unplugging the cord from the wall, and you don't just kill the MySQL process. Instead you use the `mysqladmin` program.

Type the RED text at the Unix prompt:

```
mysql> exit
Bye
cold:~/mysql$ mysqladmin -u root -p shutdown
```

After entering your password, you should see something like the following:

OBSERVE:

```
Enter password:
STOPPING server from pid file /users/certjosh/mysql/data/cold.pid
070604 21:46:55  mysqld ended

[1]+  Done                    bin/mysqld_safe --defaults-file=/users/certjosh/mysql/my.cnf
```

Note If you log out and back into your account you won't see the `[1]+ Done` message.

Keep the startup and shutdown commands handy. While you can leave your MySQL server running if you are not logged in, there is a chance it won't be running when you log back in. It is best to shutdown the server if you are not going to be logged in for a while. If you are running MySQL on your own machine it is OK to leave the server running at all times.

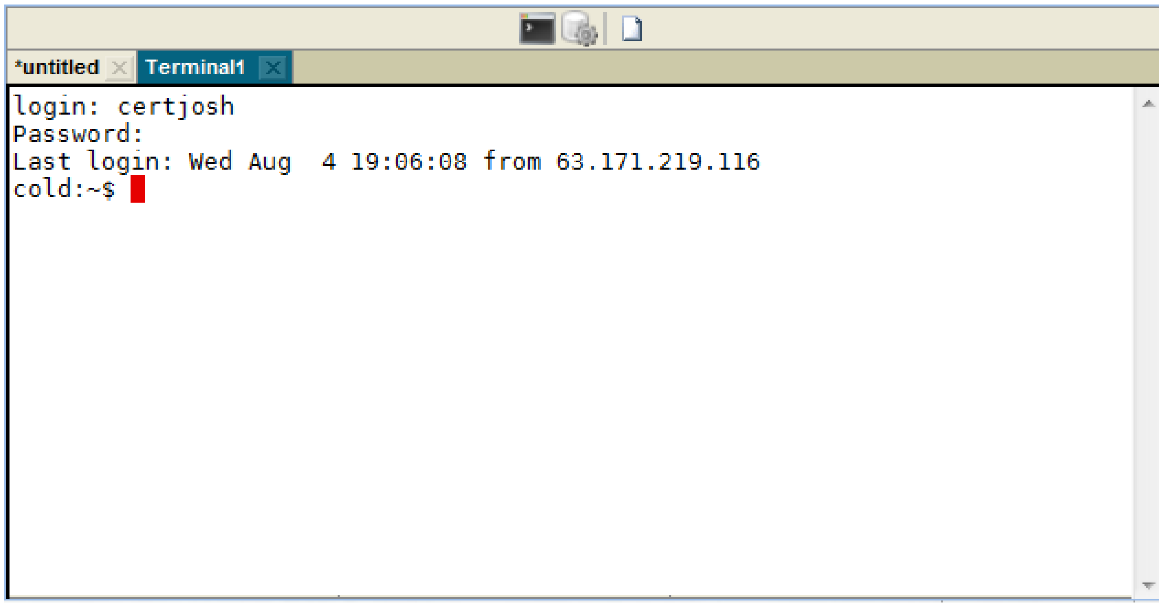
Congratulations! You've done a lot in this lesson. In the next lesson you'll learn how to use your newly installed MySQL server to estimate server capacity. See you then!

Estimating Database Capacity
DBA 2: Administering MySQL Lesson 2

Estimating Database Capacity

Welcome back!

Before you get started, let's make sure your MySQL server is running. Click on the **New Terminal** button to connect to the Unix Terminal now. Remember, always replace references to **certjosh** with your own username.



```
login: certjosh
Password:
Last login: Wed Aug  4 19:06:08 from 63.171.219.116
cold:~$
```

Now let's check to see if MySQL is running.

Type the RED text at the Unix prompt:

```
cold login: certjosh
Password:
Last login: Wed Aug  4 19:06:08 from 63.171.219.116
cold:~$ ps x | grep mysqld
```

If you shutdown MySQL in the previous lesson, you'll see something like the following:

It should look something like this:

```
cold:~$ ps x | grep mysqld
14831 ttyp6    S      0:00 grep mysqld
cold:~$
```

If you left MySQL running, you'll see a much different result:

OBSERVE:

```
cold:~$ ps x | grep mysqld
16333 ttyp6    S      0:00 /bin/sh bin/mysqld_safe --basedir=/users/certjosh/mys
16373 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16375 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16377 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16378 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16379 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16380 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16381 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16382 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16383 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16384 ttyp6    SN     0:00 /users/certjosh/mysql/libexec/mysqld --basedir=/users
16389 ttyp6    S      0:00 grep mysqld
```

```
cold:~$
```

If you need to start it, follow these commands. Make sure you replace the **GREEN certjosh** with your username.

Type the RED text at the Unix prompt:

```
cold:~$ cd ~/mysql ; bin/mysqld_safe --defaults-file=/users/certjosh/mysql/my.cnf &
```

Database Size

In the last lesson we installed a database server. Since that is out of the way, we can use the server to get a good estimate on our future database size. For this we'll use MySQL's sample database called *sakila*. The sakila database represents a DVD rental store and utilizes the most interesting features of MySQL.

You may have noticed two files in the directory listing in the last lesson - **sakila-data.sql** and **sakila-schema.sql**. The **schema** is the definition of all objects (tables, procedures, views, etc.) in the sakila database, and the **data** file contains sample data for every table in the database. Database developers should be able to provide you, the database administrator, with two similar files for every database project at your company. The data file doesn't have to be your complete data set - instead it should be a subset of your data.

Ideally you would be given data that is split across several files (perhaps one file for each table). With multiple files you could load each file individually, measuring disk usage after each load. Each load would tell you how much space load would require.

If you were a backpacker, you would want to know how much gear you could carry on a hike. You might determine this by weighing your backpack:

1. completely empty - like a new database server
2. with empty storage containers in it - like an empty (schema only) database
3. full - like a database loaded with sample data

Before we create our database and populate it with data, let's find out the minimum disk usage required by our current MySQL install (the completely empty backpack). We can determine this information by using the `INFORMATION_SCHEMA.TABLES` view. If you need a refresher on `INFORMATION_SCHEMA`, feel free to review DBA 1 Lesson 12.

Log into your mysql as root.

Type the following at the MySQL prompt:

```
mysql> SELECT
SUM(data_length+index_length)/1048576 AS Total,
SUM(data_length)/1048576 AS Data_MB,
SUM(index_length)/1048576 AS Index_MB
FROM information_schema.tables;
```

The results of your query may be different, but should look something like the following:

OBSERVE:

```
mysql> SELECT
    -> SUM(data_length+index_length)/1048576 AS Total,
    -> SUM(data_length)/1048576 AS Data_MB,
    -> SUM(index_length)/1048576 AS Index_MB
    -> FROM INFORMATION_SCHEMA.TABLES;
+------------+------------+------------+
| Total      | Data_MB    | Index_MB   |
+------------+------------+------------+
| 0.5223     | 0.4530     | 0.0693     |
+------------+------------+------------+
1 row in set (0.08 sec)

mysql>
```

This query uses `INFORMATION_SCHEMA.TABLES` to add the data and index usage for all tables. `data_length` is in bytes, so we divide it by 1048576 (1024 times 1024) to convert it to megabytes. The result shows that our empty MySQL server uses around .5 MB (megabytes).

Note It is fine if your values differ slightly from those listed above.

Now, let's create our database! First, we'll create a schema only database so that we can take another look at disk usage. The file **sakila-schema.sql** contains all of the commands necessary to create our database and all tables within the database. We'll use the mysql command to execute the commands in the file.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ mysql -u root -p < sakila-schema.sql
```

You will be asked to enter your password. Keep in mind that this command won't return anything unless there is an error.

Note If you see an error such as:

```
mysql: error while loading shared libraries: libmysqlclient.so.15: cannot open shared
object file: No such file or directory
```

or:

```
ERROR 1064 at line 182: You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'DELIMITER' at
line 1
```

go back to the first lesson and make sure you repeat the installation steps necessary for mysql.

Now that the database has been created, let's check the size of the new sakila database.

Type the following at the MySQL prompt:

```
mysql> SELECT
SUM(data_length+index_length)/1048576 AS Total,
SUM(data_length)/1048576 AS Data_MB,
SUM(index_length)/1048576 AS Index_MB
FROM information_schema.tables
WHERE table_schema='sakila';
```

Looks like sakila adds around .6 MB to the database server.

OBSERVE:

```
mysql> SELECT
    -> SUM(data_length+index_length)/1048576 AS Total,
    -> SUM(data_length)/1048576 AS Data_MB,
    -> SUM(index_length)/1048576 AS Index_MB
    -> FROM information_schema.tables
    -> WHERE table_schema='sakila';
+--------+---------+----------+
| Total  | Data_MB | Index_MB |
+--------+---------+----------+
| 0.6104 |  0.2344 |   0.3760 |
+--------+---------+----------+
1 row in set (0.04 sec)

mysql>
```

Now that we've examined a schema-only database, let's load some data! We'll use the mysql command, just like we did when we created the database.

Note This script may take some time to run - it has a lot of work to do.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ mysql -u root -p < sakila-data.sql
```

Just like before, you'll be asked to enter your password, but you won't see any messages unless there was an error.

Now that the database has been created, let's check the size of the new sakila database with data. We can use the same query as before.

Type the following at the MySQL prompt:

```
mysql> SELECT
SUM(data_length+index_length)/1048576 AS Total,
SUM(data_length)/1048576 AS Data_MB,
SUM(index_length)/1048576 AS Index_MB
FROM information_schema.tables
WHERE table_schema='sakila';
```

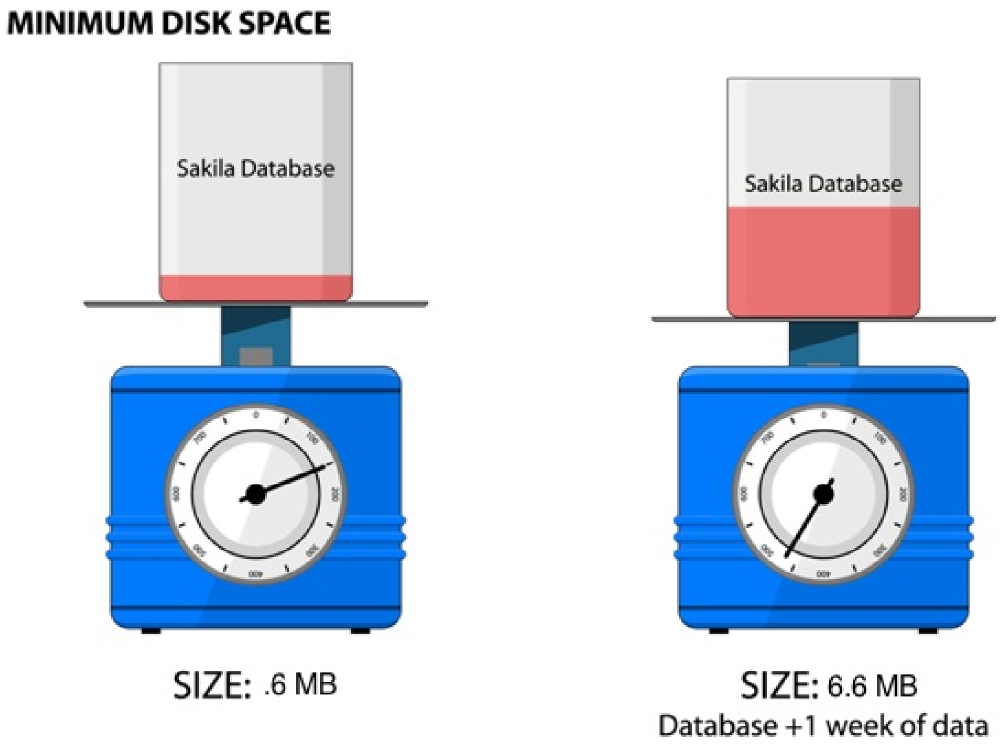
The database is now 6.6 MB in size, an increase in 6 MB!

OBSERVE:

```
mysql> SELECT
    -> SUM(data_length+index_length)/1048576 AS Total,
    -> SUM(data_length)/1048576 AS Data_MB,
    -> SUM(index_length)/1048576 AS Index_MB
    -> FROM information_schema.tables
    -> WHERE table_schema='sakila';
+--------+---------+----------+
| Total  | Data_MB | Index_MB |
+--------+---------+----------+
| 6.6229 |  4.0984 |   2.5244 |
+--------+---------+----------+
1 row in set (0.07 sec)

mysql>
```

Now, suppose our developers believe that this sample data file represents one week of transactions. With this in mind, we are ready to do some estimations.

Subtraction shows that our database increased by **6 MB** when we loaded the data. Since the sample data file represents one week's worth of transactions, we could estimate our database would grow by 24 MB a month. We'll round that up to 30 MB a month to give us a little cushion. Multiplying this by 12 will give us a year's worth of data at 360 MB.



Note These data sizes are small in comparison to most database workloads, however the estimation concepts are the same for GB of data as they are for MB of data.

This isn't the only way to estimate database size, but it is a good way to get a general idea of how much disk space you're machine will need in the next few years.

Data Files

MySQL stores all database objects in the `data` directory. Let's take a look at this directory to see what it contains. First, make sure you're in the **mysql** directory.

Type the RED text at the Unix prompt:

```
cold:~$ cd ~/mysql/
```

Next we'll use the **du** (disk usage) command to get a current summary of disk space in the data directory.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ du -h data/*
```

You're results may vary, however you should see something similar to this (without the color):

OBSERVE:

```
8.0K    data/cold1.useractive.com.err
0       data/cold1.useractive.com.pid
4.0K    data/cold.err
0       data/cold.pid
19M     data/ibdata1
5.1M    data/ib_logfile0
5.1M    data/ib_logfile1
836K    data/mysql
0       data/mysql.sock
576K    data/sakila
4.0K    data/test
```

MySQL creates a directory for each database in the system. This listing shows three databases: **mysql**, **sakila** and **test**.

For InnoDB tables, data is stored in the **ibdata1** file and **ib_logfile** files. Since InnoDB is a transaction-safe database engine, your INSERT, UPDATE and DELETE queries are recorded in a **log** file before being committed to the **data** file. This extra step ensures full ACID-compliance. For a refresher on ACID, take a look at DBA 1 Lesson 5.

Let's take a peek inside a database directory to see what is inside.

Type the RED text at the Unix prompt:

```
cold:~/mysql$ du -h data/mysql/*
```

Your results should look something like the following:

OBSERVE:

```
12K     data/mysql/columns_priv.frm
8.0K    data/mysql/columns_priv.MYD
8.0K    data/mysql/columns_priv.MYI
12K     data/mysql/db.frm
4.0K    data/mysql/db.MYD
4.0K    data/mysql/db.MYI
12K     data/mysql/func.frm
0       data/mysql/func.MYD
4.0K    data/mysql/func.MYI
12K     data/mysql/help_category.frm
24K     data/mysql/help_category.MYD
4.0K    data/mysql/help_category.MYI
12K     data/mysql/help_keyword.frm
80K     data/mysql/help_keyword.MYD
16K     data/mysql/help_keyword.MYI
12K     data/mysql/help_relation.frm
8.0K    data/mysql/help_relation.MYD
16K     data/mysql/help_relation.MYI
12K     data/mysql/help_topic.frm
332K    data/mysql/help_topic.MYD
20K     data/mysql/help_topic.MYI
12K     data/mysql/host.frm
0       data/mysql/host.MYD
4.0K    data/mysql/host.MYI
12K     data/mysql/proc.frm
8.0K    data/mysql/proc.MYD
4.0K    data/mysql/proc.MYI
12K     data/mysql/procs_priv.frm
0       data/mysql/procs_priv.MYD
4.0K    data/mysql/procs_priv.MYI
12K     data/mysql/tables_priv.frm
40K     data/mysql/tables_priv.MYD
```

```
8.0K    data/mysql/tables_priv.MYI
12K     data/mysql/time_zone.frm
12K     data/mysql/time_zone_leap_second.frm
0       data/mysql/time_zone_leap_second.MYD
4.0K    data/mysql/time_zone_leap_second.MYI
0       data/mysql/time_zone.MYD
4.0K    data/mysql/time_zone.MYI
12K     data/mysql/time_zone_name.frm
0       data/mysql/time_zone_name.MYD
4.0K    data/mysql/time_zone_name.MYI
12K     data/mysql/time_zone_transition.frm
0       data/mysql/time_zone_transition.MYD
4.0K    data/mysql/time_zone_transition.MYI
12K     data/mysql/time_zone_transition_type.frm
0       data/mysql/time_zone_transition_type.MYD
4.0K    data/mysql/time_zone_transition_type.MYI
12K     data/mysql/user.frm
4.0K    data/mysql/user.MYD
4.0K    data/mysql/user.MYI
```

There are three files for each table in the database. You might be asking yourself - "what *are* all of these files?" Check out the table below.

Extension	**Contents**
frm	Data dictionary. Contains the definition of a table.
MYD	Data file. This file holds all data for a table.
MYI	Index file. This file contains the index structures for a table.
opt	Stores database characteristics, such as the character set.
TRG	Triggers file. Contains a list of triggers for a database table.
TRN	Trigger file. Each database trigger is stored in its own file.

MySQL creates some files for the sakila database, even though the sakila database is InnoDB. Take a look:

Type the RED text at the Unix prompt:

```
cold:~/mysql$ du -h data/sakila/*
```

There are few MYD files in this listing since the tables are nearly all InnoDB.

OBSERVE:

```
8.5K    data/sakila/actor.frm
2.0K    data/sakila/actor_info.frm
8.6K    data/sakila/address.frm
8.4K    data/sakila/category.frm
8.5K    data/sakila/city.frm
8.4K    data/sakila/country.frm
8.7K    data/sakila/customer.frm
1.2K    data/sakila/customer_list.frm
65      data/sakila/db.opt
36      data/sakila/del_film.TRN
882     data/sakila/film.TRG
9.0K    data/sakila/film.frm
8.4K    data/sakila/film_actor.frm
8.5K    data/sakila/film_category.frm
1.6K    data/sakila/film_list.frm
0       data/sakila/film_text.MYD
1.0K    data/sakila/film_text.MYI
8.4K    data/sakila/film_text.frm
36      data/sakila/ins_film.TRN
8.5K    data/sakila/inventory.frm
8.4K    data/sakila/language.frm
2.0K    data/sakila/nicer_but_slower_film_list.frm
8.6K    data/sakila/payment.frm
8.6K    data/sakila/rental.frm
1.1K    data/sakila/sales_by_film_category.frm
1.5K    data/sakila/sales_by_store.frm
8.7K    data/sakila/staff.frm
1.1K    data/sakila/staff_list.frm
8.5K    data/sakila/store.frm
```

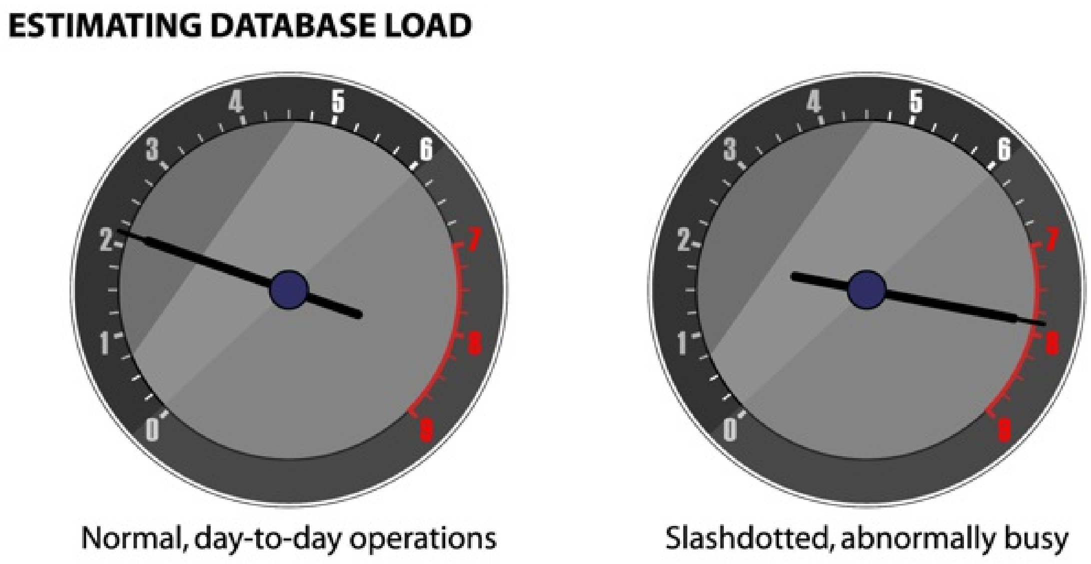
```
36      data/sakila/upd_film.TRN
```

Estimating Database Load

Estimating database load can be much more difficult than estimating disk usage. To a certain extent you may not be able to begin estimation until your developers have created a prototype or demo application. Many systems allow users to perform ad-hoc queries for data, which also complicates analysis.

Your business may offer clues as to how busy the database could be at various times during the day and week. Our sakila sample database is for a DVD rental store. More than likely Friday and Saturday nights are the busiest times for the store. For one store that might mean 100 customers an hour, for another it might mean 250 customers an hour.

Some businesses are very busy at quarter end; some are busiest for the two hours proceeding the stock market close. Some internet sites are not busy until a story appears on Slashdot!



Construct a survey to help you estimate database load. Some issues to consider are as follows:

1. How many users will access the application?
2. What types of applications will use the database? Web applications? Desktop applications? Batch jobs?
3. How much data needs to be stored? Is it megabytes, or gigabytes?
4. How much history needs to be kept? All? Three years?
5. What types of queries will be run? Simple transactional queries, or complex analysis queries? Both?
6. Will users have the ability to query the database using SQL or reporting tools?

The answers to these questions will help you get a picture as to what size (and what quantity) of database servers you'll need for your application. There isn't necessarily a correct answer to the question "is our database server large enough," but you'll absolutely know the wrong answers. Hopefully you'll pick something between too small and too large!

Congratulations! You've done a lot of work in this lesson. You created a new database, populated it with some sample data, and measured how much space the new database used. You surveyed your business users to determine application load, and now you're ready to make a server recommendation!

In the next lesson you'll dive into administration of users, groups and roles to make sure your database is secure and reliable. See you then!

Users and Permissions
DBA 2: Administering MySQL Lesson 3

Planning for Users

Welcome back! In the last lesson we set up our test database and populated it with some data. In this lesson we'll take the next step - set up our roles, groups, and users in order to manage and secure access to our database.

You might be wondering - why do we need to create users and log into the database server? Why can't we just *use* the database?

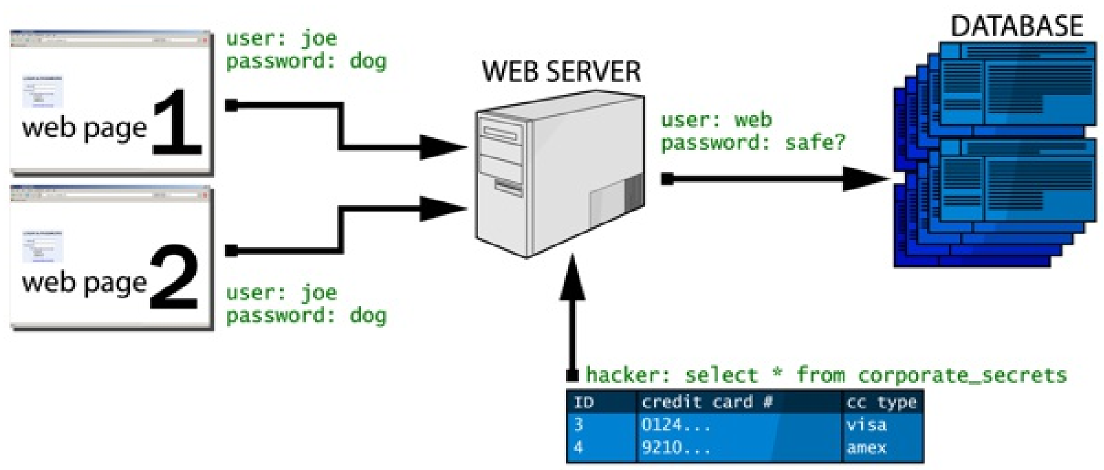
Answer: we log into the database in order to control access to its resources. If nobody logged into the server, we would have no control over its use. How could we keep people from deleting records from the database, or from looking at sensitive salary information?

If we really wanted to avoid creating users, we could create one database user, set up permissions for that user, then share the password with everyone (or even make the password empty)! This *will* work, however one day you'll regret the day you decided to take this short cut!

There are serious drawbacks to having one username and password:

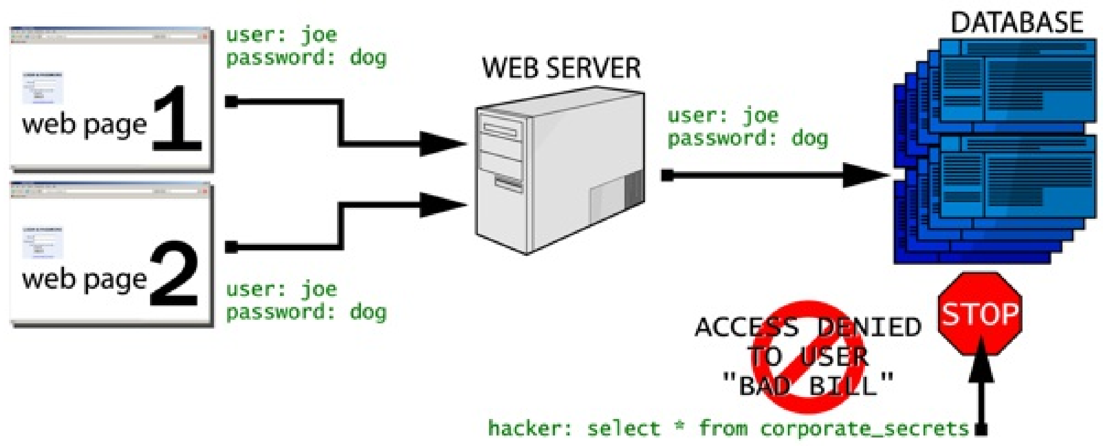
1. Blank passwords can be a huge security risk.
2. Shared passwords must be reset when personnel leave the company, or when a security breach occurs.
3. It can be nearly impossible to track down "bad" users who do nasty things to the database.
4. Database security must be enforced in applications, instead of the database. This is redundant programming.

Instead of having one username and password, we are going to create a separate username and password for everyone who will access the database. We'll also create a user (or *service account*) for every anonymous web site, batch job, or application in use at the company. This will help isolate potential security problems in case the web site is hacked or a bug is discovered in an application.



ID	credit card #	cc type
3	0124...	visa
4	9210...	amex

OR



Suppose you want to make a web application that accesses a database. There are two ways you could handle data security:

1. Create one database user, a few database tables, and application code to enforce security.

2. Let the database handle security

The argument most web developers say with solution #1 is **"I don't want users logging directly into the database!"** As a result, great effort is expended to create and maintain permissions for the web application.

Why go through the trouble? As it turns out most of this code **is already written** - right in the database! What better place to enforce security than right next to the data! This has large benefits:

- Security is implemented uniformly, for all applications that access the database, automatically
- User accounts are tightly locked down, so no single user will ever have more permissions than he or she needs

This setup allows developers to focus on code that is important to your business instead of reinventing yet another wheel.

Before we go and try to create user accounts for everyone who needs access to the database, we need to do some initial planning. Otherwise our multiple users setup will quickly become cumbersome to manage. We need to split our users into *user groups* or *roles*.

Many databases allow administrators to create *user groups* such as **Sales** or **Marketing**, then apply permissions and security to those groups instead of individual users. In addition, databases allow administrators to create *roles* such as **Sales Associate** or **Marketing Manager** and apply permissions to those roles. In an ideal world, users, groups and/or roles would mirror corporate structure.

Groups and roles are different words for the same thing - they allow permissions to be managed for many users. Some databases such as SQL Server and Oracle only have roles, whereas older versions of PostgreSQL (such as 7.3) only have groups.

Unfortunately for us, MySQL doesn't have groups or roles. Fortunately we can do some things to simulate roles and use SQL to make our lives easier. Thinking in terms of groups and roles makes the transition to other databases much easier.

The sakila database we created in the last lesson represents a DVD rental store. For now let's assume this simple store has three types of employees:

- Owners - can see everything, can do everything
- Managers - can add new DVDs to the store inventory, but cannot add new staff
- Clerks - can view DVDs, add and update customers, but cannot view sales data

Let's take a look at the tables in the sakila database we created in the last lesson, and think of how those tables might be used by the roles we've defined. Permissions are determined by the application specification created by developers and users. Suppose your developers and users have defined the following access permissions in the following permissions table:

Table	Roles	Permissions
actor	Owner, manager	Full
	Clerk	Select
actor_info *(a view)*	*all*	Select
address	Owner	Full
	Manager, clerk	Select, Update, Insert
category	Owner	Full permission
	Manager, clerk	Select
city	Owner	Full permission
	Manager, clerk	Select
country	Owner	Full permission
	Manager, clerk	Select
customer	Owner	Full
	Manager, clerk	Select, Update, Insert
customer_list *(a view)*	*all*	Select
film	Owner, manager	Full
	Clerk	Select
film_actor	Owner, manager	Full
	Clerk	Select
film_category	Owner, manager	Full
	Clerk	Select
film_list *(a view)*	*all*	Select
film_text	Owner, manager	Full
	Clerk	Select

inventory	Owner, manager	Full
	Clerk	Select
language	Owner	Full permission
	Manager, clerk	Select
nicer_but_slower_film_list *(a view)*	*all*	Select
payment	Owner	Full
	Manager	Select, Update, Insert
	Clerk	Select, Insert
rental	Owner	Full
	Manager	Select, Update, Insert
	Clerk	Select, Insert
sales_by_film_category *(a view)*	Owner	Full
	Manager, Clerk	No access
sales_by_store *(a view)*	Owner	Full
	Manager, Clerk	No access
staff	Owner	Full permission
	Manager, clerk	Select
staff_list *(a view)*	*all*	Select
store	Owner	Full permission
	Manager, clerk	Select

Let's take a closer look at one of the tables:

payment	Owner	Full
	Manager	Select, Update, Insert
	Clerk	Select, Insert

The owner has full access to everything. The manager cannot delete rows, but can update rows. Clerks can only select and insert rows. In the real world, clerks can accept payments and see past payments, but only managers can make updates to past payments. The manager cannot delete payments. The owner can do anything - including deleting payments. More than likely the owner wouldn't delete rows unless it was policy not to keep records past a certain time.

Let's look at another table:

staff	Owner	Full permission
	Manager, clerk	Select

Perhaps the owner is the only person responsible for hiring staff and maintaining staff records. For this reason only the owner has full access to staff. Others can select from the table, so managers and clerks can look up coworkers' phone numbers, for example.

Adding and Removing Users

Now that we've decided on our roles and permissions, let's add some users! Since MySQL doesn't have roles, we'll add three new dummy user accounts as model accounts for each role. To make sure the account isn't actually used, we'll set the password to some random characters.

Adding users to MySQL is fairly straightforward. However there is one unique aspect to MySQL: user access is defined not only by a *username*, but also by the *host* computer from which the user is connecting. This means that a user called **dave** connecting across a network to the database server could have different permissions than the user called **dave** connecting to the database server, from the database server itself.

The syntax for this is usually as follows:

OBSERVE:

```
'username' @ 'host'
```

Both username and host name must be in quotes. Hosts can be specified using either an IP address like `10.0.2.102` or DNS name like `dvd.biz`. The special character `%` is used to denote *any* - and can be used to specify all hosts, or only part of the host. This is useful if you wish to include both `www.biz.dvd` as well as `accounting.biz.dvd` as acceptable hosts. Below is a breakdown of some other possible user & host combinations using the special % wild card. Keep in mind that if you omit the host, MySQL assumes you don't want to limit security by host.

User @ Host	Notes
'dave' @ 'accounting.dvd.biz'	Only the machine `accounting.dvd.biz`
'dave' @ '%.dvd.biz'	Any machine in the domain `dvd.biz` - like `www.biz.dvd` *and* `accounting.biz.dvd`
'dave' @ '%'	Any machine
'dave'	Any machine

Note Normally you'll lock down all user accounts to specific machines, or domains. The special version of MySQL we're using in this course is only accessible from the same physical machine it is installed on, so it is OK to limit access to `localhost`.

Let's get started by adding a test user called dave. We'll let dave connect from localhost. Make sure you are logged into the Unix Terminal, and MySQL is running. Be sure to connect to your MySQL database server as the root user.

Type the following at the MySQL prompt:

```
mysql> create user 'dave'@'localhost' identified by 'asdf1249ljkasdf';
```

If you typed everything correctly you'll see the standard result:

OBSERVE:

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

As you can see, we used the **create user** statement to add a user called **dave**. We set **dave**'s password using the **identified by** keyword, and specified the password to be **asdf1249ljkasdf**.

Removing a user is similar to removing a table or database - we'll use the **drop** keyword.

Type the following at the MySQL prompt:

```
mysql> drop user 'dave'@'localhost';
```

If you typed everything correctly you'll see the standard result:

OBSERVE:

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

Note You might encounter an error like `ERROR 1396: Operation DROP USER failed for 'dave'@'localhost'`. This means your database server doesn't have a user called 'dave' who has access from `localhost`.

Since we know how to add users, let's add three new accounts called **owner**, **manager** and **clerk**. For now we'll set the passwords to something meaningful so we can test the changes we make to the permissions.

Type the following at the MySQL prompt:

```
mysql> create user 'owner'@'localhost' identified by 'dvdStore';
mysql> create user 'manager'@'localhost' identified by 'dvdStore';
mysql> create user 'clerk'@'localhost' identified by 'dvdStore';
```

If you type everything correctly you'll see the standard MySQL result for each line - `Query OK, 0 rows affected (0.01 sec)`.

Before we log out, let's use the `show databases;` command to display all databases.

Type the following at the MySQL prompt:

```
mysql> show databases;
```

If you typed everything correctly you'll see the following:

OBSERVE:

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| classicmodels      |
| mysql              |
| sakila             |
| test               |
+--------------------+
5 rows in set (0.07 sec)

mysql>
```

Let's see if we have permission to use the sakila database.

Type the following at the MySQL prompt:

```
mysql> use sakila;
```

If you typed everything correctly, you'll see:

OBSERVE:

```
mysql> use sakila;
Database changed
mysql>
```

This is the expected result since we are currently logged in as root. As such, we should have access to all databases.

To which databases do you suppose **owner** has access? Let's find out! Log out of MySQL, then log in again, this time as **owner**. Keep the preceding list of databases in mind as you log in as **owner**.

Type the RED text at the Unix prompt:

```
mysql> exit
Bye
cold:~/mysql$ mysql -u owner -p
```

If you typed the password correctly, you'll see something similar to:

OBSERVE:

```
cold:~$ mysql/bin/mysql -u owner -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 54
Server version: 5.0.41-OREILLY Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Let's see the available databases.

Type the following at the MySQL prompt:

```
mysql> show databases;
```

Interesting! The sakila database is nowhere to be seen!

OBSERVE:

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| test               |
+--------------------+
2 rows in set (0.03 sec)
```

Try using sakila, just to see what happens.

Type the following at the MySQL prompt:

```
mysql> use sakila;
```

We've only given the user **owner** access to the database server. Since we didn't grant access to the sakila database, or any of its objects, **owner** isn't able to do to many things.

OBSERVE:

```
mysql> use sakila;
ERROR 1044 (42000): Access denied for user 'owner'@'localhost' to database 'sakila'
mysql>
```

Setting up and using permissions

Permissions in MySQL can be granted at the global, database, table, column, and routine levels, and there are many types of permissions that can be set on a user@host basis. For a full description of these privileges, check out [MySQL's web site](#).

For now we'll stick to the DELETE, INSERT, SELECT and UPDATE privileges. Let's focus on the actor table, granting access to the **owner** user first. To do this we will have to be logged into MySQL as **root.**

Note If you are uncertain about which user you are logged in as, try running the command:
mysql> select current_user();

Let's allow the **owner** full control over the **actor** table, since this is what we specified in the permissions table examined earlier in the lesson.

Type the following at the MySQL prompt:

```
mysql> grant delete,insert,select,update on sakila.actor to 'owner'@'localhost';
```

If everything went OK you'll see the standard message: `Query OK, 0 rows affected (0.41 sec).`

We're on our way! We used the **grant** statement to give **delete, insert, select, and update** permissions to the user called 'owner.' Those rights are limited to the **actor** table in the **sakila** database.

Let's check those permissions. Log out of MySQL, then log back in as **owner**.

Type the following at the MySQL prompt:

```
mysql> show databases;
```

This looks better - **owner** now has access to sakila.

OBSERVE:

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| sakila             |
| test               |
+--------------------+
3 rows in set (0.06 sec)
```

Let's dive a bit deeper, just to be sure. This time we'll run a command to see which **tables** within the **sakila** database the owner has access to.

Type the following at the MySQL prompt:

```
mysql> use sakila; show tables;
```

Sure enough, **owner** can only see one table.

OBSERVE:

```
mysql> use sakila; show tables;
Database changed
+------------------+
| Tables_in_sakila |
+------------------+
| actor            |
+------------------+
1 row in set (0.00 sec)
```

Let's do one last check! If the owner indeed has access to the **actor** table, then this user should be able to successfully SELECT from the table.

Type the following at the MySQL prompt:

```
mysql> select * from actor limit 0, 5;
```

Looks good!

OBSERVE:

```
mysql> select * from actor limit 0, 5;
+----------+------------+--------------+---------------------+
| actor_id | first_name | last_name    | last_update         |
+----------+------------+--------------+---------------------+
|        1 | PENELOPE   | GUINESS      | 2006-02-15 04:34:33 |
|        2 | NICK       | WAHLBERG     | 2006-02-15 04:34:33 |
|        3 | ED         | CHASE        | 2006-02-15 04:34:33 |
|        4 | JENNIFER   | DAVIS        | 2006-02-15 04:34:33 |
|        5 | JOHNNY     | LOLLOBRIGIDA | 2006-02-15 04:34:33 |
+----------+------------+--------------+---------------------+
5 rows in set (0.00 sec)
```

Looks good!

If we consult the permissions table from earlier in the lesson, we'll see managers and clerks are not allowed to insert, update or delete rows from the actors table. In order to set this up, we'll first grant them delete, insert, select and update, then **REVOKE** delete, insert and update.

You usually don't GRANT, then **REVOKE** permissions like this; We're doing it in this case to demonstrate the use of **REVOKE**. Usually you only GRANT the specific permissions a user needs.

Log out of MySQL, then log back in as **root**. Let's look at the 'manager' first.

Type the following at the MySQL prompt:

```
mysql> grant delete,insert,select,update on sakila.actor to 'manager'@'localhost';
```

If you get an error when you tried the previous step, check your typing and try again. Now we'll **REVOKE** the delete, insert, and update permissions for 'manager.'

Type the following at the MySQL prompt:

```
mysql> revoke delete,insert,update on sakila.actor from 'manager'@'localhost';
```

As usual, you should see something like `Query OK, 0 rows affected (0.01 sec)` if your command was successful.

Log out of MySQL, then log back in, this time as **manager**, and try running a select statement on the **actor** table.

Type the following at the MySQL prompt:

```
mysql> use sakila; select * from actor limit 0,5;
```

Note that **manager** indeed has select permission on the actors table.

OBSERVE:

```
mysql> use sakila; select * from actor limit 0,5;
Database changed
+----------+------------+--------------+---------------------+
| actor_id | first_name | last_name    | last_update         |
+----------+------------+--------------+---------------------+
|        1 | PENELOPE   | GUINESS      | 2006-02-15 04:34:33 |
|        2 | NICK       | WAHLBERG     | 2006-02-15 04:34:33 |
|        3 | ED         | CHASE        | 2006-02-15 04:34:33 |
|        4 | JENNIFER   | DAVIS        | 2006-02-15 04:34:33 |
|        5 | JOHNNY     | LOLLOBRIGIDA | 2006-02-15 04:34:33 |
+----------+------------+--------------+---------------------+
5 rows in set (0.00 sec)

mysql>
```

Can **manager** delete rows from actors? Let's find out.

Type the following at the MySQL prompt:

```
mysql> delete from actor where actor_id=1;
```

Doesn't look like **manager** can do much!

OBSERVE:

```
mysql> delete from actor where actor_id=1;
ERROR 1142 (42000): DELETE command denied to user 'manager'@'localhost' for table 'actor'
mysql>
```

This is great news - no matter how hard a manager tries, actors cannot be updated, deleted, or even added to the system.

Copying permissions to new users

MySQL stores all user permissions in its own internal database called **mysql**. Let's take a look at the mysql database. Connect to MySQL as **root**.

Type the following at the MySQL prompt:

```
mysql> use mysql; show tables;
```

There are a lot of tables, some for functionality, some for time zone information, some for privileges.

OBSERVE:

```
mysql> use mysql; show tables;
Database changed
+---------------------------+
| Tables_in_mysql           |
+---------------------------+
| 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                      |
+---------------------------+
17 rows in set (0.00 sec)
```

Instead of using the GRANT keyword to set permissions, we can modify MySQL's privilege tables directly. We'll do this to copy privileges from a "model" user (like owner) to any new users we create.

For this example, we will make **dave** a **manager**. Before we can create **dave** by copying **manager** permissions, we will need to use GRANT to set up permissions for the manager role.

Type the following at the MySQL prompt:

```
grant delete,insert,select,update on sakila.actor to 'manager'@'localhost';
grant select, update, insert on sakila.actor_info  to 'manager'@'localhost';
grant select, update, insert on sakila.address to 'manager'@'localhost';
grant select on sakila.category to 'manager'@'localhost';
grant select on sakila.city to 'manager'@'localhost';
grant select on sakila.country to 'manager'@'localhost';
grant select, update, insert on sakila.customer to 'manager'@'localhost';
grant select on sakila.customer_list to 'manager'@'localhost';
grant select, update, insert, delete on sakila.film to 'manager'@'localhost';
grant select, update, insert, delete on sakila.film_actor to 'manager'@'localhost';
grant select, update, insert, delete on sakila.film_category to 'manager'@'localhost';
grant select on sakila.film_list to 'manager'@'localhost';
grant select, update, insert, delete on sakila.film_text to 'manager'@'localhost';
grant select, update, insert, delete on sakila.inventory to 'manager'@'localhost';
grant select on sakila.language to 'manager'@'localhost';
grant select on sakila.nicer_but_slower_film_list to 'manager'@'localhost';
grant select, update, insert on sakila.payment to 'manager'@'localhost';
grant select, update, insert on sakila.rental to 'manager'@'localhost';
grant select on sakila.staff to 'manager'@'localhost';
grant select on sakila.staff_list to 'manager'@'localhost';
grant select on sakila.store to 'manager'@'localhost';
```

MySQL stores table-related permissions in its internal table called `tables_priv`. Our previous `grant` statements created several rows in this table. Let's take a look at the contents of this table, for the user `manager`.

Type the following at the MySQL prompt:

```
mysql> select * from tables_priv where user='manager' and db='sakila';
```

If you typed everything correctly, you'll see the following:

OBSERVE:

```
mysql> select * from tables_priv where user='manager' and db='sakila';
+-----------+--------+---------+----------------------------+----------------+---------------------+--
| Host      | Db     | User    | Table_name                 | Grantor        | Timestamp           | T
+-----------+--------+---------+----------------------------+----------------+---------------------+--
| localhost | sakila | manager | actor                      | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | actor_info                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | address                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | category                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | city                       | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | country                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | customer                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | customer_list              | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | film                       | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | film_actor                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | film_category              | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | film_list                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | film_text                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | inventory                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | language                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | nicer_but_slower_film_list | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | payment                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | rental                     | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | staff                      | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | staff_list                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | manager | store                      | root@localhost | 2009-01-09 13:44:57 | S
+-----------+--------+---------+----------------------------+----------------+---------------------+--
21 rows in set (0.31 sec)
```

```
mysql>
```

We can see the changes we made to the permissions for the **manager** user and the **sakila** database.

Note Ideally we wouldn't have to touch a database's internal tables, since the structure of the tables tends to change from time to time. Since MySQL doesn't have roles, we'll have to touch the tables directly unless we want to spend a lot of time working on permissions.

Let's take a closer look at this table.

Type the following at the MySQL prompt:

```
mysql> explain tables_priv;
```

If you typed everything correctly, you'll see the following:

OBSERVE:

```
mysql> explain tables_priv;
+-------------+----------------------------------------------------------------------------------
| Field       | Type
+-------------+----------------------------------------------------------------------------------
| Host        | char(60)
| Db          | char(64)
| User        | char(16)
| Table_name  | char(64)
| Grantor     | char(77)
| Timestamp   | timestamp
| Table_priv  | set('Select','Insert','Update','Delete','Create','Drop','Grant','References','Index','
| Column_priv | set('Select','Insert','Update','References')
+-------------+----------------------------------------------------------------------------------
8 rows in set (0.00 sec)

mysql>
```

We'll select from this table where `User='manager'`, and insert new rows back into `tables_priv` for our new manager, username **dave**. We still need to use `create user` before we do anything else.

Type the following at the MySQL prompt:

```
mysql> create user 'dave'@'localhost' identified by 'dvdStore';
```

Let's set up dave's manager permissions. We can use an insert and select query for this task.

Type the following at the MySQL prompt:

```
mysql> insert into tables_priv
(Host, Db, User, Table_name, Grantor, Table_priv, Column_priv)
 select Host, Db, 'dave' as User, Table_name, 'root@localhost' as Grantor, Table_priv, Column_priv
 from tables_priv
 WHERE user='manager';
```

If you typed everything correctly, you'll see the following message:

OBSERVE:

```
Query OK, 21 rows affected (0.02 sec)
Records: 21  Duplicates: 0  Warnings: 0
```

Let's double check the rows we just inserted.

Type the following at the MySQL prompt:

```
mysql> select * from tables_priv where user IN ('manager','dave') and db='sakila' order by 4;
```

If you typed everything correctly, you'll see the following:

OBSERVE:

```
mysql> select * from tables_priv where user IN ('manager','dave') and db='sakila' order by 4;
+-----------+--------+---------+----------------------------+----------------+---------------------+--
| Host      | Db     | User    | Table_name                 | Grantor        | Timestamp           | T
+-----------+--------+---------+----------------------------+----------------+---------------------+--
| localhost | sakila | manager | actor                      | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | actor                      | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | actor_info                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | actor_info                 | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | address                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | address                    | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | category                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | category                   | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | city                       | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | city                       | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | country                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | country                    | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | customer                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | customer                   | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | customer_list              | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | customer_list              | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | film                       | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | film                       | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | film_actor                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | film_actor                 | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | film_category              | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | film_category              | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | film_list                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | film_list                  | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | film_text                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | film_text                  | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | inventory                  | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | inventory                  | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | language                   | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | language                   | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | nicer_but_slower_film_list | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | nicer_but_slower_film_list | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | payment                    | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | payment                    | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | rental                     | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | rental                     | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | staff                      | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | staff                      | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | staff_list                 | root@localhost | 2009-01-09 13:44:56 | S
| localhost | sakila | dave    | staff_list                 | root@localhost | 2009-01-09 13:46:25 | S
| localhost | sakila | manager | store                      | root@localhost | 2009-01-09 13:44:57 | S
| localhost | sakila | dave    | store                      | root@localhost | 2009-01-09 13:46:25 | S
+-----------+--------+---------+----------------------------+----------------+---------------------+--
42 rows in set (0.00 sec)

mysql>
```

We sorted by table, so we can do a quick visual comparison of the `dave` and `manager` users. Looks like our query worked - `dave` is a good clone of `manager`.

Any time we touch a table in the mysql database we need to run the `FLUSH PRIVILEGES;` command. Since we touched MySQL's internal tables, we have one more command to run.

Type the following at the MySQL prompt:

```
mysql> FLUSH PRIVILEGES;
```

Note

If you get the error message **ERROR 1146 (42S02): Table 'mysql.servers' doesn't exist** when trying to use FLUSH PRIVILEGES, exit your mysql server and execute the following command at the shell prompt:

```
cat /usr/share/mysql/mysql_fix_privilege_tables.sql | /usr/bin/mysql --no-defaults --force --
user=root --socket=mysql/data/mysql.sock --database=mysql -p
```

You will likely see some error messages, but these can be ignored. If you continue to receive errors with FLUSH PRIVILEGES after running this command, contact your mentor.

Once that is done, log out of MySQL, and log back in as **dave**. We can check dave's permissions by running a query.

Type the following at the MySQL prompt:

```
mysql> use sakila; select * from address limit 0, 5;
```

As long as you typed everything correctly in the previous steps, you'll see the desired results:

OBSERVE:

```
mysql> use sakila; select * from address limit 0, 5;
Database changed
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
| address_id | address              | address2 | district | city_id | postal_code | phone       | last
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
|          1 | 47 MySakila Drive    | NULL     | Alberta  |     300 |             |             | 2006
|          2 | 28 MySQL Boulevard   | NULL     | QLD      |     576 |             |             | 2006
|          3 | 23 Workhaven Lane    | NULL     | Alberta  |     300 |             | 14033335568 | 2006
|          4 | 1411 Lillydale Drive | NULL     | QLD      |     576 |             | 6172235589  | 2006
|          5 | 1913 Hanoi Way       |          | Nagasaki |     463 | 35200       | 28303384290 | 2006
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
5 rows in set (0.01 sec)

mysql>
```

Looks great!

To check the permissions, log out of MySQL, and log back in as **manager**.

Type the RED text at the Unix prompt:

```
mysql> exit
Bye
cold:~/mysql$ mysql -u manager -p
```

If you type everything correctly you'll see the exact same results for manager as you did for dave.

OBSERVE:

```
mysql> use sakila; select * from address limit 0, 5;
Database changed
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
| address_id | address              | address2 | district | city_id | postal_code | phone       | last
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
|          1 | 47 MySakila Drive    | NULL     | Alberta  |     300 |             |             | 2006
|          2 | 28 MySQL Boulevard   | NULL     | QLD      |     576 |             |             | 2006
|          3 | 23 Workhaven Lane    | NULL     | Alberta  |     300 |             | 14033335568 | 2006
|          4 | 1411 Lillydale Drive | NULL     | QLD      |     576 |             | 6172235589  | 2006
|          5 | 1913 Hanoi Way       |          | Nagasaki |     463 | 35200       | 28303384290 | 2006
+------------+----------------------+----------+----------+---------+-------------+-------------+-----
5 rows in set (0.01 sec)

mysql>
```

Adding the Grader

Before you go any further in this course, you need to add an account for your instructor so they can see your database for grading purposes. Reconnect as root before executing the following statement.

CODE TO TYPE:

```
mysql> GRANT ALL PRIVILEGES ON *.* to 'gradessl'@'%' REQUIRE SUBJECT '/C=US/ST=Illinois/L=Champaign/O=
```

You've covered a lot of ground in this lesson! User security is of top importance for all databases. Always follow the rule of least privilege: only grant users the minimum amount of privilege.

In the next lesson we'll continue our discussion of database security by examining column level security, and discussing how we can limit access to individual rows in the database. See you then!

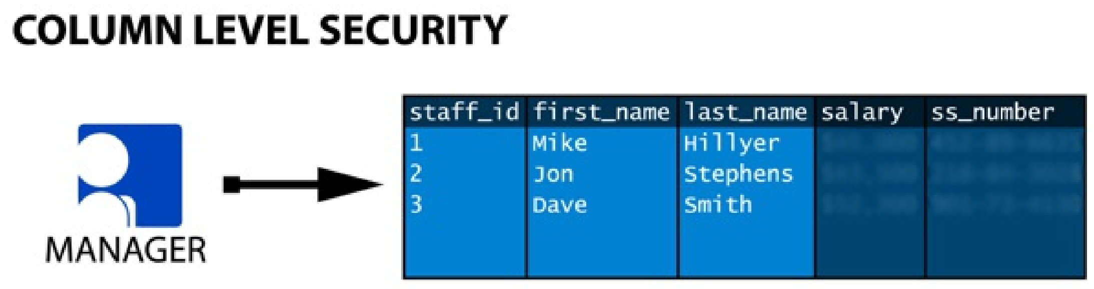
Advanced Security
DBA 2: Administering MySQL Lesson 4

Column Security

Welcome back! In the previous lesson we learned how to add users to the database, and how to specify table level security for those users.

Many databases (MySQL included) also allow administrators to specify security for individual columns in database tables. This can be very useful, depending on your application.

In our sakila database, we have a table called staff to store information about employees. Suppose the owner needs to store payroll information like salary and social security number. This is very private information that should only be viewed by the owner. Managers and clerks should have no access to that information. However, employee information such as address or phone number can safely be seen by everyone.



staff_id	first_name	last_name	salary	ss_number
1	Mike	Hillyer		
2	Jon	Stephens		
3	Dave	Smith		

We can utilize MySQL's built-in column level security to limit access to sensitive columns. Before we can limit access, we'll need to add columns that will store salary and social security information to the staff table.

Make sure MySQL is running, and log in as root. If you need a refresher on how to do this, refer back to the previous lessons.

Type the following at the MySQL prompt:

```
mysql> use sakila;
mysql> alter table staff add salary decimal (10,2);
mysql> alter table staff add ss_number varchar(9);
```

Now that we have the new columns, let's limit access for the manager user. We'll do this by removing **all permissions** (just to make sure we start from an empty set of permissions), then adding back desired column permissions. The syntax for the `grant` statement is nearly the same as the statement you saw in the previous lesson, but this time we'll specify the individual columns that a manager is allowed to select (in **red**).

Type the following at the MySQL prompt:

```
mysql> revoke all on sakila.staff from 'manager'@'localhost';
mysql> grant select(staff_id, first_name, last_name, address_id, picture, email,
store_id, active, username, last_update) on sakila.staff to 'manager'@'localhost';
```

If everything was entered correctly you'll see something like this:

OBSERVE:

```
mysql> revoke all on sakila.staff from 'manager'@'localhost';
Query OK, 0 rows affected (0.01 sec)

mysql> grant select(staff_id, first_name, last_name, address_id, picture, email, store_id, active, use
Query OK, 0 rows affected (0.01 sec)

mysql>
```

Looks good! Let's try out our new security settings. Log out of MySQL, and log back in as **manager**. Once back in, we'll connect to the sakila database, and try to view the structure of the **staff** table.

Type the following at the MySQL prompt:

```
mysql> use sakila; explain staff;
```

Well, MySQL doesn't even shows us that the columns **password**, **ss_number** and **salary** are part of the staff table.

OBSERVE:

```
mysql> use sakila; explain staff;
Database changed
+-------------+----------------------+------+-----+-------------------+----------------+
| Field       | Type                 | Null | Key | Default           | Extra          |
+-------------+----------------------+------+-----+-------------------+----------------+
| staff_id    | tinyint(3) unsigned  | NO   | PRI | NULL              | auto_increment |
| first_name  | varchar(45)          | NO   |     | NULL              |                |
| last_name   | varchar(45)          | NO   |     | NULL              |                |
| address_id  | smallint(5) unsigned | NO   | MUL | NULL              |                |
| picture     | blob                 | YES  |     | NULL              |                |
| email       | varchar(50)          | YES  |     | NULL              |                |
| store_id    | tinyint(3) unsigned  | NO   | MUL | NULL              |                |
| active      | tinyint(1)           | NO   |     | 1                 |                |
| username    | varchar(16)          | NO   |     | NULL              |                |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+-------------+----------------------+------+-----+-------------------+----------------+
10 rows in set (0.00 sec)

mysql>
```

But can we query those columns? As a manager, we should not be able to. Try it!

Type the following at the MySQL prompt:

```
mysql> select password, salary, ss_number from staff;
```

MySQL is doing its job, it doesn't let us return the password, salary or ss_number columns.

OBSERVE:

```
mysql> select password, salary, ss_number from staff;
ERROR 1143 (42000): SELECT command denied to user 'manager'@'localhost' for column 'password' in table
mysql>
```

password is the first column in the query that manager cannot SELECT, so it is the column specified in the error you see. Can we still see the other columns? A simple query will tell us.

Type the following at the MySQL prompt:

```
mysql> select first_name, last_name from staff;
```

If everything goes correctly you'll see the first and last name of two rows in the table.

OBSERVE:

```
mysql> select first_name, last_name from staff;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
| Jon        | Stephens  |
+------------+-----------+
2 rows in set (0.00 sec)
```

Looks great!

Row Security

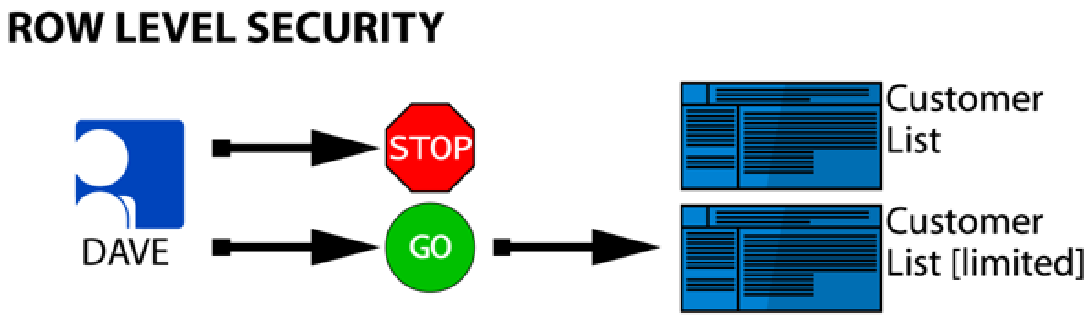
In the last section we successfully limited access to three columns in the staff table. Doing so prevents ordinary users from accessing secure information.

What if you need to restrict access to certain rows in your table? Consider the **customer** table in our **sakila** database. Each customer is tied to a specific store through *store_id*. Let's say the owner creates a new policy stating that managers and clerks can only deal with customers within their own store. How will you enforce this policy?

In some databases this type of restriction can only be done at the application level. This is potentially insecure, since each application might implement access rules in a slightly different way. Additionally, the user could still try to query the database directly.

MySQL doesn't have any specific functionality to implement row level security, unlike larger databases such as Oracle. There are two common ways to implement row level security without specific database assistance - by using **views** and/or **stored procedures**. Instead of granting users access to tables, users are only allowed access to a view or stored procedure.

Suppose **dave** wants to access the customer list. He is not allowed access to the **Customer List** table, instead he must query the **Customer List Limited** view. This view contains a where clause to limit the rows returned to the current user, **dave**.



In order to limit the rows a specific user can view, we will need to know the username of the person who is connected to the database. In MySQL this is determined by the `user()` function.

Make sure you're connected to MySQL as root.

Type the following at the MySQL prompt:

```
mysql> select user();
```

MySQL happily responds with your user information:

OBSERVE:

```
mysql> select user();
+----------------+
| user()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)

mysql>
```

Recall in the last lesson that we granted **dave** access to the database, but we didn't add his data to the appropriate places within the application. Usually this is done by a manager, using the application, but we'll just use SQL since this course doesn't use an application.

To add him to the application we'll need to add an address for 'dave' and add him to the staff table. Let's do this now, assigning dave to store_id = 1. Make sure you're connected to MySQL as **root**.

Type the following at the MySQL prompt:

```
mysql> insert into address (address, district, city_id, phone)
values ('123 4th Street', 'Alberta', 300, '8885551212');

mysql> insert into staff (first_name, last_name, address_id, email, store_id, active, username)
values ('Dave','Smith', LAST_INSERT_ID(), 'dave@sakilastore.org', 1, 1, 'dave');
```

Run a quick query to make sure Dave was entered into the system.

Type the following at the MySQL prompt:

```
mysql> select * from staff_list;
```

Your results should look something like the following:

OBSERVE:

```
mysql> select * from staff_list;
```

```
+----+--------------+-----------------------+----------+-------------+------------+-----------+-----+
| ID | name         | address               | zip code | phone       | city       | country   | SID |
+----+--------------+-----------------------+----------+-------------+------------+-----------+-----+
|  1 | Mike Hillyer | 23 Workhaven Lane     |          | 14033335568 | Lethbridge | Canada    |   1 |
|  2 | Jon Stephens | 1411 Lillydale Drive  |          | 6172235589  | Woodridge  | Australia |   2 |
|  3 | Dave Smith   | 123 4th Street        |          | 8885551212  | Lethbridge | Canada    |   1 |
+----+--------------+-----------------------+----------+-------------+------------+-----------+-----+
3 rows in set (0.00 sec)
```

Now we can focus on granting dave access to the customers that share his store_id. We'll do this by creating a new view **customer_list_limited** which will be based on the existing view called **customer_list**.

What if we don't know the select statement that is the basis of the **customer_list** view? Fortunately MySQL keeps the definition for us, in its special **INFORMATION_SCHEMA** database. We can query that to get the view definition.

Note For more information on **INFORMATION_SCHEMA**, refer back to DBA 1 Lab 12, or visit MySQL's web site.

Type the following at the MySQL prompt:

```
mysql> select view_definition from information_schema.views where table_name='customer_list' \G
```

The output is a little difficult to read, but we can reformat it.

OBSERVE:

```
mysql>  select view_definition from information_schema.views where table_name='customer_list' \G
*************************** 1. row ***************************
view_definition: /* ALGORITHM=UNDEFINED */ select `cu`.`customer_id` AS `ID`,concat(`cu`.`first_name`,
1 row in set (0.01 sec)
```

Below is the reformatted query.

OBSERVE:

```
select
cu.customer_id AS ID,
concat(cu.first_name,_utf8' ',cu.last_name) AS name,
a.address AS address,
a.postal_code AS 'zip code',
a.phone AS phone,
sakila.city.city AS city,
sakila.country.country AS country,
if(cu.active,_utf8'active',_utf8'') AS notes,
cu.store_id AS SID
from
sakila.customer cu
join sakila.address a on (cu.address_id = a.address_id)
join sakila.city on (a.city_id = sakila.city.city_id)
join sakila.country on (sakila.city.country_id = sakila.country.country_id)
```

Try running this query. It will return 599 rows, just like the view **customer_list**:

OBSERVE:

```
mysql> select
    -> cu.customer_id AS ID,
    -> concat(cu.first_name,_utf8' ',cu.last_name) AS name,
    -> a.address AS address,
    -> a.postal_code AS 'zip code',
    -> a.phone AS phone,
    -> sakila.city.city AS city,
    -> sakila.country.country AS country,
    -> if(cu.active,_utf8'active',_utf8'') AS notes,
    -> cu.store_id AS SID
    -> from
    -> sakila.customer cu
    -> join sakila.address a on (cu.address_id = a.address_id)
    -> join sakila.city on (a.city_id = sakila.city.city_id)
    -> join sakila.country on (sakila.city.country_id = sakila.country.country_id);
```

```
+-----+------------------------+-----------------------------------------+----------+---------------+---
| ID  | name                   | address                                 | zip code | phone         | cit
+-----+------------------------+-----------------------------------------+----------+---------------+---
| 218 | VERA MCCOY             | 1168 Najafabad Parkway                  | 40301    | 886649065861 | Kab
| 441 | MARIO CHEATHAM         | 1924 Shimonoseki Drive                  | 52625    | 406784385440 | Bat
|  69 | JUDY GRAY              | 1031 Daugavpils Parkway                 | 59025    | 107137400143 | Bch
... lines omitted
|   7 | MARIA MILLER           | 900 Santiago de Compostela Parkway      | 93896    | 716571220373 | Kra
| 553 | MAX PITT               | 1917 Kumbakonam Parkway                 | 11892    | 698182547686 | Nov
| 438 | BARRY LOVELACE         | 1836 Korla Parkway                      | 55405    | 689681677428 | Kit
+-----+------------------------+-----------------------------------------+----------+---------------+---
599 rows in set (0.06 sec)

mysql>
```

To make this query (and eventually the view) handle row level security, we'll have to add a join to the **staff** table, as well as a `WHERE` clause to limit the current `user()`. Since the `user()` function returns a slightly different result than we need (we only want the username, not the "@localhost" part), we'll also have to do a bit of manipulation.

Removing the "@localhost" part from `user()` isn't difficult - we can use the `SUBSTRING_INDEX` function. `SUBSTRING_INDEX` takes three arguments - a *string*, a *delimiter*, and a *count*. `SUBSTRING_INDEX` returns the text from the *string* before *count* occurrences of the *delimiter*. For more information, check out MySQL's web site.

Try it out!

Type the following at the MySQL prompt:

```
mysql> select substring_index(user(), '@', 1);
```

As long as you're still logged in as root you'll see:

OBSERVE:

```
mysql> select substring_index(user(), '@', 1);
+----------------------------------+
| substring_index(user(), '@', 1)  |
+----------------------------------+
| root                             |
+----------------------------------+
1 row in set (0.01 sec)

mysql>
```

Let's rewrite our select query. We'll add a **join** to the **sakila.staff** table on **store_id**, and add a **WHERE** clause to limit the **staff** table to the current username - **substring_index(user(),'@', 1)**. Try it while logged in as **root**.

Type the following at the MySQL prompt:

```
select
cu.customer_id AS ID,
concat(cu.first_name,_utf8' ',cu.last_name) AS name,
a.address AS address,
a.postal_code AS 'zip code',
a.phone AS phone,
sakila.city.city AS city,
sakila.country.country AS country,
if(cu.active,_utf8'active',_utf8'') AS notes
from
sakila.customer cu
join sakila.address a on (cu.address_id = a.address_id)
join sakila.city on (a.city_id = sakila.city.city_id)
join sakila.country on (sakila.city.country_id = sakila.country.country_id)
join sakila.staff s on (cu.store_id = s.store_id)
WHERE s.username = substring_index(user(), '@', 1);
```

When logged in as root you'll see something like `Empty set (0.01 sec)` when you run the query. Why do you think this is? This is actually expected, since our **staff** table doesn't have a row with username of **root**.

Log out, then reconnect to MySQL as **dave** Try the query again, making sure you are using the **sakila** database. This time your results will be much different:

OBSERVE:

```
+-----+----------------------+-----------------------------------------+----------+--------------+-----
| ID  | name                 | address                                 | zip code | phone        | city
+-----+----------------------+-----------------------------------------+----------+--------------+-----
|   1 | MARY SMITH           | 1913 Hanoi Way                          | 35200    | 28303384290  | Sase
|   2 | PATRICIA JOHNSON     | 1121 Loja Avenue                        | 17886    | 838635286649 | San 
|   3 | LINDA WILLIAMS       | 692 Joliet Street                       | 83579    | 448477190408 | Athe
|   5 | ELIZABETH BROWN      | 53 Idfu Parkway                         | 42399    | 10655648674  | Nant

... (lines omitted)

| 594 | EDUARDO HIATT        | 1837 Kaduna Parkway                     | 82580    | 640843562301 | Jini
| 595 | TERRENCE GUNDERSON   | 844 Bucuresti Place                     | 36603    | 935952366111 | Jinz
| 596 | ENRIQUE FORSYTHE     | 1101 Bucuresti Boulevard                | 97661    | 199514580428 | Patr
| 597 | FREDDIE DUGGAN       | 1103 Quilmes Boulevard                  | 52137    | 644021380889 | Sull
| 598 | WADE DELVALLE        | 1331 Usak Boulevard                     | 61960    | 145308717464 | Laus
+-----+----------------------+-----------------------------------------+----------+--------------+-----
326 rows in set (0.03 sec)
```

Sure enough, the customer list is now being limited by the current user's store. This means we are ready to implement the view. Log back into MySQL as **root** -- otherwise you won't have the permission to create a view!

Type the following at the MySQL prompt:

```
CREATE VIEW customer_list_limited
AS
select
cu.customer_id AS ID,
concat(cu.first_name,_utf8' ',cu.last_name) AS name,
a.address AS address,
a.postal_code AS 'zip code',
a.phone AS phone,
sakila.city.city AS city,
sakila.country.country AS country,
if(cu.active,_utf8'active',_utf8'') AS notes
from
sakila.customer cu
join sakila.address a on (cu.address_id = a.address_id)
join sakila.city on (a.city_id = sakila.city.city_id)
join sakila.country on (sakila.city.country_id = sakila.country.country_id)
join sakila.staff s on (cu.store_id = s.store_id)
WHERE s.username = substring_index(user(), '@', 1);
```

You'll also need to grant access to the appropriate people for your new view, so let's do that now.

Type the following at the MySQL prompt:

```
mysql> grant select on customer_list_limited to 'manager'@'localhost';
mysql> grant select on customer_list_limited to 'dave'@'localhost';
```

If you've typed everything correctly you'll see the familiar `Query OK, 0 rows affected (0.03 sec)` for the view creation and for the grant statements.

Now let's try selecting from the view. Log out of MySQL, and back in as **dave**. If our view was created successfully, dave will only see customers at his own store (326 customers).

Type the following at the MySQL prompt:

```
mysql> select * from customer_list_limited;
```

It should look something like this:

OBSERVE:

```
mysql> select * from customer_list_limited;
+-----+----------------------+-----------------------------------------+----------+--------------+-----
| ID  | name                 | address                                 | zip code | phone        | city
+-----+----------------------+-----------------------------------------+----------+--------------+-----
|   1 | MARY SMITH           | 1913 Hanoi Way                          | 35200    | 28303384290  | Sase
|   2 | PATRICIA JOHNSON     | 1121 Loja Avenue                        | 17886    | 838635286649 | San 
|   3 | LINDA WILLIAMS       | 692 Joliet Street                         | 83579    | 448477190408 | Athe
|   5 | ELIZABETH BROWN      | 53 Idfu Parkway                           | 42399    | 10655648674  | Nant

... (lines omitted)

| 594 | EDUARDO HIATT        | 1837 Kaduna Parkway                       | 82580    | 640843562301 | Jini
| 595 | TERRENCE GUNDERSON   | 844 Bucuresti Place                       | 36603    | 935952366111 | Jinz
| 596 | ENRIQUE FORSYTHE     | 1101 Bucuresti Boulevard                  | 97661    | 199514580428 | Patr
| 597 | FREDDIE DUGGAN       | 1103 Quilmes Boulevard                    | 52137    | 644021380889 | Sull
| 598 | WADE DELVALLE        | 1331 Usak Boulevard                       | 61960    | 145308717464 | Laus
+-----+----------------------+-------------------------------------------+----------+--------------+----
326 rows in set (0.03 sec)
```

There you have it - you've limited access to rows of data based on the current user. At this point you'd probably replace the view **customer_list** with the new view **customer_list_limited**, or at least remove access to the old view **customer_list** for dave.

Normally views and stored procedures are available only to the user who created them. A user with sufficient grant privileges (such as those held by the root account) can allow other users to use views and stored procedures that those other users did not create. In fact, we did just that when we granted the select privilege on the customer_list_limited view to the users 'manager'@'localhost' and 'dave'@'localhost'. (While we use the select privilege in the grant statement for views, the corresponding privilege for stored procedures is executed within procedure.)

The creator of a view or stored procedure can include an optional SQL SECURITY characteristics clause in the create view or create procedure statement. The default SQL SECURITY value is DEFINER. With SQL SECURITY DEFINER, a user selects from a view or calls a procedure with the privileges of the user who created it. Using SQL SECURITY DEFINER, a DBA can allow users who do not have specific table privileges to select from a view or call a procedure that accesses those tables.

Programmers may also specifically define the SQL SECURITY value to be INVOKER. With the more restrictive SQL SECURITY INVOKER, a user selects from a view or calls a procedure using their own privileges. If a view or procedure accesses a particular table, the user would require the appropriate privileges for the view or procedure, as well as the underlying table.

For more information on stored program and view security visit the MySQL web site.

In the last two lessons we've covered many aspects of database security, from granting and revoking user permissions, to limiting access to columns and rows of data. In the next lesson we'll shift our focus to the tables themselves, and learn how to keep our database performing well through proper index management. See you there!

Indexing Databases, Part I
DBA 2: Administering MySQL Lesson 5

In the past few lessons we've focused primarily on securing our database. In this lesson we are going to shift gears to discuss another important topic for database administrators: **indexes**.

What are Indexes?

Database indexes are just like book indexes - they are structures the database engine consults to quickly find information located in a table. If a table does not have an index, a database must read the entire table, row by row, in order to answer a query. This may not take a lot of time if a table has a few rows, but it certainly would take a very long time with millions of rows. According to MySQL, reading a table with 1,000 rows (and no index) is at least 100 times slower than using an index.

Note The words *index* and *key* are usually interchangeable when talking about databases.

In MySQL, indexes are usually stored in structures called *B-trees*. *B-trees* keep data in a sorted order. Databases let you specify the default sort - either *ascending* or *descending*. MySQL version 5 lets you specify the sort order, but currently ignores your specification and stores the index in ascending order.

Physically, indexes are stored in a file next to table data. With very large databases, you might start to consider physical storage when administering and optimizing databases. Before you do that, you have to pick the correct indexes. We won't concern ourselves with the physical storage for this course - that is a topic for very advanced database administration.

Indexes have many uses in databases. They:

- **...are consulted when a column is referenced in the `WHERE` clause.** Matching (and even non-matching) rows can filtered using an index.
- **...are used to quickly join tables.** Columns must be the same data type and size in order for an index to be used. Indexes won't be used if data types don't match, or if the join includes a function, like `substring`.
- **...are used to sort rows**, since indexes are sorted.

Primary Keys (indexes)

A *primary key* specifies the minimum set of columns needed to uniquely identify a row in a database table. This means that the primary key column(s) must be unique cross the entire table. A good example of a unique bit of information would be an account number - one account number would point to exactly one row in a table of accounts. A column that isn't unique would be name - many people in the United States share the first name *John*.

All tables should have a primary key. Tables that do not have primary keys could contain duplicate rows, which cannot be individually selected, updated, or deleted. This usually presents a problem!

What if we don't have any unique information? We can have the database create a unique column for us. MySQL has a keyword called **AUTO_INCREMENT** that will automatically populate a column with an increasing integer number. The value for the first row is 1, then the next row is 2, and so on. It does not repeat numbers.

This type of key is also known as a *surrogate key*, and is what was used for the **payment_id** column of the `payment` table in the sakila database. Take a look at the `CREATE TABLE` statement:

OBSERVE:

```
CREATE TABLE payment (
  payment_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_id SMALLINT UNSIGNED NOT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (payment_id),
  KEY idx_fk_staff_id (staff_id),
  KEY idx_fk_customer_id (customer_id),
  CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id)
    REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT fk_payment_customer FOREIGN KEY (customer_id)
   REFERENCES customer (customer_id) ON DELETE RESTRICT ON UPDATE CASCADE,
  CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id)
   REFERENCES staff (staff_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

For the `payment` table, **payment_id** is a *surrogate key* and an **AUTO_INCREMENT** column. It is also the **PRIMARY KEY** for the table.

Other Indexes

Chances are we need to index columns besides the primary key. We can do this when creating the table as well:

OBSERVE:

```
CREATE TABLE payment (
  payment_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_id SMALLINT UNSIGNED NOT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (payment_id),
  KEY idx_fk_staff_id (staff_id),
  KEY idx_fk_customer_id (customer_id),
  CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id)
    REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT fk_payment_customer FOREIGN KEY (customer_id)
   REFERENCES customer (customer_id) ON DELETE RESTRICT ON UPDATE CASCADE,
  CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id)
   REFERENCES staff (staff_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

In the `payment` table we specified an index called **idx_fk_staff_id** on the **staff_id** column. This column is also a *foreign key*, as we will soon see.

Foreign Keys in MySQL

A *foreign key* is an indexed column (or set of columns) in a child table that references the primary key column (or set of columns) in a parent table. A row in the child table cannot have a foreign key value that does not exist in the parent table. It is both an index to speed query execution, and a constraint that enforces the parent-child relationship between two tables.

You can think of the *parent* table like the Department of Motor Vehicles (the DMV). If you want a driver's license number, you must consult the DMV, and only the DMV. No other agency can assign a driver's license number, or verify the authenticity of a driver's license number.

Child tables are like your bank, insurance company, and employer. All want to know your driver's license number and keep track of it, but they are not the authority over it.

In sakila, there is a table that stores customer information, called `customer`. Each row in this table uniquely identifies a single customer by a column called **customer_id**. We know that **customer_id** is unique because it is also the **PRIMARY KEY**. Take a look at the table:

OBSERVE:

```
CREATE TABLE customer (
  customer_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  store_id TINYINT UNSIGNED NOT NULL,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  email VARCHAR(50) DEFAULT NULL,
  address_id SMALLINT UNSIGNED NOT NULL,
  active BOOLEAN NOT NULL DEFAULT TRUE,
  create_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (customer_id),
  KEY idx_fk_store_id (store_id),
  KEY idx_fk_address_id (address_id),
  KEY idx_last_name (last_name),
  CONSTRAINT fk_customer_address FOREIGN KEY (address_id) REFERENCES address (address_id) ON DE
  CONSTRAINT fk_customer_store FOREIGN KEY (store_id) REFERENCES store (store_id) ON DELETE RES
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Since this table is *the* single source of customers, uniquely identified by **customer_id**, it is a parent table to other tables that reference **customer_id**. One of the child tables of the `customer` table is the `payment` table:

OBSERVE:

```
CREATE TABLE payment (
  payment_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_id SMALLINT UNSIGNED NOT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (payment_id),
  KEY idx_fk_staff_id (staff_id),
  KEY idx_fk_customer_id (customer_id),
  CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id)
    REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT fk_payment_customer FOREIGN KEY (customer_id)
   REFERENCES customer (customer_id) ON DELETE RESTRICT ON UPDATE CASCADE,
  CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id)
   REFERENCES staff (staff_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

In this case, `payment` has a column called **customer_id** that is also a foreign key called **fk_payment_customer** that references the **customer** table.

Note Foreign keys don't have to reference other tables. Foreign keys that reference the same table are known as *self-referencing* or *recursive* foreign keys.

In order to specify foreign keys in MySQL, and have the database enforce those constraints, you must use the *InnoDB* table type.

What happens to child tables when rows change in the parent tables? Databases let you specify the behavior for `updates` and `deletes`. For updates, only changes to the primary key are considered. The actions are:

- **CASCADE**: When the parent row is updated or deleted, child rows are updated or deleted as well.
- **RESTRICT**: Updates or deletes are not allowed for parent rows, if corresponding child rows exist.
- **NO ACTION**: Child rows are not updated or deleted. Changes that would orphan child rows are still not allowed, and would result in a foreign key relationship violation.
- **SET NULL**: The child row is set to NULL when a parent is updated or deleted.
- **SET DEFAULT**: The child row is set to the column default value when a parent is updated or deleted. (The default value could be NULL).

You can specify the behavior for a foreign key when you create the table. Let's take a look at the `CREATE TABLE` statement for the `payment` table in the sakila database.

OBSERVE:

```
CREATE TABLE payment (
  payment_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_id SMALLINT UNSIGNED NOT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (payment_id),
  KEY idx_fk_staff_id (staff_id),
  KEY idx_fk_customer_id (customer_id),
  CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id)
    REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT fk_payment_customer FOREIGN KEY (customer_id)
   REFERENCES customer (customer_id) ON DELETE RESTRICT ON UPDATE CASCADE,
  CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id)
   REFERENCES staff (staff_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

In this statement, a **CONSTRAINT** called **fk_payment_rental** is added to `payment`. It is a **FOREIGN KEY** on the column **rental_id** whose parent is the column **rental_id** on the **rental** table. The statement **ON DELETE SET NULL**, means that if a row is deleted from the parent **rental** table, corresponding **rental_id** values in the child **payment** table are set to **NULL**. The statement **ON UPDATE CASCADE** means that updates to the parent **rental_id** column in the **rental** table are automatically updated on rows in the child `payment` table.

Earlier we examined the index on the **staff_id** column, called **idx_fk_staff_id**. The **CONSTRAINT** in this `CREATE TABLE` statement also makes **staff_id** a **FOREIGN KEY**. This foreign key is slightly different - the **ON DELETE RESTRICT** tells MySQL that deletes are not allowed from the parent **staff** table if corresponding rows exist in the child `payment` table.

For more information on foreign keys, check out MySQL's web site.

Fulltext Indexes

MySQL has another type of index: a *full-text* index. This type of index is used to search through long ("*full*") text columns. Full-text indexes are not replacements for traditional indexes, since they are only good at one thing: searching full text.

MySQL has some restrictions on full-text indexes. The most important restriction is that full-text indexes can only be used with the MyISAM table type

Full-text indexes were discussed in the previous DBA course, in lesson 11. Feel free to revisit that lesson if you would like more information, or take a look at MySQL's web site.

Indexes on Existing Tables

You don't have to drop and recreate a table just to add an index. You can use the `ALTER TABLE` command.

Suppose a developer sends you an email asking you to create an index on the `last_update` column of the payment table. Let's add one! Make sure you are logged in as root. We will use **ALTER TABLE** to **add an index** called **ix_payment_date** on the column **payment_date** of the `payment` table.

Type the following at the MySQL prompt:

```
mysql> ALTER TABLE payment ADD INDEX ix_payment_date (payment_date);
```

If everything went well, you will see something like:

OBSERVE:

```
mysql> ALTER TABLE payment ADD INDEX ix_payment_date (payment_date);
Query OK, 16049 rows affected (1.42 sec)
Records: 16049  Duplicates: 0  Warnings: 0

mysql>
```

Creating indexes takes time, since the database must read every row in the table to build the index. During this time no other users are allowed to insert, update or delete rows from the table. For this reason you should be careful when adding indexes to existing tables - large tables may be unavailable for a very long time due to index changes.

Suppose the same developer sends you another email, saying the index on `last_update` wasn't needed. How can you delete the unnecessary index? Use **ALTER TABLE** with the **DROP INDEX** keyword and **the name of the index**.

Type the following at the MySQL prompt:

```
mysql> ALTER TABLE payment DROP INDEX ix_payment_date;
```

If you typed that correctly you will see:

OBSERVE:

```
mysql> ALTER TABLE payment DROP INDEX ix_payment_date;
Query OK, 16049 rows affected (0.82 sec)
Records: 16049  Duplicates: 0  Warnings: 0

mysql>
```

That is all there is to it! If you would like more information on the `ALTER` command, be sure to check out MySQL's web site.

Now we know what indexes can do for us. In the next lesson we'll discuss the procedure for maintaining these indexes. See you then!

Indexing Databases, Part II
DBA 2: Administering MySQL Lesson 6

In the last lesson we discussed indexes, and how they are used in the database. In this lesson we'll discuss the steps you should take when indexing a database table.

Selecting Columns

All tables have an optimal set of indexes - which may be no indexes at all. Over time, indexes may outlive their usefulness. Indexes will be added, removed, and altered as data is added to the database and query patterns change. A common mistake made by young programmers and database administrators who are eager to "optimize" performance is to add indexes to satisfy every possible query imaginable. This overzealous indexing strategy is the wrong approach for a few reasons:

- Too many indexes will hurt database performance, especially for inserts, updates and deletes.
- Query patterns change, so it is impossible to implement indexes to cover every possible query.
- For small tables database engines may ignore all indexes since it is often faster to grab all rows in the table than to work with indexes.

Above all, how do you know an index will help performance, without any data to support your solution? You wouldn't replace the engine in your car just because it started running slowly!

Note Some databases (such as SQL Server) have two or more types of indexes. Each type of index has certain properties, and choosing the proper index has a direct effect on database performance. Generally speaking, MySQL only has one type of index, and we don't have to concern ourselves with that type of detail.

If you are creating a new database, there are a couple of base rules you should follow to get your database off to a good start. Below are the steps you should take when starting a new database project.

INDEXING STEPS

1. Always create primary keys for tables. If your table doesn't seem to have a primary key, or the primary key would be all (or nearly all) columns, consider adding a *surrogate key*. A surrogate key is a primary key that is generated by the database - in MySQL this would be an **AUTO_INCREMENT** column.
2. Set up foreign key constraints and index appropriately.
3. Create indexes for columns used in the WHERE clause for frequently used queries. For example, if you are always querying the `customer` table, using `WHERE last_name='some value`, consider indexing the `last_name` column.

To make the job of the database administrator easier, MySQL will let you see its query plan in order to figure out what is going on "under the hood." The command you use to view the query plan is the same command you use to show the structure of the table - `EXPLAIN`. `DESCRIBE` is a synonym that can also be used.

`EXPLAIN` shows the tables used, indexes (keys) used, indexes that could have been used in your query, and parts of the `WHERE` clause. It also shows some statistics such as size of indexes and the number of rows in the table.

Let's **explain** a simple query. Make sure you're logged into MySQL as root and connected to the sakila database before you perform these steps.

Type the following at the MySQL prompt:

```
mysql> explain select * from rental;
```

As long as you typed everything correctly, you'll see one row returned:

OBSERVE:

```
mysql> explain select * from rental;
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
| id | select_type | table  | type | possible_keys | key  | key_len | ref  | rows  | Extra |
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
|  1 | SIMPLE      | rental | ALL  | NULL          | NULL | NULL    | NULL | 16298 |       |
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
1 row in set (0.00 sec)
```

What do these columns mean? A full description can be found at MySQL's site, however a condensed version is below.

Column	Description

id	The sequential number of the `select` in the query
select_type	Type of select statement - if it is a simple query, sub query, or other
table	The name of the table used in the query
type	Join type - how this table is combined with other tables to form the result
possible_keys	The indexes that might be used to satisfy the query
key	The actual index that will be used in the query, or NULL if no index was used
key_len	The length of key that will be used
ref	The columns used to join tables
rows	The number of rows examined for the query
Extra	Notes on how MySQL will process the query, like if it needs to use a temporary table

Let's look back at the results from the previous `EXPLAIN` statement.

OBSERVE:

```
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
| id | select_type | table  | type | possible_keys | key  | key_len | ref  | rows  | Extra |
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
|  1 | SIMPLE      | rental | ALL  | NULL          | NULL | NULL    | NULL | 16298 |       |
+----+-------------+--------+------+---------------+------+---------+------+-------+-------+
```

We are selecting from the **rental** table, which doesn't have any **possible_keys** (**NULL**), so MySQL **won't use any indexes**, and the database needs to look at approximately **16298 rows**. Your row count might be slightly different - it is just an estimation. There are no other tables listed since we don't have any joins. There is no information about a `WHERE` clause, since we didn't use a `WHERE` clause.

The result of `EXPLAIN` doesn't show any **possible_keys** on the table rental. This means MySQL didn't find any indexes on **rental** that could be used to answer the query. This doesn't mean that **rental** doesn't have any indexes, however. How can we check to see what indexes exist on rental? We'll use the `show index` statement.

Type the following at the MySQL prompt:

```
mysql> show index from rental;
```

This query returns a lot of useful information.

OBSERVE:

```
mysql> show index from rental;
+--------+------------+----------------------+--------------+--------------+-----------+-------------+-
| Table  | Non_unique | Key_name             | Seq_in_index | Column_name  | Collation | Cardinality |
+--------+------------+----------------------+--------------+--------------+-----------+-------------+-
| rental |          0 | PRIMARY              |            1 | rental_id    | A         |       16298 |
| rental |          0 | rental_date          |            1 | rental_date  | A         |       16298 |
| rental |          0 | rental_date          |            2 | inventory_id | A         |       16298 |
| rental |          0 | rental_date          |            3 | customer_id  | A         |       16298 |
| rental |          1 | idx_fk_inventory_id  |            1 | inventory_id | A         |       16298 |
| rental |          1 | idx_fk_customer_id   |            1 | customer_id  | A         |        1253 |
| rental |          1 | idx_fk_staff_id      |            1 | staff_id     | A         |           3 |
+--------+------------+----------------------+--------------+--------------+-----------+-------------+-
7 rows in set (0.03 sec)
```

The keys in `rental` are:

- the **PRIMARY** key on **rental_id**
- the composite key on **rental_date** on **rental_date**, **inventory_id** and **customer_id**
- the foreign key **idx_fk_inventory_id** on **inventory_id**
- the foreign key **idx_fk_customer_id** on **customer_id**
- the foreign key **idx_fk_staff_id** on **staff_id**

A full description of these fields is located on MySQL's web site, but the most important fields are below in **bold**.

Column	**Description**
Table	**The table name**

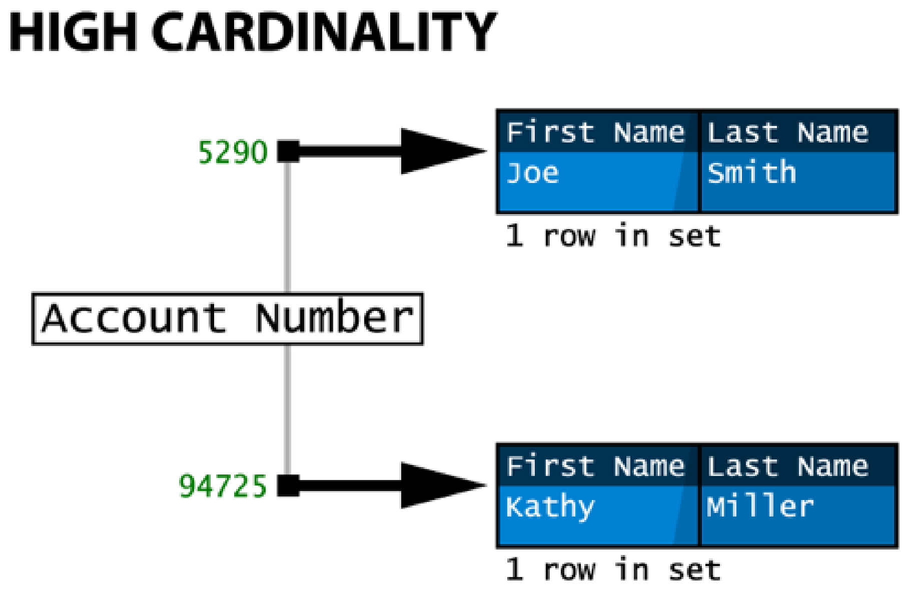
Non_unique	If true, the columns in the index must be uniquely identify a row
Key_name	**The name of the index**
Seq_in_index	For composite indexes (indexes across multiple columns) - the position of the column in the index
Column_name	**The name of the table column**
Collation	The sort of the index (not currently used)
Cardinality	**An estimate of the number of unique rows in the index**
Sub_part	The number of indexed characters, if the column is only partly indexed (perhaps first 5 characters of last name are indexed)
Packed	How the index is packed
Null	If yes, the column may contain NULL
Index_type	The index method (BTREE, FULLTEXT, etc)
Comment	Index comments

Picking good columns to index

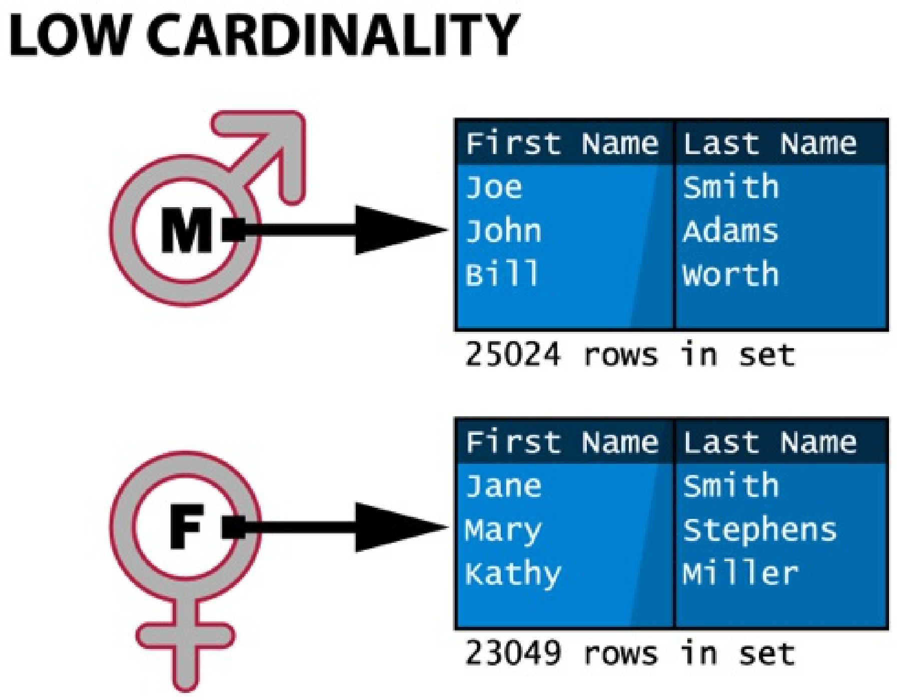
When choosing the columns to index, it is important to consider the data you're trying to index, and how that index will be used.

Ideally you'll choose a column that has a *high cardinality*. An example of a column that has *high cardinality* might be account number, since this number should be unique to one person.

If a column can be used to limit a table from one million rows to a few dozen or so rows, it is a *highly selective* column. A column such as **Postal Code** might be *highly selective*, depending on the rows of the table. A table's *selectivity* can change over time - if many people move to the same **Postal Code** then the **Postal Code** column will no longer be *highly selective*.



Columns that have *low cardinality* are columns like gender. A column like gender would (on average) limit a data set of one million rows to half a million rows. The database engine would still have to work with a large data set when processing a query including this column. Because of this the database engine might choose to ignore your index.



Most databases let you specify *collation* (sort order) for columns in indexes. If you have an index on a column called TransactionDate, and you're always interested in the most recent TransactionDates (like `ORDER BY TransactionDate DESC`), you might want to specify the sort order to be *descending* so the database doesn't have to do extra work to order your query.

Note Unfortunately, as of MySQL 5.0, you can specify a sort order when you create an index but the index created is always ascending. In the future this situation should change.

Managing Indexes

In the previous lesson, we looked at indexes in our sample database. Now let's examine the effects of index changes. To start, let's remove an index from a heavily used table, **rental**, to show the performance of a common query without indexes. Make sure you're logged in as root, and connected to sakila.

Note Our sakila database is very small, so performance will be pretty good even with bad or missing indexes. In the real world you'll see a much greater performance problem.

Type the following at the MySQL prompt:

```
mysql> alter table payment drop FOREIGN KEY fk_payment_rental;
mysql> alter table payment drop index fk_payment_rental;
mysql> alter table rental modify rental_id int not null;
mysql> alter table rental drop primary key;
```

If you typed everything correctly, you'll see the following result:

OBSERVE:

```
mysql> alter table payment drop FOREIGN KEY fk_payment_rental;
Query OK, 16049 rows affected (3.32 sec)
Records: 16049  Duplicates: 0  Warnings: 0

mysql> alter table payment drop index fk_payment_rental;
Query OK, 16049 rows affected (2.53 sec)
Records: 16049  Duplicates: 0  Warnings: 0

mysql> alter table rental modify rental_id int not null;
Query OK, 16044 rows affected (4.32 sec)
Records: 16044  Duplicates: 0  Warnings: 0

mysql> alter table rental drop primary key;
Query OK, 16044 rows affected (4.13 sec)
Records: 16044  Duplicates: 0  Warnings: 0
```

```
mysql>
```

Recall from the previous lesson - the rental table is joined to the payment table on the primary key, rental_id. Let's run a pretty small query to join those bits of information so that we can see the total rentals from August 22nd to August 26th..

Type the following at the MySQL prompt:

```
mysql> select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
FROM rental as R
JOIN payment P on (R.rental_id = P.rental_id)
WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
GROUP BY cast(R.rental_date as date);
```

Whoa! This query performance is terrible!

OBSERVE:

```
mysql> select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
    -> FROM rental as R
    -> JOIN payment P on (R.rental_id = P.rental_id)
    -> WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
    -> GROUP BY cast(R.rental_date as date);
+------------+----------+
| rentaldate | TotalAmt |
+------------+----------+
| 2005-08-22 |  2576.74 |
| 2005-08-23 |  2521.02 |
+------------+----------+
2 rows in set (1 min 6.70 sec)

    mysql>
```

There has to be a better way. Before we implement new indexes, let's examine why the query performed so poorly.

Type the following at the MySQL prompt:

```
mysql> explain select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
   FROM rental as R
   JOIN payment P on (R.rental_id = P.rental_id)
   WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
   GROUP BY cast(R.rental_date as date);
```

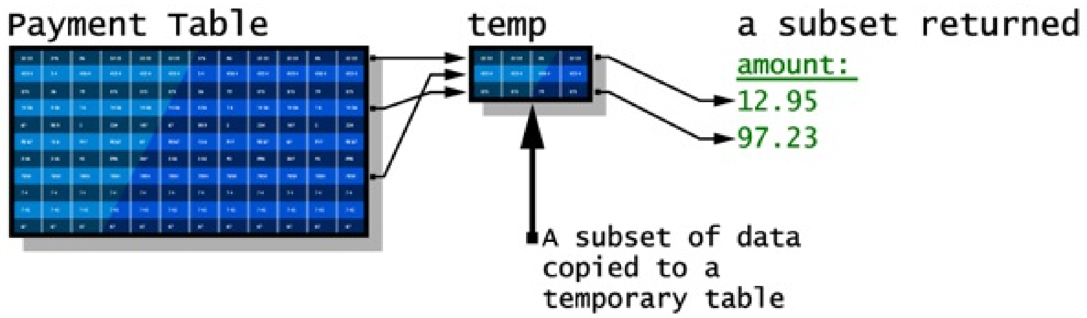
Looks like we didn't give MySQL any assistance for this query, so it has a lot of difficult work to do.

OBSERVE:

```
+----+-------------+-------+------+---------------+------+---------+------+-------+-------------------
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows  | Extra
+----+-------------+-------+------+---------------+------+---------+------+-------+-------------------
|  1 | SIMPLE      | P     | ALL  | NULL          | NULL | NULL    | NULL | 16326 | Using where; Using
|  1 | SIMPLE      | R     | ALL  | NULL          | NULL | NULL    | NULL | 16487 | Using where
+----+-------------+-------+------+---------------+------+---------+------+-------+-------------------
2 rows in set (0.00 sec)
```

MySQL couldn't find an index to use - as noted by the **NULL** under the **key** column. Additionally, there were no **possible_keys** MySQL could have used. MySQL also decided to use **a temporary table** - a temporary copy of some of the table rows, which take a lot of time to generate.

TEMPORARY TABLE



MySQL also had to do a **filesort**, which means it had to do a lot of work to order rows in order to answer the query.

All of those problems mean poor query performance. For a longer description of the `Extra` column, visit MySQL's web site.

Let's take a look at our query to see what we can improve. The first candidates to improve speed are the join columns, and the next would be the items in the WHERE clause.

OBSERVE:

```
select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
FROM rental as R
JOIN payment P on (R.rental_id = P.rental_id)
WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
GROUP BY cast(R.rental_date as date);
```

Let's tackle the join column first Take a look at the indexes defined on the payment and rental tables to see if any contain the column "**rental_id**." For this task we'll use the **INFORMATION_SCHEMA** database maintained by MySQL.

Note The **INFORMATION_SCHEMA** database provides access to all meta data associated with databases, tables, indexes, and procedures. For additional information, check out MySQL's web site.

Type the following at the MySQL prompt:

```
mysql> select * from information_schema.key_column_usage where
table_name in ('payment', 'rental') and column_name = 'rental_id';
```

Looks like neither table has any indexes for this column. If any index did include **rental_id** we would have seen rows returned from our query.

OBSERVE:

```
mysql> select * from information_schema.key_column_usage where
table_name in ('payment', 'rental') and column_name = 'rental_id';
Empty set (0.29 sec)

mysql>
```

We definitely need to add indexes on the rental_id columns in both tables. Recall earlier in this lesson we removed indexes from `payment` and `rental`. Let's add the indexes back now.

The definition of the indexes is from the file `sakila-schema.sql`. If you take a look at that file, you'll see the following definition for `rental` and `payment` starting at line 234.

OBSERVE:

```
--
-- Table structure for table `payment`
--

CREATE TABLE payment (
  payment_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  customer_id SMALLINT UNSIGNED NOT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  rental_id INT DEFAULT NULL,
  amount DECIMAL(5,2) NOT NULL,
  payment_date DATETIME NOT NULL,
  last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (payment_id),
  KEY idx_fk_staff_id (staff_id),
  KEY idx_fk_customer_id (customer_id),
  CONSTRAINT fk_payment_rental FOREIGN KEY (rental_id) REFERENCES rental (rental_id) ON DELETE SET NUL
  CONSTRAINT fk_payment_customer FOREIGN KEY (customer_id) REFERENCES customer (customer_id) ON DELETE
  CONSTRAINT fk_payment_staff FOREIGN KEY (staff_id) REFERENCES staff (staff_id) ON DELETE RESTRICT ON
)ENGINE=InnoDB DEFAULT CHARSET=utf8;


--
-- Table structure for table `rental`
--

CREATE TABLE rental (
  rental_id INT NOT NULL AUTO_INCREMENT,
  rental_date DATETIME NOT NULL,
  inventory_id MEDIUMINT UNSIGNED NOT NULL,
  customer_id SMALLINT UNSIGNED NOT NULL,
  return_date DATETIME DEFAULT NULL,
  staff_id TINYINT UNSIGNED NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (rental_id),
  UNIQUE KEY  (rental_date,inventory_id,customer_id),
  KEY idx_fk_inventory_id (inventory_id),
  KEY idx_fk_customer_id (customer_id),
  KEY idx_fk_staff_id (staff_id),
  CONSTRAINT fk_rental_staff FOREIGN KEY (staff_id) REFERENCES staff (staff_id) ON DELETE RESTRICT ON
  CONSTRAINT fk_rental_inventory FOREIGN KEY (inventory_id) REFERENCES inventory (inventory_id) ON DEL
  CONSTRAINT fk_rental_customer FOREIGN KEY (customer_id) REFERENCES customer (customer_id) ON DELETE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Earlier in the lesson we used four alter statements on the `payment` and `rental` tables:

OBSERVE:

```
mysql> alter table payment drop FOREIGN KEY fk_payment_rental;
mysql> alter table payment drop index fk_payment_rental;
mysql> alter table rental modify rental_id int not null;
mysql> alter table rental drop primary key;
```

We'll transform those four alter statements, which removed indexes from `payment` and `rental`, into three new alter statements, which will restore those indexes. The previous lines in **red**, **blue** and **green** refer to lines in the alter statements below.

Type the following at the MySQL prompt:

```
alter table rental add primary key(rental_id);
alter table rental modify rental_id int not null AUTO_INCREMENT;
alter table payment add constraint fk_payment_rental FOREIGN KEY (rental_id)
 REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE;
```

If everything went well you'll see the following:

OBSERVE:

```
mysql> alter table rental add primary key(rental_id);
Query OK, 16044 rows affected (4.07 sec)
Records: 16044  Duplicates: 0  Warnings: 0

mysql> alter table rental modify rental_id int not null AUTO_INCREMENT;
Query OK, 16044 rows affected (4.54 sec)
Records: 16044  Duplicates: 0  Warnings: 0

mysql> alter table payment add constraint fk_payment_rental FOREIGN KEY (rental_id)
    ->  REFERENCES rental (rental_id) ON DELETE SET NULL ON UPDATE CASCADE;
Query OK, 16049 rows affected (4.12 sec)
Records: 16049  Duplicates: 0  Warnings: 0

mysql>
```

With those indexes back in place, let's try the query again.

Type the following at the MySQL prompt:

```
mysql> select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
FROM rental as R
JOIN payment P on (R.rental_id = P.rental_id)
WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
GROUP BY cast(R.rental_date as date);
```

Whoa! These indexes made a difference, since the query time is now much shorter.

OBSERVE:

```
mysql> select cast(R.rental_date as date) as rentaldate, sum(P.amount) as TotalAmt
    -> FROM rental as R
    -> JOIN payment P on (R.rental_id = P.rental_id)
    -> WHERE P.payment_date BETWEEN '2005-08-22' and '2005-08-26'
    -> GROUP BY cast(R.rental_date as date);
+------------+----------+
| rentaldate | TotalAmt |
+------------+----------+
| 2005-08-22 |  2576.74 |
| 2005-08-23 |  2521.02 |
+------------+----------+
2 rows in set (0.11 sec)

mysql>
```

This speed-up is huge! Imagine if this query was used tens, or even hundreds of times per day - your users would be very happy to see such a large improvement in performance.

In the previous steps we did the following:

1. Ran the query to see how long it would take to execute.
2. Used `EXPLAIN` to see how MySQL was trying to answer the query.
3. Guessed that adding an index on the join columns would improve performance.
4. Checked `INFORMATION_SCHEMA` to see if an index existed on the join columns.
5. Implemented indexes on the join columns.
6. Ran the query again to see if performance was improved.

Tip When working with indexes, it is important to perform tests on a different computer than your production environment. Changing indexes on large databases can take hours, and users may not be able to work with tables until your index statements are complete. You don't want to take down your production system, especially if you are adding or removing indexes that may or may not improve performance!

Managing indexes is only one aspect of database maintenance. In the next lesson we'll examine additional things administrators need to do to keep databases operating at optimal efficiency. See you then!

Maintaining Databases
DBA 2: Administering MySQL Lesson 7

Databases are not static entities. For most applications, rows are continuously being added, updated, and removed from tables. The types of queries presented to the database often evolves. In the previous lesson we learned how we maintain proper indexes on tables to keep everything performing well as queries and data change.

What happens when queries don't change, but performance starts to suffer? What if our disk develops problems, and we need to validate the integrity of our tables? We'll look into these issues in this lesson.

Analyzing Tables

When MySQL processes a query with a join. it must make decisions on how to use indexes, how to read tables, and how to join the tables together in order to produce the desired result in the shortest time possible. MySQL keeps track of data distributions on indexes in order to make these decisions fast.

Make sure you're logged into the Unix Terminal and you're MySQL is running. Log into the database as root, and connect to the **sakila** database.

A common query for our rental store would be to view sales by staff. The very nature of this query seems to indicate the database will have to read the whole *payment* table and the whole *sales* table. We might write the query this way:

Type the following at the MySQL prompt:

```
mysql> select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
GROUP BY s.first_name, s.last_name;
```

Running that query does in fact give us the desired results.

OBSERVE:

```
mysql> select s.first_name, s.last_name, sum(p.amount)
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> GROUP BY s.first_name, s.last_name;
+------------+-----------+---------------+
| first_name | last_name | sum(p.amount) |
+------------+-----------+---------------+
| Jon        | Stephens  |      33927.04 |
| Mike       | Hillyer   |      33489.47 |
+------------+-----------+---------------+
2 rows in set (0.53 sec)

mysql>
```

Before we take a look at the execution plan, let's see how many rows are in the tables.

Type the following at the MySQL prompt:

```
mysql> select 'staff' as Tbl, count(*) as RowCount from staff
union
select 'Payment', count(*) from payment;
```

As long as you typed everything correctly MySQL will give us its answer:

OBSERVE:

```
mysql> select 'staff' as Tbl, count(*) as RowCount from staff
    -> union
    -> select 'Payment', count(*) from payment;
+---------+----------+
| Tbl     | RowCount |
+---------+----------+
| staff   |        3 |
| Payment |    16049 |
+---------+----------+
2 rows in set (0.03 sec)
```

```
mysql>
```

This is consistent with our application - we don't change staff often, but we record a new row in *payment* for a sale. So how exactly will MySQL join these tables? It could probably choose one of two methods:

1. Read each row in staff, looking for a matching row in payment
2. Read each row in payment, looking for a matching row in staff

Since the staff table only has three entries, it seems the first scenario would be the fastest. Let's see what MySQL thinks.

Type the following at the MySQL prompt:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
GROUP BY s.first_name, s.last_name;
```

Sure enough, it picks *staff* first, then joins it on *payment*. Looks like it ignores the index on *staff_id* as well.

OBSERVE:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> GROUP BY s.first_name, s.last_name;
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
| id | select_type | table | type | possible_keys   | key             | key_len | ref                |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
|  1 | SIMPLE      | s     | ALL  | PRIMARY         | NULL            | NULL    | NULL               |
|  1 | SIMPLE      | p     | ref  | idx_fk_staff_id | idx_fk_staff_id | 1       | sakila.s.staff_id |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
  2 rows in set (0.26 sec)
```

This result seems a little strange - it shows **4070** rows for the payment table, however we know that there are many more rows in the payment table. Recall from the previous lesson that this count isn't exactly the number of rows in the table - instead it is the number of rows MySQL believes it must examine in order to fulfil the query.

Note Your value for rows may be different than listed above, and that is okay. MySQL may already have updated statistics.

This number still might be off. In order to give MySQL some extra help when it plans its query, we can tell it to **analyze** the tables. The syntax for this command is straightforward.

Type the following at the MySQL prompt:

```
mysql> analyze table staff;
```

As long as you typed everything correctly, MySQL will return with a success message.

OBSERVE:

```
mysql> analyze table staff;
+--------------+---------+----------+----------+
| Table        | Op      | Msg_type | Msg_text |
+--------------+---------+----------+----------+
| sakila.staff | analyze | status   | OK       |
+--------------+---------+----------+----------+
1 row in set (0.10 sec)
```

Let's see if this had any effect on our query plan.

Type the following at the MySQL prompt:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
GROUP BY s.first_name, s.last_name;
```

It doesn't look like there was any change.

OBSERVE:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> GROUP BY s.first_name, s.last_name;
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
| id | select_type | table | type | possible_keys   | key             | key_len | ref                |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
|  1 | SIMPLE      | s     | ALL  | PRIMARY         | NULL            | NULL    | NULL               |
|  1 | SIMPLE      | p     | ref  | idx_fk_staff_id | idx_fk_staff_id | 1       | sakila.s.staff_id |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
2 rows in set (0.00 sec)
```

Let's analyze the payments table now.

Type the following at the MySQL prompt:

```
mysql> analyze table payment;
```

Again you'll see a familiar message from MySQL.

OBSERVE:

```
mysql> analyze table payment;
+----------------+---------+----------+----------+
| Table          | Op      | Msg_type | Msg_text |
+----------------+---------+----------+----------+
| sakila.payment | analyze | status   | OK       |
+----------------+---------+----------+----------+
1 row in set (0.03 sec)
```

Lets check the query plan once again.

Type the following at the MySQL prompt:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
GROUP BY s.first_name, s.last_name;
```

This time there was a change - the rows for *payment* are now significantly lower.

OBSERVE:

```
mysql> explain select s.first_name, s.last_name, sum(p.amount) as TotalPayments
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> GROUP BY s.first_name, s.last_name;
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
| id | select_type | table | type | possible_keys   | key             | key_len | ref                |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
|  1 | SIMPLE      | s     | ALL  | PRIMARY         | NULL            | NULL    | NULL               |
|  1 | SIMPLE      | p     | ref  | idx_fk_staff_id | idx_fk_staff_id | 1       | sakila.s.staff_id |
+----+-------------+-------+------+-----------------+-----------------+---------+--------------------+-
2 rows in set (0.00 sec)
```

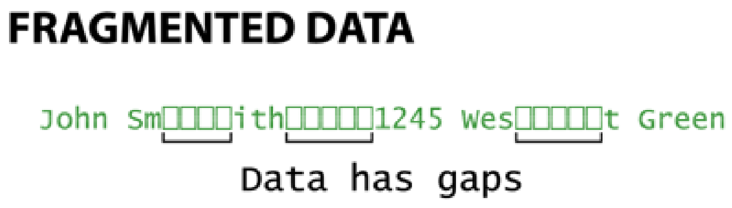
To be fair - it is unlikely this example will show any noticeable impact on query performance. On much larger tables, however, this difference can be quite substantial.

Optimizing Tables

MySQL is a very fast database, and it usually does a good job answering queries in the shortest amount of time possible. Under normal database activity you won't have to worry about optimizing tables, but there are times when you'll have to do a lot of inserts, updates, or deletes.

Occasionally you may want to import records from another operational system to produce unified reports. You may have to update a lot of records in your system to fix periodic bugs. Depending on your business, you may decide to delete (or at least archive) transactions older than a certain period of time such as five years.

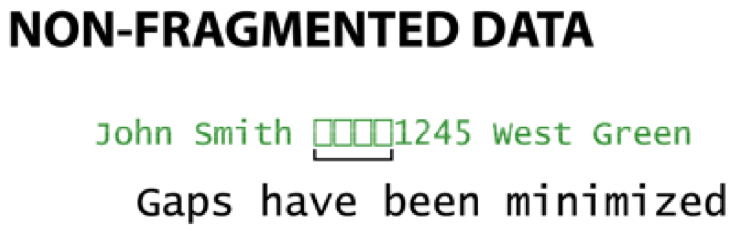
MySQL is typically "lazy" when it comes to these massive operations. In order to provide the best performance *right now* it often does a minimum amount of work. This means table structures may not be in optimal condition if many rows are added, updated, or deleted. Tables can become *fragmented*.



You can force MySQL to rebuild its table and index structures by using the **optimize table** command. This command performs three steps:

1. Optimizes deleted or split rows
2. Sorts indexes that are not sorted
3. Updates table statistics

The result of optimization is a table that is no longer fragmented (or has a minimum amount of fragmentation).



Note Optimizing tables can be a lengthy process, and running the optimize command locks other people out of the table. It should only be done during normal maintenance times.

Before we optimize our tables, lets run a simple query to get a baseline.

Type the following at the MySQL prompt:

```
select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
WHERE s.staff_id=1
GROUP BY s.first_name, s.last_name;
```

For this run, you might see the result in a fairly short amount of time.

OBSERVE:

```
mysql> select s.first_name, s.last_name, sum(p.amount) as TotalPayments
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> WHERE s.staff_id=1
    -> GROUP BY s.first_name, s.last_name;
+------------+-----------+---------------+
| first_name | last_name | TotalPayments |
+------------+-----------+---------------+
| Mike       | Hillyer   |      33489.47 |
+------------+-----------+---------------+
1 row in set (0.15 sec)
```

Our baseline time is 0.15 sec. Now lets try the **optimize** command.

Type the following at the MySQL prompt:

```
mysql> optimize table staff;
```

As long as you typed everything correctly you'll see a positive response from MySQL.

OBSERVE:

```
mysql> optimize table staff;
+--------------+----------+----------+----------+
| Table        | Op       | Msg_type | Msg_text |
+--------------+----------+----------+----------+
| sakila.staff | optimize | status   | OK       |
+--------------+----------+----------+----------+
1 row in set (0.34 sec)
```

While we are at it, lets optimize the payment table as well.

Type the following at the MySQL prompt:

```
mysql> optimize table payment;
```

You should see a similar status message as long as you typed everything correctly. This table is larger, so it will take longer to optimize.

OBSERVE:

```
mysql> optimize table payment;
+----------------+----------+----------+----------+
| Table          | Op       | Msg_type | Msg_text |
+----------------+----------+----------+----------+
| sakila.payment | optimize | status   | OK       |
+----------------+----------+----------+----------+
1 row in set (3.42 sec)
```

With both of those tables optimized, we can run our query again to see if it had any impact.

Type the following at the MySQL prompt:

```
select s.first_name, s.last_name, sum(p.amount) as TotalPayments
from  payment p
INNER join staff s on (s.staff_id=p.staff_id)
WHERE s.staff_id=1
GROUP BY s.first_name, s.last_name;
```

Looks like we have a marginal improvement.

OBSERVE:

```
mysql> select s.first_name, s.last_name, sum(p.amount) as TotalPayments
    -> from  payment p
    -> INNER join staff s on (s.staff_id=p.staff_id)
    -> WHERE s.staff_id=1
    -> GROUP BY s.first_name, s.last_name;
+------------+-----------+---------------+
| first_name | last_name | TotalPayments |
+------------+-----------+---------------+
| Mike       | Hillyer   |      33489.47 |
+------------+-----------+---------------+
   1 row in set (0.14 sec)
```

This shows a tiny improvement, however the difference is so small that `optimize table` likely had little or no effect.

Note You're query may show a slightly longer query time after you run the `optimize table` statement. In this case your table was likely already in pretty good shape. `optimize table` won't harm your table in any way, or cause you to experience longer query times in the long run.

To see a significant improvement in query performance you'll need to have a heavily abused table. In this case the `optimize` command may take tens of minutes (or even hours) to complete.

Other commands

MySQL has several other commands you can use to maintain tables:

- CHECK TABLE - Checks tables for errors
- CHECKSUM TABLE - Calculates a checksum on a table

- REPAIR TABLE - Repairs table errors

It is unlikely you'll have to use these commands in normal database usage unless you begin to experience hardware problems, or if you are testing a beta version of MySQL.

Further information on these commands is available on MySQL's web site, MySQL.com.

You learned two important ways to maintain tables in your databases. In the next lesson you'll learn two more important topics - how to backup and restore your database. See you then!

Backups & Restores
DBA 2: Administering MySQL Lesson 8

Welcome back! In the last lesson we learned a few useful techniques to maintain our databases. In this lesson we'll learn the most important type of maintenance: backups and restores.

Backups

Good systems administrators have a regular schedule for backing up computers. Good database administrators should also have regular schedules for backing up databases. If you don't have a backup, you won't be able to recover from hardware failures, accidental updates, or accidental deletes.

Most databases support two types of backups - **full** backups and **transactional** backups.

Full backups make a complete copy of the database at a given point in time, and are at least the same size as the database (or table) you are backing up.

Transactional backups only contain the changes made to a database (or table) since the last full backup.

By default MySQL doesn't enable a binary log, so it isn't possible to perform transactional backups. The binary log is an extra file where MySQL saves every change made to the database (as well as making the change on the database). It is like keeping a notebook where you record changes to your address book, in addition to updating your friends phone numbers in your address book.

Backups with MySQL are usually fairly fast for most databases, so MySQL administrators don't usually worry about performing transactional backups.

Note For additional information on binary logs, check out MySQL's web site.

MySQL stores its data in files computer, so it seems like it should be very easy to backup data: just copy MySQL's data directory to a separate backup location. Unfortunately it isn't quite so simple. MySQL is very protective over its data files, which, in an active database, are constantly being updated and searched. If you copy a file that MySQL is updating, can you be sure that your copy is valid?

In order to copy MySQL's data files, you would need to:

1. tell MySQL to disallow updates to your tables (by *locking* the tables)
2. copy the data files
3. then tell MySQL to allow updates again (by *unlocking* the tables)

Not performing these steps could result in an incomplete backup.

Fortunately MySQL ships with two programs to make backups easier.

Using mysqldump

The easiest, and most flexible backup program is called `mysqldump`. It can create a SQL script that represents the structure and data contained in your database. It can also create an CSV, delimited, or XML text file for your database and data. It isn't the fastest way to backup your database, however it is often plenty fast. Since you are not touching MySQL's data files directly, you don't have to worry about locking or unlocking anything.

Let's try it! First, make sure your MySQL server is running. We'll run the **mysqldump** program and backup the **sakila** database, and use the standard Unix output redirection - **>** - to save to a backup file called **backup.sql**.

Type the following at the Unix prompt:

```
cold:~$ mysqldump -u root -p sakila > backup.sql
```

At the prompt type your **root** password. As long as you typed everything correctly you'll see no results.

OBSERVE:

```
cold:~$ mysqldump -u root -p sakila > backup.sql
Enter password:
cold:~$
```

To see the contents of the backup file, we'll use the **head** command so that we can see the beginning of the file.

Type the following at the Unix prompt:

```
cold:~$ head backup.sql -n 30
```

The first few lines are comments and commands that prepare MySQL in the event the script is used to restore your database. Next is the table structure.

OBSERVE:

```
-- MySQL dump 10.11
--
-- Host: localhost    Database: sakila
-- ------------------------------------------------------
-- Server version       5.0.41-OREILLY

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

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

DROP TABLE IF EXISTS `actor`;
CREATE TABLE `actor` (
`actor_id` smallint(5) unsigned NOT NULL auto_increment,
`first_name` varchar(45) NOT NULL,
`last_name` varchar(45) NOT NULL,
`last_update` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
PRIMARY KEY  (`actor_id`),
KEY `idx_actor_last_name` (`last_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The first several lines are comments that tell us **the version of mysqldump (10.11)**, the **host and database**, the version of the **mysql server**, and some **MySQL-specific settings** to help load the file into a different MySQL server. The next lines **drop and recreate** the **actor** table.

Not shown in the first 30 lines is data for the tables. Open the file **backup.sql** in your favorite text editor if you want to see the rest of its contents.

Before we continue, let's check the size of the backup file. We can use the **du** command to accomplish this.

Type the following at the Unix prompt:

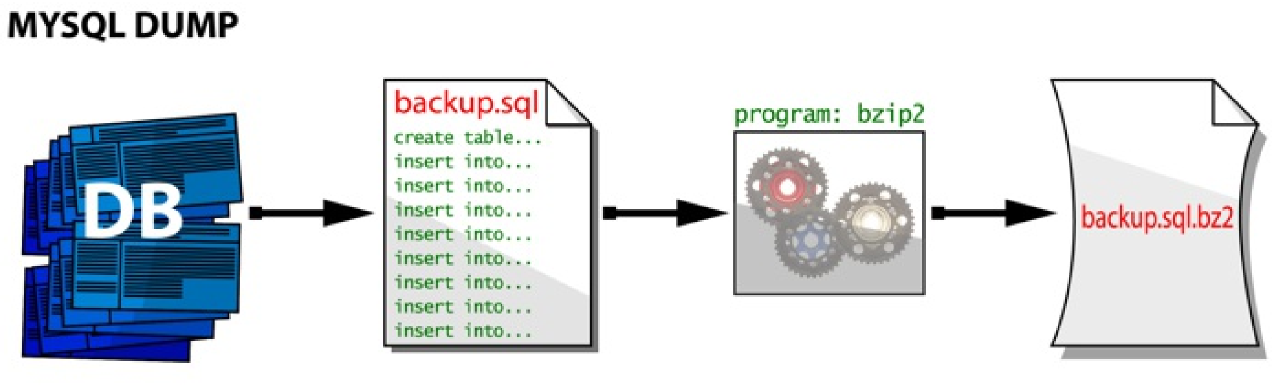
```
cold:~$ du -h backup.sql
```

Looks like the file is around 3.2 MB:

OBSERVE:

```
cold:~$ du -h backup.sql
3.2M    backup.sql
cold:~$
```

This text file contains a lot of redundant data. To reduce space, it would make sense to compress this file as it is being created. In the last example we used output redirection - > - to save **mysqldump**'s output to a file. This time we can use a different type of output redirection - **|** - the *pipe* - to send the backup data to the **bzip2** program, which will compress the data and save it to a file. Note that it will take a bit longer to run this command than the one executed previously.



Type the following at the Unix prompt:

```
cold:~$ mysqldump -u root -p sakila | bzip2 > backup.sql.bz2
```

Just like last time, you won't see any messages unless you typed something incorrectly. Let's compare the file sizes.

Type the following at the Unix prompt:

```
cold:~$ du -h backup.*
```

Looks like the compressed file is only 16% the size of the original.

OBSERVE:

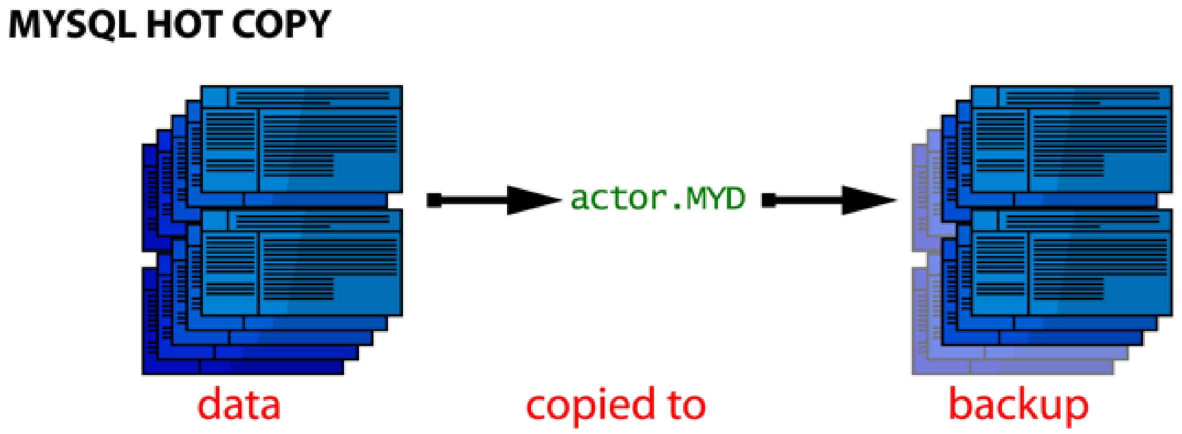
```
cold:~$ du -h backup*
3.2M    backup.sql
499K    backup.sql.bz2
cold:~$
```

Using mysqlhotcopy

Another tool you can use to backup MySQL databases is **mysqlhotcopy**. This script is very fast, but does have a couple of drawbacks:

- It only works with MyISAM tables
- It must be run on the same machine as the database server

mysqlhotcopy works by locking each table in the database, copying the physical files (like *actor.MYD*) from the database server's data directory to the backup directory, then unlocking the table. This is a very fast way to make a complete backup of the database. The reason it is so fast is no updates are allowed during the copy, so the computer can focus completely on copying the data files.



Let's copy the sakila database using this method.

Currently `mysqlhotcopy` does not prompt for a password. You need to either specify your password on the command line or specify it in your `.my.cnf` file. If you plan on using mysqlhotcopy for production backups you should put your password in the `.my.cnf` file. Just edit the line for **password** and add a line for **user**:

OBSERVE:

```
[client]
user = username
password = password
```

As you type the command below, make sure you change the word **password** to your actual password. The **tilde ~** tells **mysqlhotcopy** that we want to backup the **sakila** database to our home directory. This means that mysqlhotcopy will actually create a new directory called **sakila** in our home directory on the cold server.

Type the following at the Unix prompt:

```
cold:~$ ./mysql/bin/mysqlhotcopy -S ~/mysql/data/mysql.sock -u root -p password sakila ~
```

`mysqlhotcopy` creates a lot of output when it runs:

OBSERVE:

```
cold:~$ ./mysql/bin/mysqlhotcopy -S ~/mysql/data/mysql.sock -u root -p password sakila ~
Locked 24 tables in 0 seconds.
Flushed tables (`sakila`.`actor`, `sakila`.`actor_info`, `sakila`.`address`, `sakila`.`category
Copying 37 files...
Copying indices for 0 files...
Unlocked tables.
mysqlhotcopy copied 24 tables (37 files) in 0 seconds (0 seconds overall).
cold:~$
```

If a directory called *sakila* already existed in your directory, `mysqlhotcopy` would fail with this message:

OBSERVE:

```
cold:~$ ./mysql/bin/mysqlhotcopy -S ~/mysql/data/mysql.sock -u root -p password sakila ~
Can't hotcopy to '/users/certjosh/sakila' because directory
already exist and the --allowold or --addtodest options were not given.
cold:~$
```

Should this happen to you, remove the *sakila* directory by using the **rm** command. The **-rf** option tells **rm** to force the removal of everything in the sakila directory and any sub directories inside of the sakila directory.

Type the following at the Unix prompt:

```
cold:~$ rm -rf sakila
```

You won't see any output from the previous command.

All 37 files for the sakila database were copied one by one to a directory called **sakila**. Let's take a look at the contents of that directory.

Type the following at the Unix prompt:

```
cold:~$ ls sakila
```

Sure enough, all files for all of the tables have been copied.

OBSERVE:

```
cold:~$ ls sakila
actor.frm                    customer_list_limited.frm    film_text.frm
actor_info.frm               db.opt                       ins_film.TRN
address.frm                  del_film.TRN                 inventory.frm
category.frm                 film.TRG                     language.frm
city.frm                     film.frm                     nicer_but_slower_film_list.frm
country.frm                  film_actor.frm               payment.TRG
customer.TRG                 film_category.frm            payment.frm
customer.frm                 film_list.frm                payment_date.TRN
customer_create_date.TRN     film_text.MYD                rental.TRG
customer_list.frm            film_text.MYI                rental.frm
```

```
cold:~$
```

Restores

Usually forgotten until an emergency strikes, restores are an important part of every backup routine. Accidents happen - tables are sometimes deleted. Update statements are accidentally run without a `WHERE` clause. If you have good backups, and are confident you can restore from those backups, you won't have to worry so much about data loss.

Note If you have the hardware, it is best to practice database restores on another server. After all - what would you do if your computer developed serious hardware problems?

Restores from SQL files

If you choose to perform your backups using `mysqldump`, you have a lot of flexibility on how and where you restore your data. You can restore to a different server, or even restore a small subset of the data you backed up. This is because the files generated by `mysqldump` are just plain old text files. You can edit the files by hand, transfer them to different machines, or even print them if you want.

The program used to restore SQL files is the same `mysql` program we use for general querying. We can use the standard unix method of input redirection to tell `mysql` to read the .sql files. Let's try it!

Type the following at the Unix prompt:

```
cold:~$ mysql -u root -p sakila < backup.sql
```

If you typed the command and your password correctly, you'll see no response from MySQL.

OBSERVE:

```
cold:~$ mysql -u root -p sakila < backup.sql
Enter password:
cold:~$
```

Don't fear - all of the commands in the file were executed.

A little extra work is required to restore from a compressed archive. To do this we need to use a program called `bunzip2` and the standard unix pipe redirection. This command will take longer to execute since the computer must decompress the file before MySQL can process it.

Type the following at the Unix prompt:

```
cold:~$ bunzip2 -c backup.sql.bz2 | mysql -u root -p sakila
```

Again, you won't see any results from MySQL unless there is an error.

OBSERVE:

```
cold:~$ bunzip2 -c backup.sql.bz2 | mysql -u root -p sakila
Enter password:
cold:~$
```

That is it! Remember the uncompressed backup files in this case are just plain text, so you can edit unwanted tables or data.

Restores from mysqlhotcopy

Restoring from a mysqlhotcopy can be a more involved process, especially if you are only restoring one database. Restoring a whole server is straightforward - stop MySQL, copy the data files to the data directory, check the data tables, and then restart MySQL

Let's practice restoring from the backup we created earlier in this lesson. Before we can do anything we'll have to stop our MySQL server.

Type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p shutdown
```

As long as you typed everything correctly you'll see a message with the words `STOPPING server` on your screen.

Now that the server has stopped, let's navigate to the directory where MySQL stores its data.

Type the following at the Unix prompt:

```
cold:~$ cd ~/mysql/data
```

MySQL creates a directory for each database. We can check to make sure it exists for **sakila**:

Type the following at the Unix prompt:

```
cold:~$ ls -d sakila
```

Sure enough, the directory exists.

OBSERVE:

```
cold:~/mysql/data$ ls -d sakila
sakila/
cold:~/mysql/data$
```

In order to restore sakila, we'll have to move (or rename) the current sakila directory using the **mv** command. Let's do this now.

Type the following at the Unix prompt:

```
cold:~/mysql/data$ mv sakila sakila.SAVED
```

You won't get any messages from the `mv` command unless something went wrong. With the old database directory out of the way, we can move our backup copy to its place.

Type the following at the Unix prompt:

```
cold:~/mysql/data$ mv ~/sakila sakila
```

Again, you won't see any messages unless there was a problem. Now with our previous data restored, and before we start the server, we need to check the database for any corruption. To do this we'll use the **myisamchk** utility. Normally **myisamchk** produces a lot of output, so we'll use the **--silent** option so it will only return a message if there is an error. **myisamchk** operates on the files themselves, and only on the database files, so we need to tell it to only check files in the sakila directory with an extension of **MYI**.

Note For a description of **MYI** files, refer back to lesson 2.

Type the following at the Unix prompt:

```
cold:~/mysql/data$ myisamchk --silent sakila/*.MYI
```

Since our backup completed without errors, there shouldn't be any errors with our database files.

OBSERVE:

```
cold:~/mysql/data$ myisamchk --silent sakila/*.MYI
cold:~/mysql/data$
```

Now we are ready to restart the server! We'll use the same command we've used through the course.

Type the following at the Unix prompt:

```
cold:~/mysql$ cd ~/mysql ;bin/mysqld_safe --defaults-file=/users/certjosh/mysql/my.cnf &
```

Since the tables checked out OK, MySQL should start without problems.

OBSERVE:

```
cold:~/mysql$ Starting mysqld daemon with databases from /users/certjosh/mysql/data

cold:~/mysql$
```

To be sure, let's run a quick query against the `staff` table.

Type the following at the Unix prompt:

```
cold:~/mysql$ mysql -u root -p sakila -e "select first_name, last_name from staff"
```

The **-e** options tells MySQL that we are only interested in running **one command**, which follows in quotes "". Looks like our restore was successful!

OBSERVE:

```
cold:~/mysql$ mysql -u root -p sakila -e "select first_name, last_name from staff"
Enter password:
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
| Jon        | Stephens  |
| Dave       | Smith     |
+------------+-----------+
cold:~/mysql$
```

In this lesson you've learned how to backup and restore your databases. In the next lesson we'll examine a topic similar to backups and restores - bulk imports and exports. See you then!

Bulk Exports and Imports
DBA 2: Administering MySQL Lesson 9

Welcome back! In the last lesson we learned two important tasks for database maintenance - backups and restores. Backups are related to *exports* - both pull data from a database and put it elsewhere. Restores are related to *imports* - both take data from outside the database and put it inside.

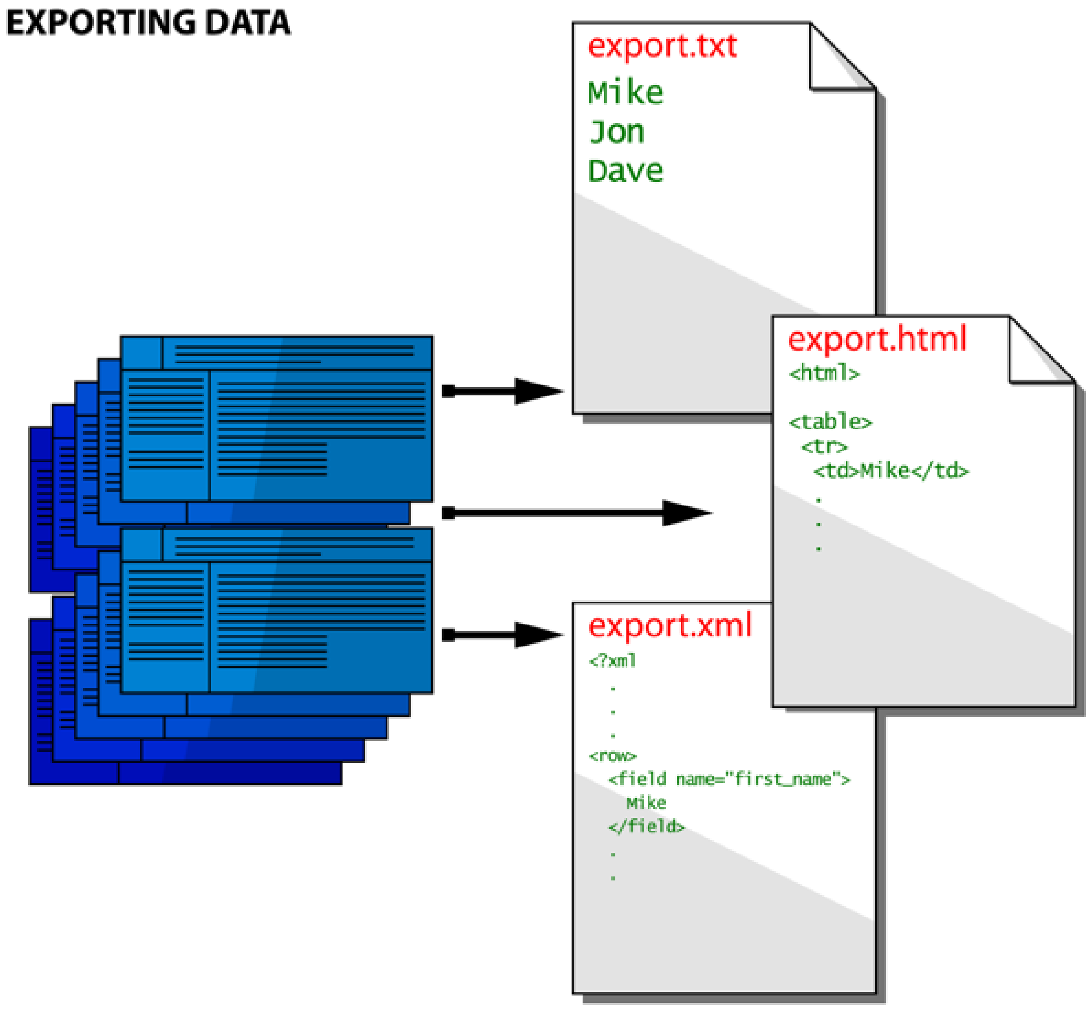
Backups and restores are a part of scheduled database maintenance, whereas imports and exports are typically "one time" ways to get a set of data in or out of your database. An import would be used when your company has just purchased some market research data, and you need to incorporate that into your database. An export would be used when your coworker needs some sales data for analysis using Excel.

Whichever task you are trying to accomplish, MySQL has a tool you can use to get the job done.

Exports

The first way you can export data from MySQL is by using the regular `mysql` command. It has several options for exporting data in different formats:

- Text (usually tab delimited)
- HTML
- XML



Exporting a query as plain text is very straightforward. The first method uses the standard `SELECT` statement with a special **INTO OUTFILE** clause. One downside to this command is that it creates a text file on the database server, not your client machine. Creating a file on the database server could be dangerous, so this clause is not allowed for most database users.

To allow a user access to the `SELECT ...` **INTO OUTFILE** statement, grant the user the `FILE` permission.

Since we are the database administrator we have enough privileges to use this command. Let's try it! Make sure your database server is running, and log into MySQL as root. Connect to the sakila database.

Type the following at the MySQL prompt:

```
mysql> select first_name, last_name FROM staff INTO OUTFILE 'staff.data';
```

If you typed everything correctly, you'll see the following results.

OBSERVE:

```
mysql> select first_name, last_name FROM staff INTO OUTFILE 'staff.data';
Query OK, 3 rows affected (0.06 sec)

mysql>
```

Let's take a look at the contents of the file. To view the contents, we'll quit MySQL and use the standard Unix `cat` command.

Type the following at the Unix prompt:

```
cold:~$ cat ~/mysql/data/sakila/staff.data
```

Columns in the file are actually delimited by tabs, so programs like Excel should be able to read the file without problems. The file should look something like this:

OBSERVE:

```
cold:~$ cat ~/mysql/data/sakila/staff.data
Mike    Hillyer
Jon     Stephens
Dave    Smith
cold:~$
```

A different way to export query data as mostly plain text is to use an argument to the mysql command itself. The **`-B`** or **BATCH** option does just that. We can combine it with the `-e` option we used in the last lesson and the standard Unix redirection to create a text file. Recall the `-e` option tells mysql to execute one command, which we provide in quotes.

Type the following at the Unix prompt:

```
cold:~$ mysql -B -u root -p sakila  -e "select first_name, last_name from staff" > staff.data.txt
```

You won't see any results unless there was an error. We can use the `cat` command to see the contents of the file we just created.

Type the following at the Unix prompt:

```
cold:~$ cat staff.data.txt
```

This output file differs a bit from the last export - the first line of this file contains column names. However just like the last time, this file is tab delimited.

OBSERVE:

```
cold:~$ cat staff.data.txt
first_name      last_name
Mike    Hillyer
Jon     Stephens
Dave    Smith
cold:~$
```

If your application requires HTML, you can use the **`-H`** option instead of the `-B` option to MySQL. Let's try it now!

Type the following at the Unix prompt:

```
cold:~$ mysql -H -u root -p sakila  -e "select first_name, last_name from staff" > staff.data.html
```

Again, you won't see any messages unless there was an error, and we can still use `cat` to view the contents of the file. You could also view the HTML file in a web browser if you want.

Type the following at the Unix prompt:

```
cold:~$ cat staff.data.html
```

It isn't a complete HTML document, but it is well-formed.

OBSERVE:

```
cold:~$ cat staff.data.html
 <TABLE BORDER=1><TR><TH>first_name</TH><TH>last_name</TH></TR><TR><TD>Mike</TD><TD>Hillyer</TD></TR><
```

If you need a complete XML document instead of an HTML fragment, you can use the **-X** option. The resulting output can be transformed using XSLT or parsed using any of the standard XML libraries.

Let's give this a try!

Type the following at the Unix prompt:

```
cold:~$ mysql -X -u root -p sakila  -e "select first_name, last_name from staff" > staff.data.xml
```

You could view the XML file in a web browser, however we'll continue to view it using `cat`.

Type the following at the Unix prompt:

```
cold:~$ cat staff.data.xml
```

The results are certainly longer than the HTML:

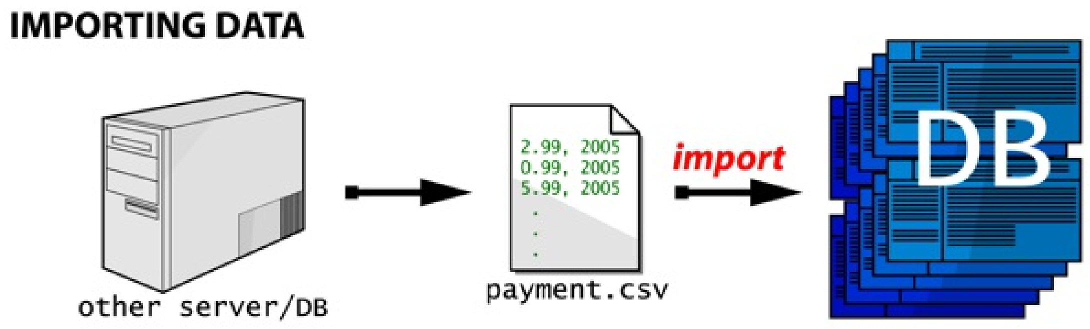
OBSERVE:

```
cold:~$ cat staff.data.xml
<?xml version="1.0"?>

<resultset statement="select first_name, last_name from staff
" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="first_name">Mike</field>
        <field name="last_name">Hillyer</field>
  </row>

  <row>
        <field name="first_name">Jon</field>
        <field name="last_name">Stephens</field>
  </row>

  <row>
        <field name="first_name">Dave</field>
        <field name="last_name">Smith</field>
  </row>
</resultset>
cold:~$
```

In these examples we used a simple query to demonstrate the various ways we can export data from MySQL. However there is no reason why we couldn't have used a complex query, queries, or even a stored procedure.

Imports

Many times you'll need to import data (usually generated from other computer systems) stored in flat files into your database.



Before we learn how to import data, we'll create a temporary table for our experimentation. Make sure you are logged into MySQL as root, and connected to the sakila database.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE payment_import
(
  amount decimal(5,2),
  payment_date timestamp
);
```

As long as you typed everything correctly you'll see the following:

OBSERVE:

```
mysql> CREATE TABLE payment_import
    -> (
    -> amount decimal(5,2),
    -> payment_date timestamp
    -> );
Query OK, 0 rows affected (0.27 sec)

mysql>
```

Next, let's download a comma delimited file we can import into our test table. The file can be downloaded from O'Reilly's server. We'll use the `wget` command to retrieve it.

Now that we have our import table and our data, let's import some data! If we are fortunate enough to have `file` permission on our database server, we can use the `LOAD DATA INFILE` command to import our data file directly to the server. We'll use the `-q` option to suppress all of wget's output.

Type the following at the Unix prompt:

```
cold:~$ wget -q "http://courses.oreillyschool.com/dba2/downloads/payment.csv"
```

Now that we have some data to import, log into MySQL as root, and connect to the sakila database.

`LOAD DATA INFILE` takes a few parameters - it needs a **input file**, a **destination table**, a **field terminator**, and a **line terminator**. In this case, we are using a comma delimited file, with single newline characters delimiting lines.

Let's try an import!

Type the following at the MySQL prompt:

```
mysql> LOAD DATA INFILE '~/payment.csv'
INTO TABLE payment_import
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
(amount, payment_date);
```

The import should run fairly fast, even for a large number of records. MySQL will tell you exactly how many rows it imported.

OBSERVE:

```
mysql> LOAD DATA INFILE '~/payment.csv'
    -> INTO TABLE payment_import
    -> FIELDS TERMINATED BY ','
    -> LINES TERMINATED BY '\n'
    -> (amount, payment_date);
Query OK, 16049 rows affected, 16049 warnings (0.37 sec)
Records: 16049  Deleted: 0  Skipped: 0  Warnings: 16049

mysql>
```

That's all there is to it!

We can check out a few rows in the `payment_import` table. Let's write a query to return the first 10 rows.

Type the following at the MySQL prompt:

```
mysql> select * from payment_import LIMIT 0, 10;
```

As long as you typed everything correctly, you will see:

OBSERVE:

```
mysql> select * from payment_import LIMIT 0, 10;
+--------+---------------------+
| amount | payment_date        |
+--------+---------------------+
|   2.99 | 2005-05-25 11:30:37 |
|   0.99 | 2005-05-28 10:35:23 |
|   5.99 | 2005-06-15 00:54:12 |
|   0.99 | 2005-06-15 18:02:53 |
|   9.99 | 2005-06-15 21:08:46 |
|   4.99 | 2005-06-16 15:18:57 |
|   4.99 | 2005-06-18 08:41:48 |
|   0.99 | 2005-06-18 13:33:59 |
|   3.99 | 2005-06-21 06:24:45 |
|   5.99 | 2005-07-08 03:17:05 |
+--------+---------------------+
10 rows in set (0.00 sec)

mysql>
```

If your data is in a different format, you'll likely have to write a program or a script to convert it to something you can use with `LOAD DATA INFILE`. You can also use a text editor to create standard `INSERT` statements from your import data.

Another option would be to write a script or program to read your data and issue `INSERT` or even `SELECT` statement necessary to put your custom data in your database. This process is called **ETL**, for **E**xtraction, **T**ransformation, and **L**oading.

Now that we've examined the various ways to export and import data, we'll switch gears in the next lesson and learn how to ways to monitor database performance. See you then!

Performance
DBA 2: Administering MySQL Lesson 10

In the last few lessons we've learned how to perform common database administration tasks such as backing up tables and exporting data. In this lesson we're going to switch gears to discuss another important aspect of database administration: *performance*.

In an earlier lesson we discussed how to use indexes to keep databases performing well. In this lesson we'll learn ways you can monitor performance, and take a look at some other factors that contribute to overall database performance.

Most cars are very reliable these days, lasting well over a 100,000 miles. Little maintenance is required to keep things working. Problems do develop over time, and performance can degrade over a period of time. You might monitor your car's gas mileage in order to spot potential engine problems. You can also monitor your database server to help spot when things go wrong.

Keeping An Eye on Performance

The first question your users may ask you when something goes wrong might be **"is the database server running?"** This might be easy question to answer - just try to run a query! Sometimes this isn't the best solution. For a test server database names might change, tables may be altered, data may be deleted. A better solution would be to **`ping`** the database server using the `mysqladmin` utility.

It doesn't matter if your database server is running or not for the following exercise.

Type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p ping
```

If your database server is up and running, you'll see the following message:

OBSERVE:

```
cold:~$ mysqladmin -u root -p ping
Enter password:
mysqld is alive
cold:~$
```

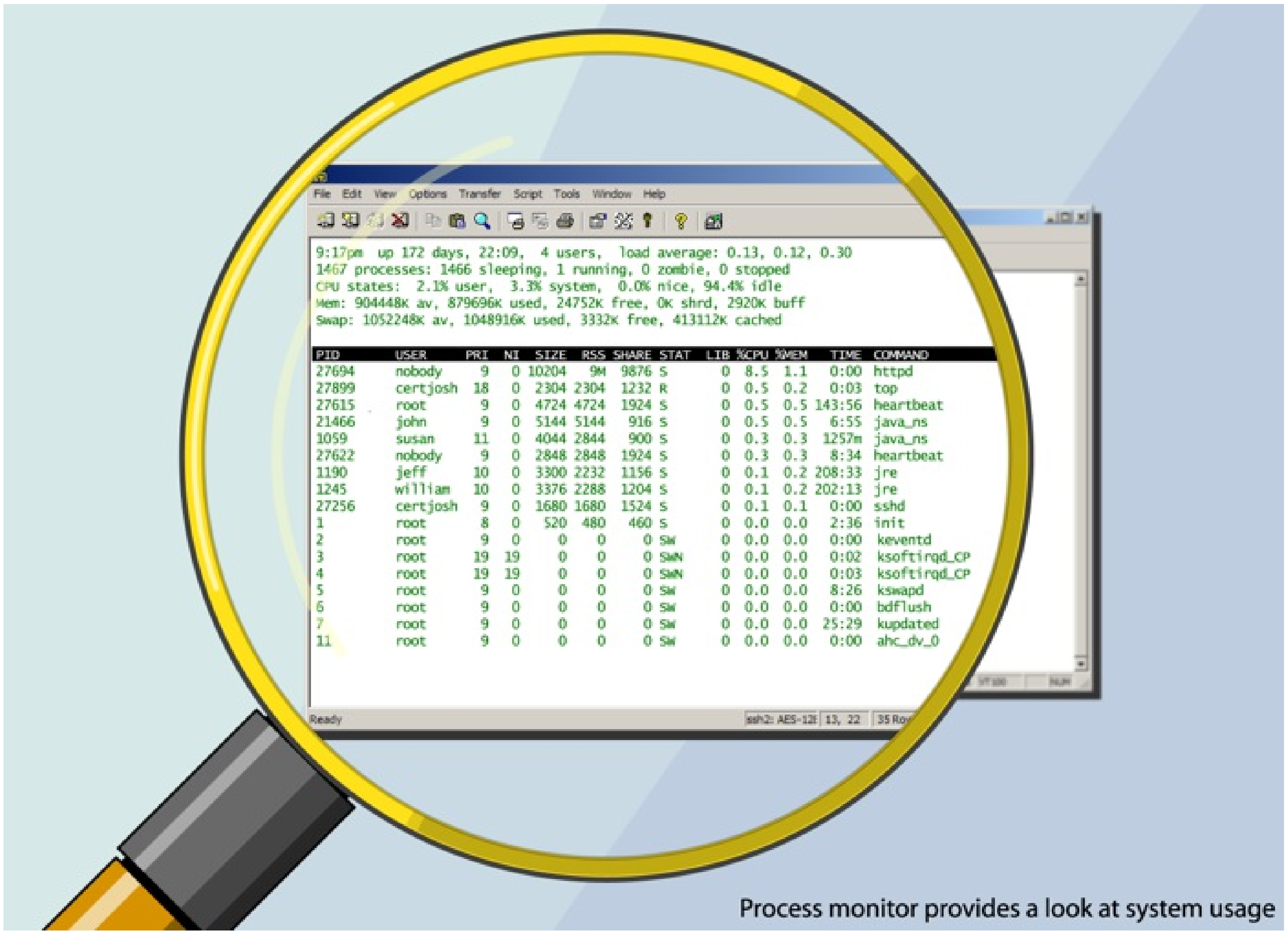
If you haven't started your server, you'll see a much longer message:

OBSERVE:

```
cold:~$ mysqladmin -u root -p ping
Enter password:
mysqladmin: connect to server at 'localhost' failed
error: 'Can't connect to local MySQL server through socket '/users/certjosh/mysql/data/mysql.sock' (2)
Check that mysqld is running and that the socket: '/users/certjosh/mysql/data/mysql.sock' exists!
cold:~$
```

In most companies the database is a critical piece of software that must be running at all times. If it is not running for some reason, the database or systems administrator needs to know. In the traditional UNIX fashion this program could be integrated into any number of systems from a simple status web page to a script that pages some database administrator in the wee hours of the morning.

Knowing the database server is running is great, but this does not tell us much about how the server is running. Before we can accurately monitor database performance, we need to monitor system performance. After all when a problem occurs it may not be MySQL - some other program may be hogging all of the computer's resources. A good program to monitor system performance on Linux/Unix is **`top`**.


File Edit View Options Transfer Script Tools Window Help
Ready
ssh2: AES-128 13, 22 35 Rows

```
9:17pm  up 172 days, 22:09,  4 users,  load average: 0.13, 0.12, 0.30
1467 processes: 1466 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  2.1% user,  3.3% system,  0.0% nice, 94.4% idle
Mem: 904448K av, 879696K used, 24752K free, 0K shrd, 2920K buff
Swap: 1052248K av, 1048916K used, 3332K free, 413112K cached

PID   USER     PRI NI  SIZE  RSS SHARE STAT LIB %CPU %MEM   TIME COMMAND
27694 nobody     9  0 10204   9M  9876 S      0  8.5  1.1   0:00 httpd
27899 certjosh  18  0  2304 2304  1232 R      0  0.5  0.2   0:03 top
27615 root       9  0  4724 4724  1924 S      0  0.5  0.5 143:56 heartbeat
21466 john       9  0  5144 5144   916 S      0  0.5  0.5   6:55 java_ns
1059  susan     11  0  4044 2844   900 S      0  0.3  0.3  1257m java_ns
27622 nobody     9  0  2848 2848  1924 S      0  0.3  0.3   8:34 heartbeat
1190  jeff      10  0  3300 2232  1156 S      0  0.1  0.2 208:33 jre
1245  william   10  0  3376 2288  1204 S      0  0.1  0.2 202:13 jre
27256 certjosh   9  0  1680 1680  1524 S      0  0.1  0.1   0:00 sshd
1     root       8  0   520  480   460 S      0  0.0  0.0   2:36 init
2     root       9  0     0    0     0 SW     0  0.0  0.0   0:00 keventd
3     root      19 19     0    0     0 SWN    0  0.0  0.0   0:02 ksoftirqd_CP
4     root      19 19     0    0     0 SWN    0  0.0  0.0   0:03 ksoftirqd_CP
5     root       9  0     0    0     0 SW     0  0.0  0.0   8:26 kswapd
6     root       9  0     0    0     0 SW     0  0.0  0.0   0:00 bdflush
7     root       9  0     0    0     0 SW     0  0.0  0.0  25:29 kupdated
11    root       9  0     0    0     0 SW     0  0.0  0.0   0:00 ahc_dv_0
```

Process monitor provides a look at system usage

Top updates your display every five seconds with refreshed system statistics. It reports statistics on memory use and process usage. It also displays which processes are using the most CPU on your system. Run the command below for a few minutes, then press `Q` to exit `top`.

Note `top` will continue to run until you press the **Q** key.

Type the following at the Unix prompt:

```
cold:~$ top
```

`top`'s display will look something similar to the output below. Looks like this machine is pretty idle - the `httpd` command is currently using the most CPU at the moment (however it is only using 8.5%).

OBSERVE:

```
  9:17pm  up 172 days, 22:09,  4 users,  load average: 0.13, 0.12, 0.30
1467 processes: 1466 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  2.1% user,  3.3% system,  0.0% nice, 94.4% idle
Mem:   904448K av,  879696K used,   24752K free,       0K shrd,    2920K buff
Swap: 1052248K av, 1048916K used,    3332K free                  413112K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
27694 nobody      9   0 10204   9M  9876 S       0  8.5  1.1   0:00 httpd
27899 certjosh   18   0  2304 2304  1232 R       0  0.5  0.2   0:03 top
27615 root        9   0  4724 4724  1924 S       0  0.5  0.5 143:56 heartbeat
21466 john        9   0  5144 5144   916 S       0  0.5  0.5   6:55 java_ns
 1059 susan      11   0  4044 2844   900 S       0  0.3  0.3  1257m java_ns
27622 nobody      9   0  2848 2848  1924 S       0  0.3  0.3   8:34 heartbeat
 1190 jeff       10   0  3300 2232  1156 S       0  0.1  0.2 208:33 jre
 1245 william    10   0  3376 2288  1204 S       0  0.1  0.2 202:13 jre
27256 certjosh    9   0  1680 1680  1524 S       0  0.1  0.1   0:00 sshd
    1 root        8   0   520  480   460 S       0  0.0  0.0   2:36 init
    2 root        9   0     0    0     0 SW      0  0.0  0.0   0:00 keventd
    3 root       19  19     0    0     0 SWN     0  0.0  0.0   0:02 ksoftirqd_CP
    4 root       19  19     0    0     0 SWN     0  0.0  0.0   0:03 ksoftirqd_CP
    5 root        9   0     0    0     0 SW      0  0.0  0.0   8:26 kswapd
    6 root        9    0      0     0      0 SW      0  0.0  0.0   0:00 bdflush
    7 root        9    0      0     0      0 SW      0  0.0  0.0  25:29 kupdated
   11 root        9    0      0     0      0 SW      0  0.0  0.0   0:00 ahc_dv_0
```

What do you do if it is MySQL that is taking up most of the CPU? Fortunately you can use `mysqladmin` to figure out who is doing what to your server. This is similar to the `top` or `ps` command on Unix/Linux systems.

Type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p processlist
```

Since your database server is only being used by you, there should only be one user connected to it right now. On a server with many users there would likely be many concurrent processes. As long as you typed everything correctly you'll see something similar to the following:

OBSERVE:

```
cold:~/mysql$ mysqladmin -u root -p processlist
Enter password:
+----+------+-----------+----+---------+------+-------+------------------+
| Id | User | Host      | db | Command | Time | State | Info             |
+----+------+-----------+----+---------+------+-------+------------------+
| 1  | root | localhost |    | Query   | 0    |       | show processlist |
+----+------+-----------+----+---------+------+-------+------------------+
cold:~/mysql$
```

This process list will always show at least one entry, since you must query the database server to get a list of the processes. The **Command** column shows the type of action the user is performing. In this example we are performing a **Query**. The **Time** column shows how long the command has taken, in seconds. In this example, the query *show processlist* has run for less than one second.

How would you know if some process was hogging server resources? The **Command** column will show **Query** and the **Time** column will show a large value. Take a look at this sample process list:

OBSERVE:

```
+----+------+-----------+----+---------+------+-------+------------------+
| Id | User | Host      | db | Command | Time | State | Info             |
+----+------+-----------+----+---------+------+-------+------------------+
| 1  | jan  | localhost |    | Query   | 3026 |       | select ...       |
| 2  | jon  | localhost |    | Query   | 0    |       | insert into ...  |
+----+------+-----------+----+---------+------+-------+------------------+
```

In this example **jan** is clearly running some complex query that has been running for over 50 minutes (3026 seconds)! It might be wise to kill Jan's process, then contact her to see what exactly she was doing.

Let's use the `kill` command on one of our processes. To do this you will need to open and log into a second Unix Terminal in CodeRunner. Each Unix Terminal will have its own number, such as "Terminal1" and "Terminal2".

Switch to the second Unix Terminal, then type the following at the Unix prompt:

```
cold:~$ mysql -u root -p
```

After you type your password, switch back to the first Unix Terminal.

Switch to the first Unix Terminal, then type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p processlist
```

If you typed your password correctly, you will see the following:

OBSERVE:

```
old:~$ mysqladmin -u root -p processlist
Enter password:
+-----+------+-----------+----+---------+------+-------+------------------+
| Id  | User | Host      | db | Command | Time | State | Info             |
+-----+------+-----------+----+---------+------+-------+------------------+
| 150 | root | localhost |    | Sleep   | 117  |       |                  |
| 151 | root | localhost |    | Query   | 0    |       | show processlist |
+-----+------+-----------+----+---------+------+-------+------------------+
cold:~$
```

The connection with Id=**151** is the command you just ran - **mysqladmin -u root -p processlist**. The connection with Id=**150** is your first connection to MySQL. Its state is **Sleep** because we are not running a query at this time.

Suppose we realized that the process with ID=**150** was taking too much resources. Lets kill that process. Make sure you are still connected to the first Unix Terminal. Be sure to replace **150** with *your* connection Id.

In the first Unix Terminal, type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p kill 150
```

MySQL won't give you much information after you run this command. You'll only see the following:

OBSERVE:

```
cold:~$ mysqladmin -u root -p kill 150
Enter password:
cold:~$
```

Let's check the process list again to see if it was killed.

In the first Unix Terminal, type the following at the Unix prompt:

```
cold:~$ mysqladmin -u root -p processlist
```

Looks like it is gone!

OBSERVE:

```
cold:~$ mysqladmin -u root -p processlist
Enter password:
+-----+------+-----------+----+---------+------+-------+------------------+
| Id  | User | Host      | db | Command | Time | State | Info             |
+-----+------+-----------+----+---------+------+-------+------------------+
| 153 | root | localhost |    | Query   | 0    |       | show processlist |
+-----+------+-----------+----+---------+------+-------+------------------+
cold:~$
```

If you switch back to the second Unix Terminal you won't see a difference until you try to run a query. Try a simple one now:

Switch to the second Unix Terminal, then type the following at the mysql prompt:

```
mysql> select now();
```

The command line MySQL program is smart enough to see the connection was killed, and to reconnect. It even tells you your new connection Id.

OBSERVE:

```
mysql> select now();
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    154
Current database: *** NONE ***

+---------------------+
| now()               |
+---------------------+
| 2008-09-24 10:52:04 |
+---------------------+
1 row in set (0.06 sec)

mysql>
```

Dealing with Performance Issues

When you identify a performance issue, what do you do? If the problem you've discovered is only temporary (such as increased traffic due to a new promotion) you may decide to do nothing since under ordinary circumstances everything works well.

The following lists some common problems you might encounter as a database administrator.

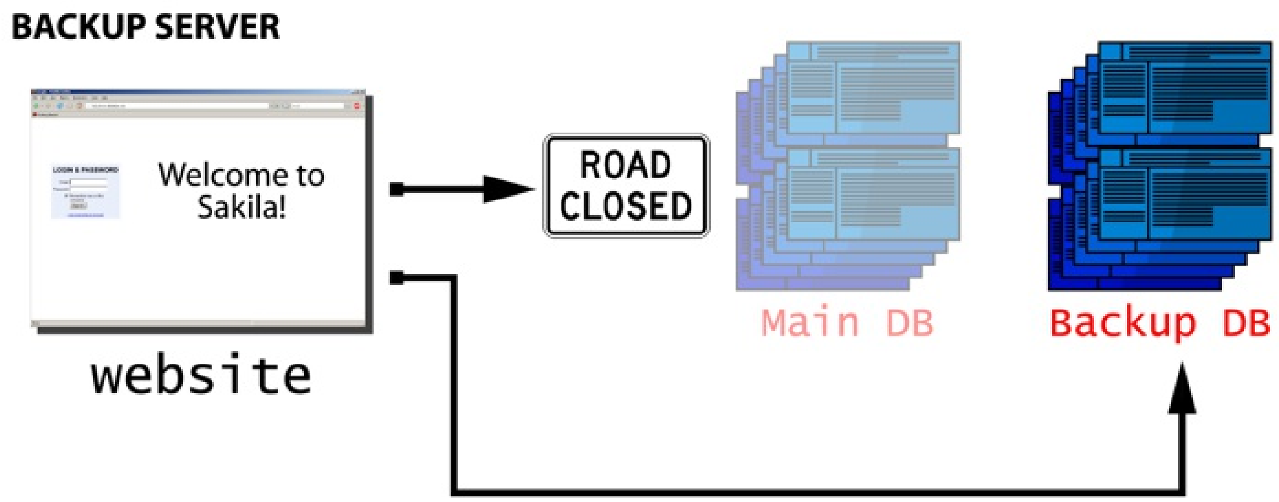
Symptom/Problem	Solution
Is any hardware malfunctioning?	Get it fixed or move to a new database server. A bad disk or flaky network card can cause serious slowdowns.
The computer doesn't have a lot of free memory.	Check to see if your database server has enough memory. If your server can accept more memory, add it. If not, consider moving to a larger server that can support more memory.
Other processes are hogging resources	Move the other processes to a different server, or get a dedicated database server.
The database server slows when users run certain reports or do ad-hoc queries.	Setup a separate database server (a *data warehouse*) just for reporting and ad-hoc queries. For additional information, check out the data warehousing course from O'Reilly.

Backup Servers and Clustering

At some point in time you will likely implement a backup server for your company's disaster recovery plan. Perhaps your database use is mostly read-only, so you want to spread queries across many physical machines (perhaps in different physical locations) as well. If you've upgraded your database server and you're still having performance problems, you may need to consider additional high availability options.

Backup Servers

When most people move beyond one database server, it is often to a backup server. A backup server is a relatively painless way to help your applications achieve 24x7 uptime by providing a spare machine that can take over in case the primary server fails. In some situations, a backup server could also service some read-only queries, thereby easing the workload on the primary database server.



A basic backup server can be set up using the `mysqldump` command to **pipe** - **|** - data into the `mysql` command. It would also be helpful to schedule this transfer, perhaps using a standard Unix scheduling facility such as `crontab`.

Note For more information on `crontab` check out the Linux/Unix System Administration series.

Let's try this with the database server you installed in the first lesson as the primary server, and the database server that comes with your learning account as the backup server. The database server is called **sql** - so we'll use the **-h sql** option when running mysql.

For this example, we'll limit our dump to the **inventory** table from the `sakila` database.

Type in the command below, and be sure to replace **certjosh** with your own username. and **PASSWORD** with your own password. Be sure not to put any spaces after the `-p` and before your actual password.

Type the following at the Unix prompt:

```
cold:~$ mysqldump -u root -pPASSWORD sakila inventory | mysql -u certjosh -pPASSWORD -h sql cer
```

Note If you mistype something, your command line may begin to act strangely. If you type and don't see anything on the screen, hit enter a few times, type `reset` and press enter again. Your terminal should be back to normal.

You won't see anything unless there was an error. To verify the tables were moved correctly, let's connect to the **sql** server.

Type the following at the Unix prompt:

```
cold:~$ mysql -u certjosh -p -h sql certjosh -e "select * from inventory limit 0, 10"
```

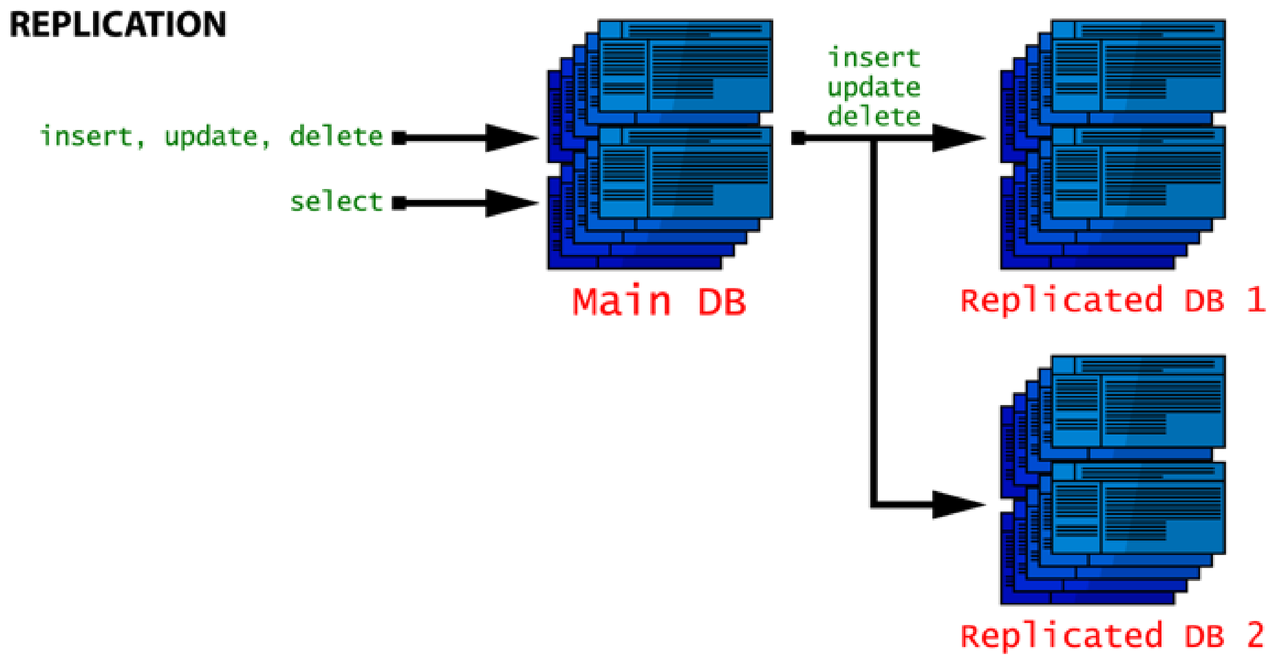
Make sure you replace **certjosh** with your own username, and be sure to type your learning account password. If you type everything correctly, you'll see the following:

OBSERVE:

```
cold:~$ mysql -u certjosh -p -h sql certjosh -e "select * from inventory limit 0, 10"
Enter password:
+--------------+---------+----------+---------------------+
| inventory_id | film_id | store_id | last_update         |
+--------------+---------+----------+---------------------+
|            1 |       1 |        1 | 2006-02-15 05:09:17 |
|            2 |       1 |        1 | 2006-02-15 05:09:17 |
|            3 |       1 |        1 | 2006-02-15 05:09:17 |
|            4 |       1 |        1 | 2006-02-15 05:09:17 |
|            5 |       1 |        2 | 2006-02-15 05:09:17 |
|            6 |       1 |        2 | 2006-02-15 05:09:17 |
|            7 |       1 |        2 | 2006-02-15 05:09:17 |
|            8 |       1 |        2 | 2006-02-15 05:09:17 |
|            9 |       2 |        2 | 2006-02-15 05:09:17 |
|           10 |       2 |        2 | 2006-02-15 05:09:17 |
+--------------+---------+----------+---------------------+
cold:~$
```

Sure enough - the data transferred to the other database server! You can schedule this command to run every night, or even every 12 hours, depending on how many times your database is updated throughout the day. If your database is fairly large, or if you need to keep your backup server updated more frequently than 12 hours or so, you'll have to begin thinking about **replication**.

In **replication**, your primary (or *master*) database server sends all changes to one or more *slave* servers. The changes are propagated instantaneously.



Here are some advantages to using replication:

- A backup server is ready to become a master, should the master server fail.
- Read only queries can be split across all database servers.
- Backups can be taken from a slave server instead of the master server, in order to place a minimal load on the master server.

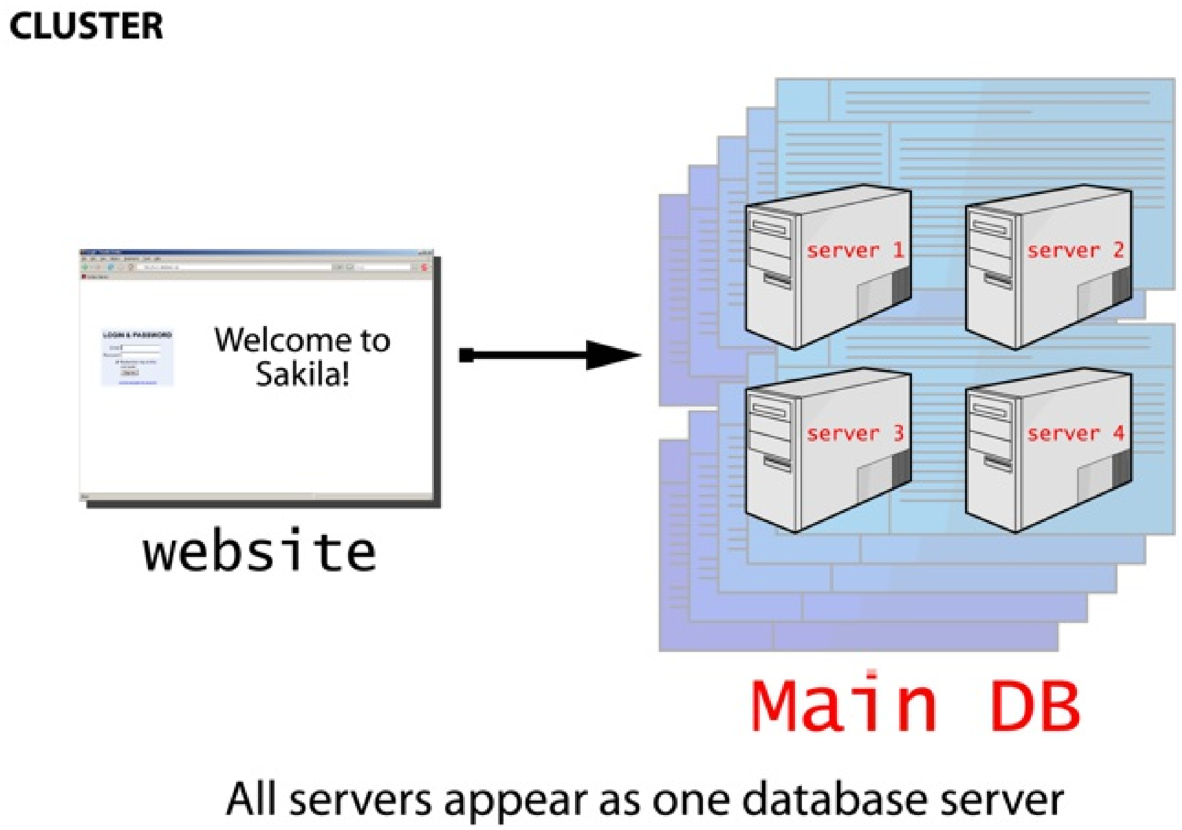
There are a couple of disadvantages, however:

- Updates can take place on slave servers, however they could be overwritten unless they are also made on the master server.
- If the master fails, an administrator must be notified, and must act to make the backup server the "new" master.

For most sites replication works very well, and is the perfect balance of security, cost and complexity.

Clustering

If your database sees a lot of update traffic, and your master server doesn't seem to handle the load anymore, you may need to consider **clustering** your database. In a database cluster, many physical computers (called *nodes*) act and appear as one single database. The end user queries the database and does not have to redirect update queries to a master server.



As of version 5.0, MySQL's cluster software has some serious drawbacks. For a complete list, check out the MySQL web site. Some of the main disadvantages include:

- There is no referential integrity when using a cluster. Foreign key constraints are ignored.
- There are limits to the names of database objects, and to the sizes of certain objects.
- You cannot use full text indexes.
- Changes to table structures are not automatically sent to all nodes on the cluster.

At this time, it is usually better to reconsider database *and* application architecture in order to split database load to multiple computers instead of using a MySQL cluster. Your company will be spending a lot of money on hardware to implement a cluster - it is worth spending money to make the application better instead of changing the application to work around limitations with the cluster software.

You are nearly there! In the next lesson we'll discuss the steps you need to take to troubleshoot your database. See you then!

Troubleshooting
DBA 2: Administering MySQL Lesson 11

In the last few lessons we've discussed some pretty big topics. In this lesson we'll examine wise troubleshooting steps you can take to diagnose problems you may encounter with MySQL. No administrator (or programmer) likes to debug software for very long!

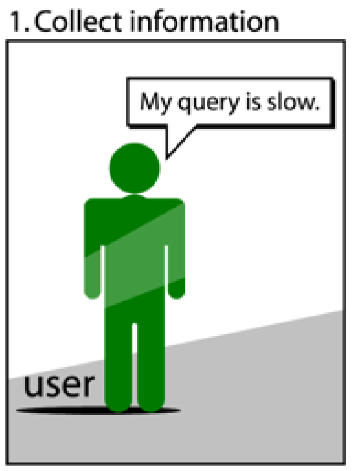
The Steps

Things can be stressful when your database is having a problem. During times of stress it is normal to want to get things back up and running as quickly as possible. This could be dangerous - if you restore from backup in an attempt to recover from an error, are you sure you'll solve the problem? What if you lose data?

Keeping cool and following a series of steps will help you solve any database problem you might encounter.

Step 1 - Collect Information

Tech support staff have countless stories of clueless users who report "computer problems." What is the problem - is the mouse not working? Is it the printer? Is it a problem with Excel, or Firefox? Sometimes narrowing down the problem is half the battle.



It is very important to clearly define the problem with your MySQL database in very specific terms. If someone is experiencing slow query times, find out *what* query is slow. Define *slow* - did it used to take one minute, but is now taking an hour?

Be sure to ask "the obvious" questions - what MySQL server (and what version of the server) are you using? How is it configured? What version of the client or program are you using? Many times there are no problems - just a confused user connected to the wrong database, or using the wrong version of the software.

Checking the version of MySQL is straightforward - just use the `--version` argument.

Type the following at the Unix prompt:

```
cold:~$ mysql --version
```

You should see something similar to the following:

OBSERVE:

```
cold:~$ mysql --version
mysql  Ver 14.12 Distrib 5.0.41, for pc-linux-gnu (i686) using  EditLine wrapper
cold:~$
```

Getting configuration settings is easy as well - you can use the `SHOW VARIABLES` command. You will have to connect to MySQL as root in order to run this command.

Type the following at the MySQL prompt:

```
mysql> SHOW VARIABLES;
```

You'll see a lot of rows - 225 for the current version of MySQL.

OBSERVE:

```
mysql> SHOW VARIABLES;
+----------------------------------+---------------------------------------------+
| Variable_name                    | Value                                       |
+----------------------------------+---------------------------------------------+
| auto_increment_increment         | 1                                           |
| auto_increment_offset            | 1                                           |
| automatic_sp_privileges          | ON                                          |
| back_log                         | 50                                          |

... many lines omitted ...

| tx_isolation                     | REPEATABLE-READ                             |
| updatable_views_with_limit       | YES                                         |
| version                          | 5.0.41-OREILLY                              |
| version_comment                  | Source distribution                         |
| version_compile_machine          | i686                                        |
| version_compile_os               | pc-linux-gnu                                |
| wait_timeout                     | 28800                                       |
+----------------------------------+---------------------------------------------+
225 rows in set (0.06 sec)

mysql>
```

Some interesting variables are:

Variable	Description
datadir	The physical location of MySQL's data files. You should check this value if you suspect your disk is running low on disk space, to make sure you are looking at the correct disk.
table_type	The default table type - more than likely MyISAM or InnoDB. For example, if you are trying to work with a full text index on a table and you are getting errors, you might have created the table as InnoDB instead of MyISAM
version	The specific version of MySQL you are using. This is useful when tracking down version specific issues or bugs.

One additional command will give you even more data: `SHOW STATUS`.

Type the following at the MySQL prompt:

```
mysql> SHOW STATUS;
```

It too returns 225 rows for this version of MySQL. Some of the rows are below:

OBSERVE:

```
mysql> SHOW STATUS;
+------------------------------------+-----------+
| Variable_name                      | Value     |
+------------------------------------+-----------+
| Aborted_clients                    | 4         |
| Aborted_connects                   | 43        |
| Binlog_cache_disk_use              | 0         |
| Binlog_cache_use                   | 0         |
| Bytes_received                     | 147       |
| Bytes_sent                         | 6488      |

... many lines omitted ...

| Threads_cached                     | 0         |
| Threads_connected                  | 1         |
| Threads_created                    | 237       |
| Threads_running                    | 1         |
| Uptime                             | 5887880   |
| Uptime_since_flush_status          | 5887880   |
+------------------------------------+-----------+
225 rows in set (0.12 sec)

mysql>
```

Some interesting variables are:

Variable	Description

Aborted_clients, Aborted_connects	The number of aborted clients and connections. If either number is increasing, there may be a network problem.
Created_tmp_disk_tables, Created_tmp_files, Created_tmp_tables	Number of temporary objects MySQL has created in order to answer queries. If the number is increasing, some queries may be performing poorly.
Slow_queries	The number of queries that MySQL considers to be "slow." A large number could indicate performance problems.
Uptime	How long the MySQL server has been running. If close to zero, the MySQL server was recently restarted, or crashed and was restarted.

If you are interested in seeing how `Aborted_clients` changes over time, you could write down the value in the morning, run `SHOW STATUS;` in the afternoon, then compare your numbers. You could also reset MySQL's counters to see how things change over time. To do this you can use the `FLUSH STATUS` command.

Type the following at the MySQL prompt:

```
mysql> FLUSH STATUS; SHOW STATUS;
```

Notice how the **Aborted_clients** and **Uptime_since_flush_status** statistics were reset to zero:

OBSERVE:

```
mysql> FLUSH STATUS; SHOW STATUS;
Query OK, 0 rows affected (0.00 sec)

+------------------------------------+-----------+
| Variable_name                      | Value     |
+------------------------------------+-----------+
| Aborted_clients                    | 0         |
| Aborted_connects                   | 0         |

... many lines omitted ...

| Uptime                             | 5888105   |
| Uptime_since_flush_status          | 0         |
+------------------------------------+-----------+
225 rows in set (0.01 sec)

mysql>
```

If you've collected all information, and you still have an issue...

Step 2 - Get More Information

Once you have a clear understanding of the problem, and are aware of details like software versions, you can start researching additional information. Some additional questions you might ask include:

- Did the data center have any power or connection issues?
- Did anyone add, remove, or change software recently?
- Is there any significance to today - did the web site get more traffic? Is end of month processing taking place?

Physically check the server. Many servers will beep or flash lights to alert users when there is a hardware problem. Are there any messages on the server's monitor? Are there any strange entries in the server log files?

On a Unix system you can use `dmesg` to see the console messages. You can also use the **tail** command to see the last entries of the log files. Let's try it!

Type the following at the Unix prompt:

```
cold:~$ tail mysql/data/cold1.useractive.com.err
```

Note The specific file used by your MySQL server may be different. It could be `cold.err` or `cold0.useractive.com.err` for example. It will always have the `.err` extension however.

The output from your error log will be different, but it may look something like the following:

OBSERVE:

```
070606 15:28:51  mysqld started
070606 15:28:51  InnoDB: Started; log sequence number 0 43655
070606 15:28:51 [Note] /users/certjosh/mysql/libexec/mysqld: ready for connections.
Version: '5.0.41-OREILLY'  socket: 'mysql.sock'  port: 0  Source distribution
```

Note The `tail` command has a `-n` option that specifies how many lines to display. By default it displays up to ten lines, but you could use `tail -n 100 mysql/data/cold1.useractive.com.err` to display the last 100 lines of the error log.

If you see something in your logs or console that looks abnormal and you don't know what it means, try Google. There is a pretty good chance that someone else has encountered your problem (and hopefully solved it)!

Let's see what happens when MySQL ends abnormally. We can simulate this type of failure by using the unix **kill** command to send a **ILL** signal to **MySQL's process**. Don't worry about the details on how we are shutting down MySQL. Make sure your MySQL server is running.

Type the following at the Unix prompt:

```
cold:~$ /bin/kill -s ILL `cat mysql/data/cold1.useractive.com.pid`
```

Note The specific file used by your MySQL server may be different. It could be `cold.pid` or `cold0.useractive.com.pid` for example. It will always have the `.pid` extension however.

If you typed everything correctly you will so no output.

Let's take a look at the error log to see what happened. We'll use the **-n** option to view the last 43 lines of the error log. Why 43? It happens to be the number of lines added by the kill command. In reality you wouldn't know how many lines to view in the error log, so you might view the last 25 lines, then the last 100 lines if 25 was not enough.

Type the following at the Unix prompt:

```
cold:~$tail -n 43 mysql/data/cold1.useractive.com.err
```

If you typed everything correctly, you would see the following:

OBSERVE:

```
cold:~$ tail -n 43 mysql/data/cold1.useractive.com.err
080925 15:14:04 - mysqld got signal 4;
This could be because you hit a bug. It is also possible that this binary
or one of the libraries it was linked against is corrupt, improperly built,
or misconfigured. This error can also be caused by malfunctioning hardware.
We will try our best to scrape up some info that will hopefully help diagnose
the problem, but since we have already crashed, something is definitely wrong
and this may fail.

key_buffer_size=16384
read_buffer_size=258048
max_used_connections=1
max_connections=100
threads_connected=1
It is possible that mysqld could use up to
```

```
key_buffer_size + (read_buffer_size + sort_buffer_size)*max_connections = 31615 K
bytes of memory
Hope that's ok; if not, decrease some variables in the equation.

thd=(nil)
Attempting backtrace. You can use the following information to find out
where mysqld died. If you see no messages after this, something went
terribly wrong...
Cannot determine thread, fp=0xbe7ff48c, backtrace may not be correct.
Stack range sanity check OK, backtrace follows:
0x8166ed3
0x40027f3c
0x40134a09
0x400282d4
0x81673c3
0x400250c8
0x401dc9ea
New value of fp=(nil) failed sanity check, terminating stack trace!
Please read http://dev.mysql.com/doc/mysql/en/using-stack-trace.html and follow instructions on
stack trace is much more helpful in diagnosing the problem, so please do
resolve it
The manual page at http://www.mysql.com/doc/en/Crashing.html contains
information that should help you find out what is causing the crash.

Number of processes running now: 0
080925 15:14:06  mysqld restarted
080925 15:14:06  InnoDB: Started; log sequence number 0 54140484
080925 15:14:06 [Note] /users/certjosh/mysql/libexec/mysqld: ready for connections.
Version: '5.0.41-OREILLY'  socket: 'mysql.sock'  port: 0  Source distribution
cold:~$
```

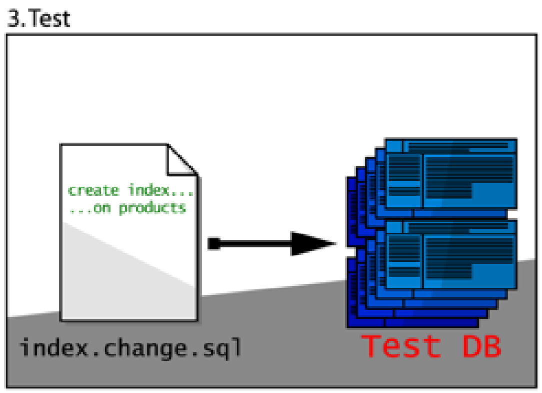
MySQL gave us a description of the error, in **red**. The very last message (in **bold**) lets us know that MySQL was able to restart itself.

This failure is just a simulation, so nothing is actually wrong with our MySQL server. If it wasn't a simulation, we could use this error log to search MySQL's web site for a solution.

Step 3 - Test

At this point, if you identified some physical problem such as a failed disk in a RAID (Redundant Array of Inexpensive Disks) array, your job will likely be done as soon as you replace the disk and the RAID finishes rebuilding. However, if your problem is software based, you have a bit more work to do.

Suppose you read on MySQL's forums that there is a known bug with the version of MySQL you are currently using. You decide your problem will be fixed by upgrading MySQL to the most current version. Do you go ahead and update your database server? **NO!!!**



Back in the first lesson we stressed the importance of having a development server. You really need to have at least one non-production machine you can use to test all changes without impacting your production server. What if the new version of MySQL has some bugs of its own, that are much worse than the bugs you are trying to fix?

How will you know if the new version of MySQL will fix your problem? Think about how you will measure success or failure. Will a query run faster after you apply the fix? Will the server stop crashing when someone tries to run a particularly difficult query? Be complete with your testing criteria.

Here are the steps you need to complete when testing a fix to your problem:

1. Make sure your test server matches your troubled server. Install old versions of MySQL

Testing your Fix

if necessary.

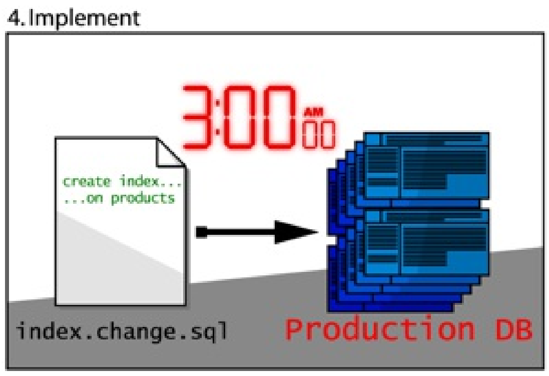
2. If possible, restore your last production backup to your test server.
3. Implement your fix.
4. Evaluate the results. If the test failed, go back to step 1.

If your test is successful, you are ready to move on to the last step.

Step 4 - Implementation

Before you jump right in and implement your fix, consider a few additional issues:

- Do you need to backup production before you continue?
- Will the server be inaccessible for a period of time?
- Can the fix wait until off-peak times?
- What should happen if the fix doesn't work?



There are no standard correct answers for these questions. For some problems, it may be okay to interrupt work to implement a fix as soon as possible. Other problems may have to wait until the night, a weekend, or even a holiday. As a database administrator you'll have to inform the business users of the situation and work with them to come to some conclusion.

You made it! You're on the path to becoming an expert MySQL administrator! Databases are complex pieces of software with many moving parts. Each situation is unique, and each company database is different. The key to your success is to practice, practice, practice! Good luck!

Final Project
DBA 2: Administering MySQL Lesson 12

Your Final Project

You made it!

For your final project, you'll need to create a new database of your choice, called **final_project**. It must have the following items:

1. At least four tables, with sample data of at least five rows
2. At least three users
    - Each user must have different permissions to tables
    - You must demonstrate row and column level security
3. Proper indexes on all tables
4. Tables must be optimized
5. The database must be backed up to a compressed file called **final_project.sql.bz2**
6. One table must be exported to HTML, and stored in a file called **final_project.html**

Sorry, but for your final project you are not allowed to use a sample database from the internet.

Before you get started, check with your mentor (by completing the quiz for this lesson) to make sure your project will meet the criteria.

Good luck!