

DBA 1: Introduction to Database Administration ([PDF](#))

Lesson 1: [Introduction](#)

[How to Learn using O'Reilly School of Technology Courses](#)

[Understanding the Learning Sandbox Environment](#)

Lesson 2: [Storing Data](#)

[Storing Data](#)

[Data and Data Types](#)

[What is NULL and NOT NULL anyway?](#)

[Our First Table](#)

Lesson 3: [Relationships Between Tables](#)

[Other Tables](#)

Lesson 4: [INSERT, SELECT, UPDATE, DELETE](#)

[DESCRIBING your tables](#)

[Insert](#)

[SELECT](#)

[Update](#)

[Delete](#)

Lesson 5: [Transactions](#)

[Making sure your commands behave](#)

[ACID](#)

[Transaction Isolation levels](#)

[Using Transactions](#)

[Committing and rolling back](#)

Lesson 6: [Joins](#)

[Combining Tables with a Join](#)

[Other Joins](#)

[Left Joins](#)

[Full Outer Joins / Union](#)

Lesson 7: [Aggregates, Functions and Conditionals](#)

[Aggregating Data](#)

[Functions](#)

[Conditionals](#)

Lesson 8: [Sub Queries and Views](#)

[Querying Queries](#)

[Views](#)

[Creating a view](#)

[Restrictions on views](#)

[Dropping a view](#)

Lesson 9: Stored Procedures

[Using Stored Procedures](#)

[Motivations](#)

[Creating Stored Procedures](#)

[Setup](#)

[Duplicating our View](#)

[Parameters](#)

[Variables](#)

Lesson 10: PIVOT and UNPIVOT

[PIVOTing data](#)

[UNPIVOTing Data](#)

Lesson 11: Full Text

[Creating Full-Text Indexes](#)

[Querying full-text Indexes](#)

Lesson 12: INFORMATION_SCHEMA

[INFORMATION_SCHEMA](#)

[Tables](#)

[Columns](#)

[IEWS](#)

[ROUTINES](#)

Lesson 13: Final Project

[Final Project](#)

Introduction

DBA 1: Introduction to Database Administration Lesson 1

How to Learn using O'Reilly School of Technology Courses

Welcome to the O'Reilly DBA Series!

In this course you will learn advanced techniques for interacting with databases. You've probably used a database before. You may have even created an application using a database like MySQL, PostgreSQL, or even Oracle or SQL Server. Whatever your experience level, it's time to take your knowledge to the next level!

In this course we'll cover the basics of queries, then move on to more advanced topics such as stored procedures, transactions and pivots.

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you **learn to learn** the technology.

Here are some tips for using O'Reilly School courses effectively:

- **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.

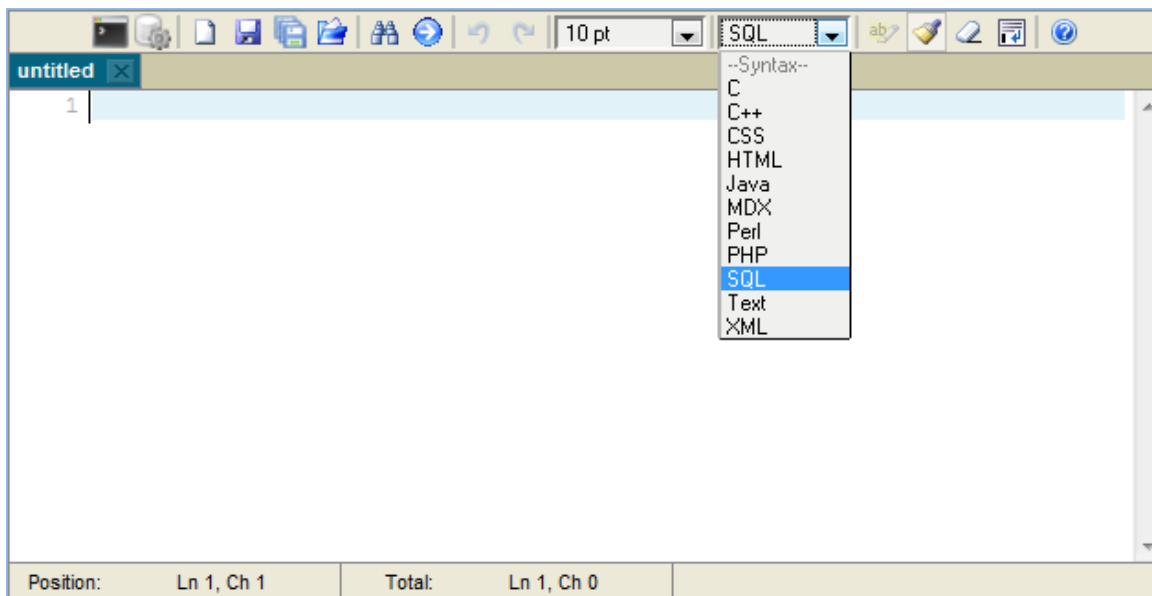
As you can see, all of the communication tools and learning content go in the upper part of the screen. Frequently we will ask you to type code into the CodeRunner Editor below and experiment by making changes. When we want you to type code on your own, you will see a white box like the one below, with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

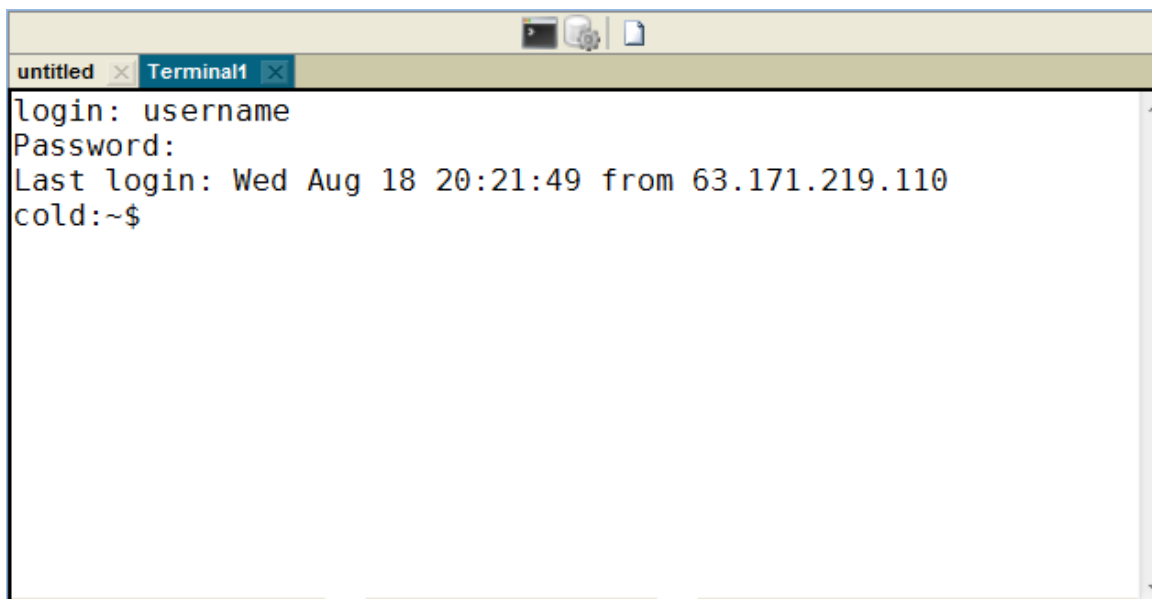
Similarly, when we want you to simply observe some code or result, we will put it in a gray box like this:

Observe:
Gray boxes will be used for observing code.
You are not expected to type anything from these "example" boxes.

In this course you will be creating SQL files. SQL files are made up of SQL statements and have an .sql extension. Click on the **New Terminal** button  to connect to Unix or click on the **Connect to MySQL** button  to connect directly to the MySQL server. In this course we'll primarily be using the Unix Terminal, and not the MySQL server directly. By doing so we have some additional control over how we connect to MySQL. Many of the projects will be turned in as SQL files.



Try clicking the **New Terminal** button  to connect to Unix 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.



Once you see the UNIX prompt, it's time to connect to MySQL! Be sure to replace **username** and **username** with your own username.

Type the following at the UNIX prompt:

```
cold:~$ mysql -h sql -p -u username username
```

If you typed everything correctly your screen will look similar to this:

OBSERVE:

```
login: username
Password:
Last login: Wed Aug 18 20:21:49 from 63.171.219.110
cold:~$ mysql -h sql -p -u username username
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 135535
Server version: 5.0.62-log Source distribution

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

```
mysql>
```

In the previous command, we are running the program **mysql**. To specify which database server we want to connect to, we type **-h sql**. And since we want to be prompted to provide a password, we use the **-p** option. To specify our username, we use the **-u username** option. Finally, the database we want to use has the same name as our username, so we provide it again: **username**

Before we are ready to use MySQL, let's type in a command that tells MySQL to act in a "traditional" way. The command will cause MySQL to act more like PostgreSQL, Microsoft SQL Server and Oracle.

Type the following at the MySQL prompt:

```
mysql> set sql_mode='traditional';
```

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

OBSERVE:


```
mysql> set sql_mode='traditional';
Query OK, 0 rows affected (0.00 sec)

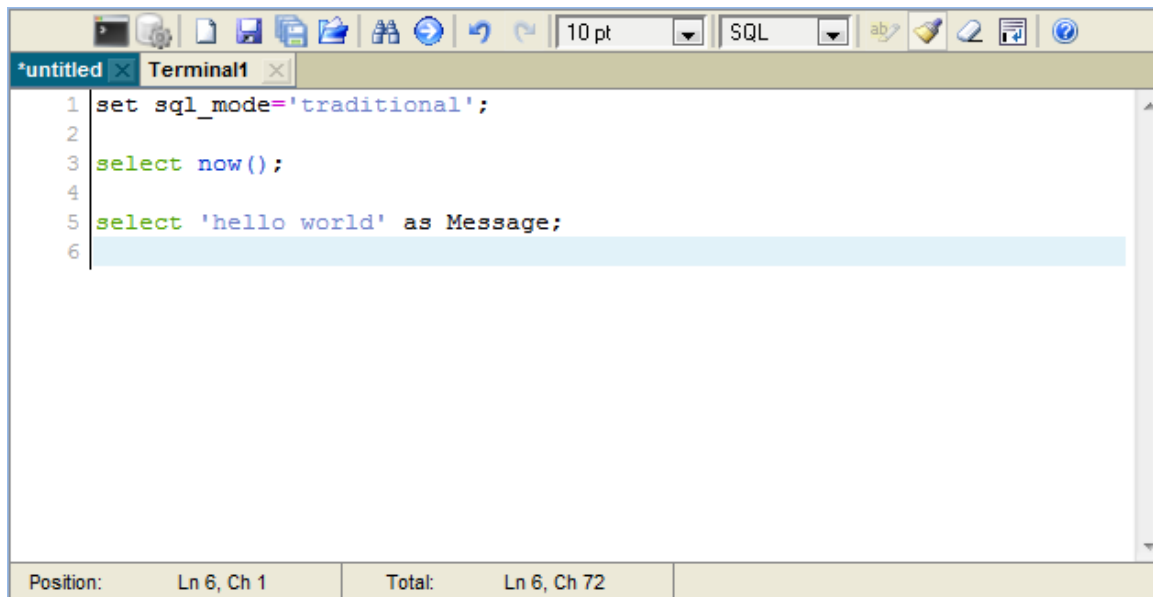
mysql>
```

When you are ready to leave mysql and return to the Unix prompt, type exit.

OBSERVE:

```
mysql> exit
Bye
cold:~$
```

For most projects you will select SQL from the Syntax menu in CodeRunner. If you are in the Unix Terminal, click on the "untitled" tab next to the Terminal tab or click on the **New File** button  in the menu bar to create a new "untitled" tab.




Try it out - make sure you are in SQL syntax.

Type the following into CodeRunner:

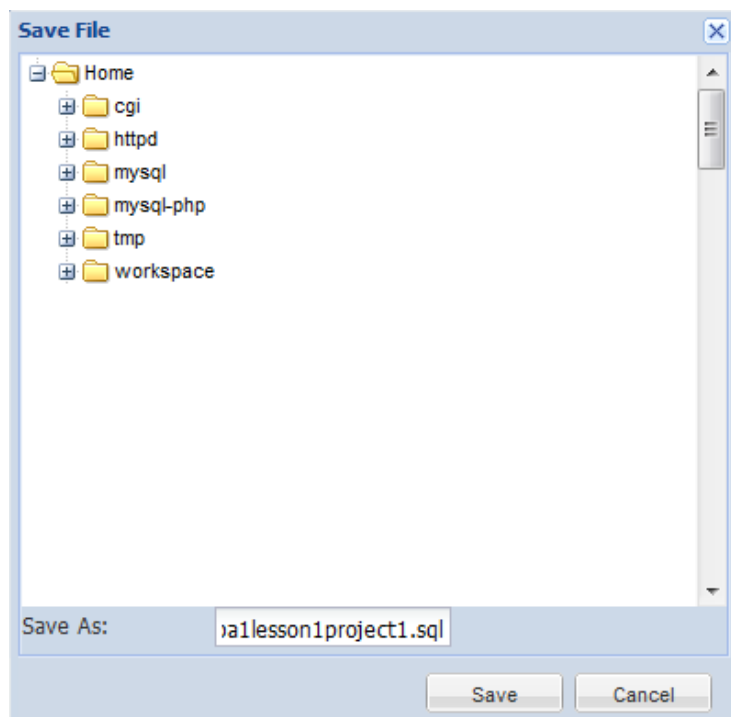
```
set sql_mode='traditional';

select now();

select 'hello world' as Message;
```

Save your file by clicking on the **Save** button . Be sure to name it appropriately and use the **.sql** extension. For this example, call it **dba1lesson1project1.sql**. When naming files and creating tables, please be sure to name them exactly as requested. In a Unix system, case is significant, and dba1lesson1project1.sql and DBA1Lesson1Project1.sql would be entirely different files and Artists and

artists would be different tables. While file and table names are case sensitive, MySQL keywords, such as **SET** and **SELECT**, are not, and can be entered in either upper or lower case.



To open a file you have previously saved, click on the **Load** button  and select the file from the **Load File** window. Note that when you save a file, the word "untitled" is replaced by the file name. When you open a file, the file opens with a new tab.

Before you hand in your SQL file be sure to test it in the Unix Terminal using the following command:

Type the following at the UNIX prompt:

```
cold:~$ mysql -t -h sql -p -u username username < dballesson1project1.sql
```

The **-t** option tells MySQL to output your queries in a nice table format. Be sure to replace **username** with your specific username, and **dballesson1project1.sql** with your project filename.

When you run the command, you'll see something like this:

OBSERVE:

```
cold:~$ mysql -t -h sql -p -u username username < dballesson1project1.sql
Enter password:
+-----+
| now() |
+-----+
| 2010-08-18 20:49:58 |
+-----+
+-----+
| Message |
+-----+
| hello world |
+-----+
cold:~$
```


When you hand in your project, you'll submit the file called **dballesson1project1.sql**.

Now you're ready to move on to the next lesson, where you'll dive right in and create some tables!

Storing Data

DBA 1: Introduction to Database Administration Lesson 2

Storing Data

Before we get started, make sure you are in the Unix Terminal and then connect to MySQL. Click on the **New Terminal** button  and login. Be sure to click in the window before you start typing. When you have logged in, connect to the MySQL server. Remember to replace **username username** with your actual username, and be sure to type your username twice!

Type the following at the UNIX prompt:

```
cold:~$ mysql -h sql -p -u username username
```

Your screen should look like this:

OBSERVE:

```
cold:~$ mysql -h sql -p -u username username
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 778597
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql>
```

Don't forget - you also need to set the 'TRADITIONAL' mode.

Type the following at the MySQL prompt:

```
mysql> set sql_mode='traditional';
```

Now you're ready to go!

Databases excel at four operations: **C**reating data, **R**eading data, **U**dating data, and **D**eleting data, or *CRUD*. The most popular databases use Structured Query Language (SQL) to perform those operations.

You cannot build a skyscraper without blueprints, and you cannot store data in a database without defining what type of information you wish to store, and how you want to store it. The subset of SQL to create your blueprints is called Data Definition Language (DDL).

Data and Data Types

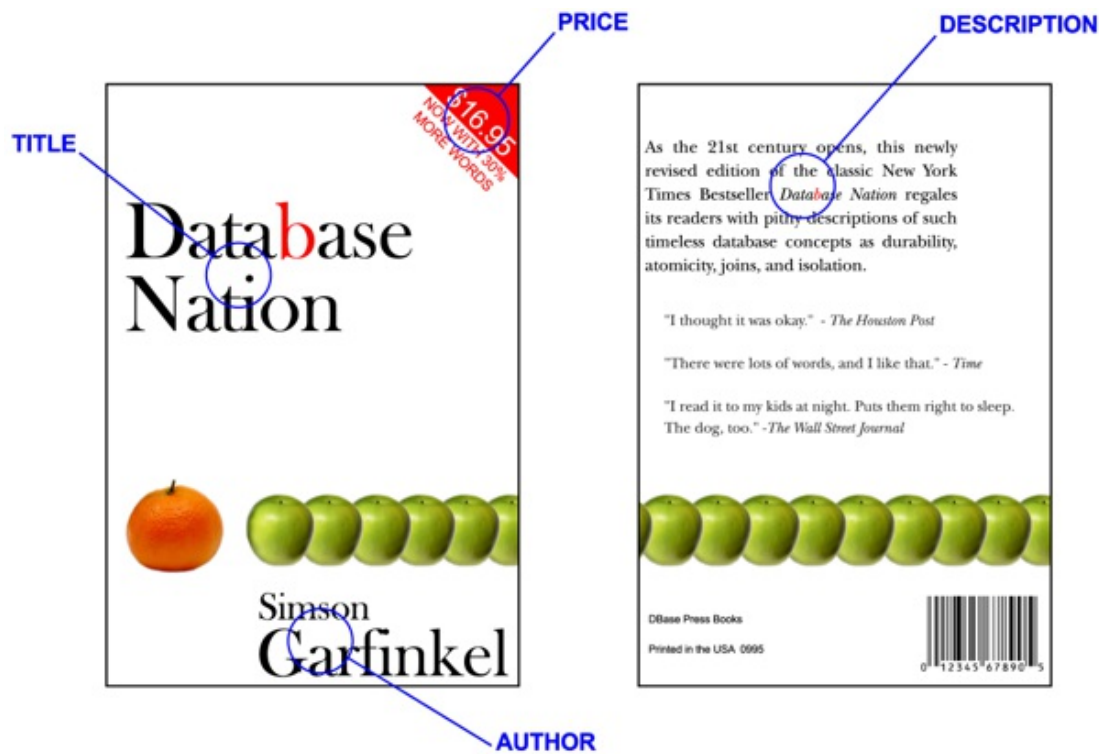
Before you can build the walls of a skyscraper, you must decide which material you will use to build your walls. Database administrators must also choose the material with which to construct their database. Let's work with an example.

Suppose you've just been hired by a used book store. Your job is to build a new database system for tracking books, among other things.

Since we are interested in keeping track of books, we need to determine the data that comprises a book. Most books have a title, an author, the date the book was published, a count of the number of pages, and a price. Some books contain a brief description of its content, others do not. For example, *Database Nation* was written by Simson Garfinkel in December 2000 and currently costs \$16.95. The description of the book reads:

"As the 21st century begins, advances in technology endanger our privacy in ways never before imagined. This newly revised update of the popular hardcover edition, 'Database Nation: The Death of Privacy in the 21st Century,' is the compelling account of how invasive technologies will affect our lives in the coming years. It's a timely, far-reaching, entertaining, and thought-provoking look at the serious threats to privacy facing us today."

Here is the cover of *Database Nation*, with some of its data items highlighted:



Database tables store information in rows and columns. The columns define the definition (or template) that each row must follow. For a table that stores book information, we might have columns for **Title**, **Author**, **Date Published**, **Number of Pages**, **Price**, and **Description**.

Each column of information is of a different type. For example, the title and author are comprised of words. The date published is exactly that: a date. The number of pages is a whole number. The cost is a decimal, with two decimal places. The description, if one exists, is also comprised of words.

As a database administrator (DBA) it's your job to pick the smallest and most appropriate data type for the information we want to store. You could just store everything as a bunch of words. For example, you could store the Date Published as **05/10/2007**. If you saw this data, would you be certain it was May 10th, 2007, or could it be October 5th, 2007? If you store it as a date there would be no question, since the database could return the date to you in any form you like, even "May 10th, 2007." It could also do some calculations for you, like tell you that "one month prior to the publish date" was "April 10th, 2007."

Data types can be split into three groups - text data types, numeric data types, and date/time data types. Below is a table of some of the data types that MySQL supports.

Group	Data Type	Description	Sample
Text	CHAR	A fixed length of zero to 255 characters. Useful for data like state codes, which are always two characters long.	IL
Text	VARCHAR	A variable length of zero to 65535 characters. Useful for data whose length varies, but is always within a certain size range. Common examples are first or last names.	ILLINOIS
Text	TEXT	A variable length of up to 65535 characters.	Illinois is a great state.
Text	MEDIUMTEXT	A variable length of up to 16777215 characters.	This history of Illinois...
Text	LONGTEXT	A variable length of up to 4294967295 characters.	The longer history of Illinois...
Numeric	INT	Integer (whole) numbers.	5
Numeric	FLOAT	Floating decimal point numbers.	5.15
Numeric	DOUBLE	Decimal number, with a fixed number of digits before and after the decimal point.	7.25
Numeric	DECIMAL	Decimal number, with a fixed number of digits, just like DOUBLE. More accurate than FLOAT or DOUBLE, use for monetary values like prices.	2.25
Date	DATE	Year, month and day.	2009-12-31
Date	DATETIME	Year, month, day and time.	2009-12-31 14:25:00

Date	TIME	Time only.	14:25:00
Date	YEAR	Year only. Year(4) has a range of 1901-2155, year(2) has a range of 1970-2069.	2012

What is NULL and NOT NULL anyway?

In a database you might have a column that is optional, such as the second address line for your customers. Elsewhere you might want to store some bit of information, like the list price (a decimal), which is also optional. You could set the list price to be \$-9999.99 and define that value as the "no list price" value, but there is a better way.

NULL is nothing. It is no value.

WARNING

Be careful with *NULL* - *NULL* is not equal to "". It is *not* an empty string. It *cannot* be compared to something else. However, even though is nothing, you can tell if something `IS NULL` or `IS NOT NULL`.

Our First Table

In order quickly interact with data in our tables, we need to define the **primary key** columns on our table. A **primary key** is the minimum set of information needed to uniquely identify a row in a database table. At first glance, the primary key for our book table could be the title.

Database Nation is one title, however it comes in a hard copy and a paperback edition. If our book store stocks both editions, the title "Database Nation" alone won't be enough for us to distinguish between the hard copy and paperback editions. Thus we won't be able to use the title as our primary key.

You might say that the price would be more expensive on the hard copy, but what if it is on sale, and is now less expensive than the paperback edition?

You might decide to shoehorn some extra information to the book's title - perhaps creating two books:

- Database Nation - Hard Copy
- Database Nation - Paperback

This type of compromise isn't a good idea. The title is **Database Nation** - not **Database Nation - Hard Copy**.

Our manager might step in at this point, and tell us that everything in the store has a unique **Product Code**. This product code should be our primary key, since it can be used to uniquely identify a row in our table.

We will use the **ProductCode** as our primary key, add a column called **Category**, and forget about the Author, the Date Published, and Number of Pages columns for now.

Before we get started, make sure you put CodeRunner in **MySQL** mode. Click in the window, and type in your password.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE Products
(
  ProductCode char(20) NOT NULL,
  Title varchar(50) NOT NULL,
  Category varchar(30) NOT NULL,
  Description text,
  Price DECIMAL(9,2) NOT NULL,
  PRIMARY KEY(ProductCode)
) ENGINE=INNODB;
```

If you typed everything correctly you will see the following:

OBSERVE:

```
mysql> CREATE TABLE Products
-> (
->   ProductCode char(20) NOT NULL,
->   Title varchar(50) NOT NULL,
->   Category varchar(30) NOT NULL,
->   Description text,
->   Price DECIMAL(9,2) NOT NULL,
->   PRIMARY KEY(ProductCode)
-> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

The **Products** table is fairly small - it contains a **ProductCode** used to uniquely identify a row, the Title of the item, a Category, an

optional Description, and the Price. The **NOT NULL** keywords indicate which columns are required or, which columns cannot be null. We didn't specify "NOT NULL" for **Description**, so we don't require a value in that column. Finally, we specify our **PRIMARY KEY** as **ProductCode**.

Can a primary key be null? Create a little test table to find out.

Note

In this course we will use the **INNODB** table type (`ENGINE=INNODB`) for most of the tables we create. This table type allows us to use more standard SQL than the normal MySQL tables.

Believe it or not - you just created a new table! In the next lesson you will create additional tables, and see how all of the tables relate to each other. See you then!

Relationships Between Tables

DBA 1: Introduction to Database Administration Lesson 3

Other Tables

In the last lesson we learned how to create tables. One table is okay, but we'll need a few additional tables to keep track of customers, inventory, and orders.

In the last lesson we decided to store the ProductCode, Title, Category, Description and Price for all of our books. Being a book store, it would be nice to track our customer and orders so we know which books are selling well. We could add a new column called "Customer" and a column called "DatePurchased" to our book table, but then we would have to copy the ProductCode, Title, Category, Description and Price each time a book was sold!

A better solution is to use a separate table for each group of information - a table for books, a table for customers, and a table for orders. This way we don't have a lot of duplicate information in our tables, wasting space and potentially causing confusion.

MySQL, PostgreSQL, Oracle, and SQL Server are all *relational* databases. They efficiently store information in multiple tables that are linked together by relationships.

The first table we'll create will keep track of customers. The data used to describe a customer should include their first and last name, address, and email address. But suppose you had two customers with the name "John Smith," or a customer who shares his email address with his wife? How would you keep these individual entries separate?

We need a primary key for our customer table, but we really don't have any unique information. Fortunately 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*. We'll use a surrogate key for our customer table, and call it **CustomerID**.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE Customers
(
  CustomerID int AUTO_INCREMENT NOT NULL,
  FirstName varchar(50) NOT NULL,
  LastName varchar(50) NOT NULL,
  AddressLine1 varchar(50) NOT NULL,
  AddressLine2 varchar(50),
  City varchar(50) NOT NULL,
  State char(2),
  PostalCode varchar(10),
  EmailAddress varchar(100),
  DateAdded DATETIME NOT NULL,
  PRIMARY KEY (CustomerID)
) ENGINE=INNODB;
```

Make sure everything is entered correctly, then execute the SQL. You should see this:

OBSERVE:

```
mysql> CREATE TABLE Customers
-> (
->   CustomerID int AUTO_INCREMENT NOT NULL,
->   FirstName varchar(50) NOT NULL,
->   LastName varchar(50) NOT NULL,
->   AddressLine1 varchar(50) NOT NULL,
->   AddressLine2 varchar(50),
->   City varchar(50) NOT NULL,
->   State char(2),
->   PostalCode varchar(10),
->   EmailAddress varchar(100),
->   DateAdded DATETIME NOT NULL,
->   PRIMARY KEY (CustomerID)
-> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.02 sec)

mysql>
```

Let's take a closer look at the command you just executed.

The first line tells the database server to create a new table called **Customers**.

This table contains our surrogate key column called **CustomerID**. Notice the use of the MySQL-specific keyword **AUTO_INCREMENT**.

Note **Auto Increment** columns sometimes have gaps. For instance, a table may have a 1 in the first row, then 5 in the next row, depending on whether rows are added or removed from the table.

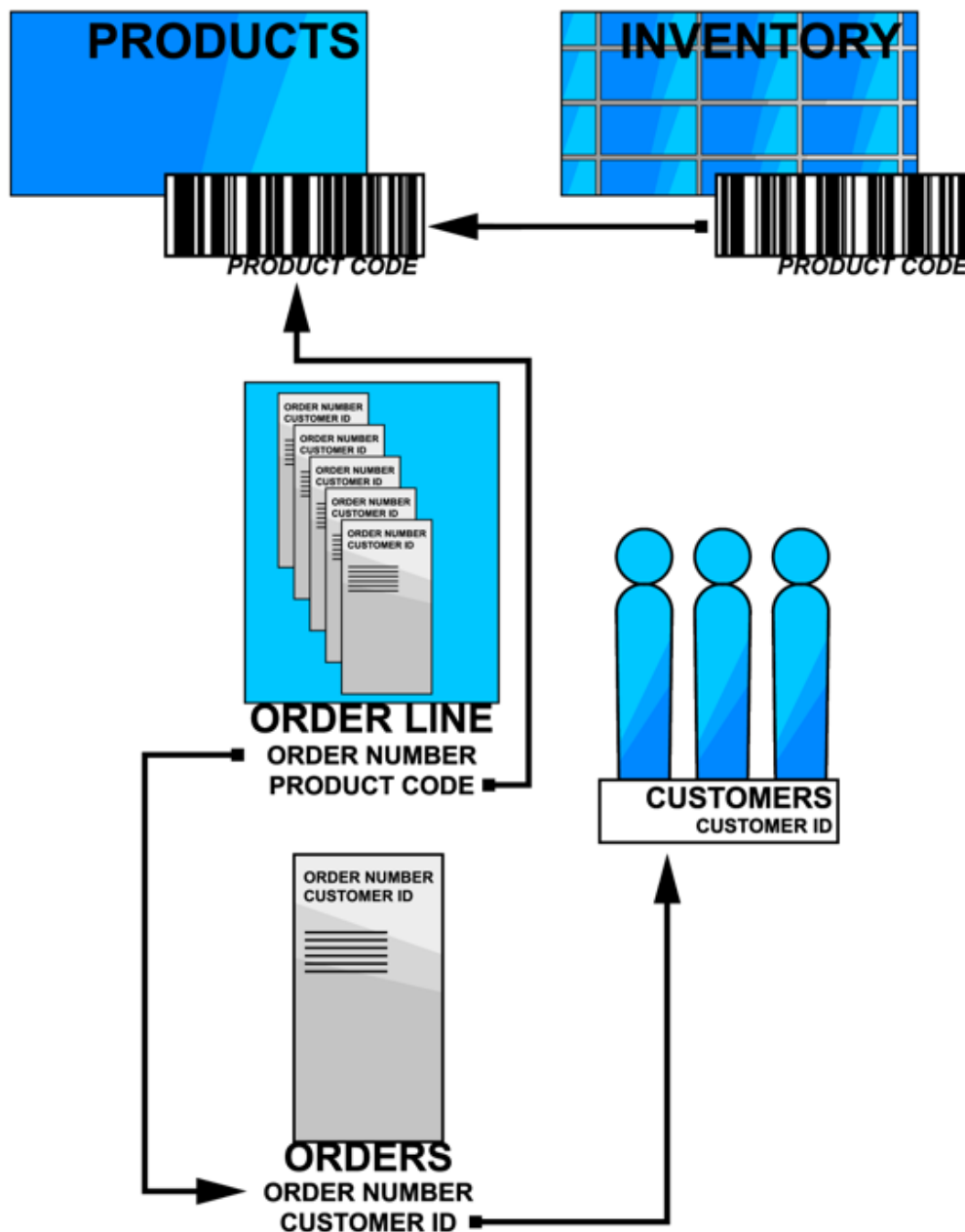
Next, the name column is defined. **FirstName** will be a **varchar** with a maximum length of **50** characters. This should be a required value for all customers, so it is defined using the **NOT NULL** keywords. The columns for LastName and AddressLine1 are defined similarly.

AddressLine2 is also a **varchar** of maximum length **50** characters. But since some of your customers don't have a second address line, you don't need to include the **NOT NULL** keywords.

DateAdded will store the date and time the customer is added to the table.

Finally, you let the database engine know that the **PRIMARY KEY** is the surrogate key, **CustomerID**.

Now that you have the Customers table in place, let's create some additional tables to help organize the bookstore. We'll need to create tables to store **Inventory**, keep track of **Customers**, track **Orders**, and tell us what products make up an order (**OrderLine**). We can represent these tables graphically this way:



In the diagram, arrows point to the **source** of the information. In other words, the **CustomerID** in Orders has a corresponding **CustomerID** in Customers.

Here are some of the other tables we'll create:

Table Name	Columns
Inventory	ProductCode, QuantityInStock
Orders	OrderNumber, CustomerID, InvoiceNumber, DatePlaced, OrderStatus
Order Line	OrderNumber, ProductCode, Quantity, ExtendedAmount

The **Inventory** table will keep track of the number of books in stock. Note that we uniquely identify the quantity in stock by **ProductCode**, which relates back to the Products table. We need to include the **ProductCode** in both tables in order to make this link. A ProductCode of 'ABC123' in Products will correspond exactly to a ProductCode of 'ABC123' in Inventory.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE Inventory
(
  ProductCode char(10) NOT NULL,
  QuantityInStock int NOT NULL,
  PRIMARY KEY(ProductCode)
) ENGINE=INNODB;
```

Again, if you typed everything correctly, you will see the message Query OK, 0 rows affected (0.00 sec).

Next we create a table to hold **Orders**. This table will contain an auto-increment column so that each order can be uniquely identified.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE Orders
(
  OrderNumber int AUTO_INCREMENT NOT NULL,
  CustomerID int NOT NULL,
  InvoiceNumber varchar(15),
  DatePlaced DATETIME,
  OrderStatus char(10),
  PRIMARY KEY(OrderNumber)
) ENGINE=INNODB;
```

Finally, the **OrderLine** table will be used to record the items purchased for each unique order.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE OrderLine
(
  OrderNumber int NOT NULL,
  ProductCode char(10) NOT NULL,
  Quantity int NOT NULL,
  ExtendedAmount DECIMAL(9, 2) NOT NULL,
  PRIMARY KEY(OrderNumber, ProductCode)
) ENGINE=INNODB;
```

This table has something new - the **PRIMARY KEY** is defined on two columns. A **PRIMARY KEY** uniquely identifies a row in a table. Many times the primary key is just one column, such as a userid or UPC code. It can be multiple columns, however.

Note A key (primary or other) that spans multiple columns is also known as a *composite key*.

In our table the **OrderNumber** and **ProductCode** together will uniquely identify a row.

Believe it or not - you just made four new tables! In the next lesson you will learn how to store data in these tables. See you then!

INSERT, SELECT, UPDATE, DELETE

DBA 1: Introduction to Database Administration Lesson 4

DESCRIBING your tables

In the last lesson you created tables that will store data for your bookstore. Now it's time to actually store some data! Before we get started, let's examine the tables created in the last lesson. MySQL has a `DESCRIBE` keyword that can be used to do just that.

Note Before you get started, make sure you are connected to MySQL, and have set the 'TRADITIONAL' mode.

Type the following at the MySQL prompt:

```
mysql> DESCRIBE Customers;
```

MySQL will return a description of the table you created in the last lesson.

Here is your Customers table:

Field	Type	Null	Key	Default	Extra
CustomerID	int(11)	NO	PRI	NULL	auto_increment
FirstName	varchar(50)	NO			
LastName	varchar(50)	NO			
AddressLine1	varchar(50)	NO			
AddressLine2	varchar(50)	YES		NULL	
City	varchar(50)	NO			
State	char(2)	YES		NULL	
PostalCode	varchar(10)	YES		NULL	
EmailAddress	varchar(100)	YES		NULL	
DateAdded	datetime	NO			

10 rows in set (0.02 sec)

`DESCRIBE` returns a lot of useful information - the column names, the data types (and sizes), whether the column can be NULL, any default values, and whether the column is `auto_increment`.

Insert

The `INSERT` keyword is used to add data to a table.

Type the following at the MySQL prompt:

```
mysql> INSERT INTO Customers
(FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
VALUES ('John', 'Smith', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'john@hotmail.com');
```

Run this command. If everything is typed correctly you should see a message similar to this:

OBSERVE:

```
mysql> INSERT INTO Customers
-> (FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
-> VALUES ('John', 'Smith', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'john@hotmail.com')
Query OK, 1 row affected (0.01 sec)
```

Congratulations, you just added a row of data into your table!

The syntax for `INSERT` is fairly compact. First, you specify the target table for your data. In this case, `Customers`. Next, you list the specific columns for which you will provide data. There are commas between each column name, and they are enclosed within parentheses `()`.

Note You don't have to provide data for all columns - only the required (non-null) columns.

Next, you specify the **VALUES** for each column. Values that will be stored in text columns must have single quotation marks (') around them.

The DateAdded column in the Customers table must be the date, *right now*. Fortunately, MySQL provides a **NOW()** function that returns the date and time in the proper format for use in our table.

Note It's okay to change the order of your columns - see how DateAdded is before EmailAddress? Just remember to provide your **VALUES** in the order specified when you run the `INSERT` command.

Type the following at the MySQL prompt:

```
mysql> INSERT INTO Customers
(FirstName, LastName, DateAdded, EmailAddress)
VALUES ('Jane', 'Adams', NOW(), 'jane@hotmail.com');
```

Run this code. If you typed everything correctly, you should see the following error:

OBSERVE:

```
mysql> INSERT INTO Customers
-> (FirstName, LastName, DateAdded, EmailAddress)
-> VALUES ('Jane', 'Adams', NOW(), 'jane@hotmail.com');
ERROR 1364 (HY000): Field 'AddressLine1' doesn't have a default value
mysql>
```

While this error message may seem strange, everything is working perfectly. In the last lesson we defined the column "AddressLine1" to be *NOT NULL*. Since we didn't provide a value for AddressLine1 in our insert statement, the database defaulted it to NULL, resulting in an error.

WARNING

If you see something like this: Query OK, 1 row affected, 2 warnings (0.00 sec) You'll need to go back to the start of the lesson and make sure you typed the `SET sql_mode='TRADITIONAL';` command. You should also remove everything from Customers and start again - you can do so by typing `DELETE FROM Customers;` at the MySQL prompt.

Fix your INSERT statement by providing the AddressLine1 and City:

Type the following at the MySQL prompt:

```
mysql> INSERT INTO Customers
(FirstName, LastName, DateAdded, EmailAddress, AddressLine1, City)
VALUES ('Jane', 'Adams', NOW(), 'jane@hotmail.com', '300 N. Michigan', 'Chicago');
```

This time the insert will succeed.

OBSERVE:

```
mysql> INSERT INTO Customers
-> (FirstName, LastName, DateAdded, EmailAddress, AddressLine1, City)
-> VALUES ('Jane', 'Adams', NOW(), 'jane@hotmail.com', '300 N. Michigan', 'Chicago');
Query OK, 1 row affected (0.01 sec)

mysql>
```

Add more customers - try adding one with an AddressLine2.

SELECT

INSERTing data into tables is fine, but chances are you want to retrieve information as well. In SQL the keyword to get information out of a table is `SELECT`.

Type the following at the MySQL prompt:

```
mysql> SELECT * FROM Customers;
```

If everything goes well, you should see something like this:

```
OBSERVE:

mysql> SELECT * FROM Customers;
+-----+-----+-----+-----+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | AddressLine1 | AddressLine2 | City | State | PostalCode |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          21 | John     | Smith   | 123 4th Street | NULL         | Chicago | IL    | 60606      |
|          31 | Jane     | Adams   | 300 N. Michigan | NULL         | Chicago | NULL  | NULL       |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Your screen may not look too nice - in fact it might be pretty ugly! That's because MySQL may have to wrap the rows to fit your screen.

Note

The CustomerID values you see probably won't match your own CustomerIDs. This is because the column is an *auto increment* column.

In a SELECT statement the asterisk (*) means "all columns." Thus the SQL statement you entered could be translated to English this way: retrieve data from all columns, from the table called Customers.

Of course, the list returned from this SELECT statement is pretty long, so let's omit the address information for now.

```
Type the following at the MySQL prompt:

mysql> SELECT CustomerID, FirstName, LastName,
           EmailAddress FROM Customers;
```

This is much easier to handle. Your results should look something like this:

```
OBSERVE:

mysql> SELECT CustomerID, FirstName, LastName, EmailAddress FROM Customers;
+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | EmailAddress |
+-----+-----+-----+-----+
|          21 | John     | Smith   | john@hotmail.com |
|          31 | Jane     | Adams   | jane@hotmail.com |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

This time only the **CustomerID, FirstName, LastName, EmailAddress** columns are retrieved.

What if you only want to retrieve customers from Illinois? The SELECT statement can include a WHERE clause that enables you to filter your rows.

```
Type the following at the MySQL prompt:

mysql> SELECT CustomerID, FirstName, LastName, EmailAddress, State FROM Customers
WHERE State='IL';
```

The results should only include rows **WHERE** the State is equal to 'IL.'

```
OBSERVE:

mysql> SELECT CustomerID, FirstName, LastName, EmailAddress, State FROM Customers
-> WHERE State='IL';
+-----+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | EmailAddress | State |
+-----+-----+-----+-----+-----+
|          21 | John     | Smith   | john@hotmail.com | IL    |
+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

mysql>
```


This time Ms. Adams isn't included because we didn't specify a state for her.

What if you only want to see your customers from Illinois who have an email address at gmail? Remember that email addresses at gmail end with gmail.com.

Type the following at the MySQL prompt:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress, State FROM Customers
WHERE State='IL' AND EmailAddress LIKE '%gmail.com';
```

We don't have any customers in Illinois with gmail accounts, so no rows were returned.

OBSERVE:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress, State FROM Customers
-> WHERE State='IL' AND EmailAddress LIKE '%gmail.com';
Empty set (0.00 sec)

mysql>
```

Notice that this query shows something new. **AND** is a conjunction used to return only customers in Illinois *AND* with email addresses at gmail.

Note You can also use **OR** in the WHERE clause. Other comparisons you can do include **IS NULL**, **IS NOT NULL**, **!=** (not equals), and **<** or **>**.

LIKE doesn't do an exact match on a column. The special character **%** means "anything" to SQL, so `EmailAddress LIKE '%gmail.com'` could be translated to English as "EmailAddress that ends with gmail.com."

Update

Now you have data in your tables, but chances are that at some point this data might change. Suppose, for example, that your customer John Smith tells you that his email address has changed. It used to be 'john@hotmail.com,' but now it's 'john@gmail.com.' Let's **UPDATE** his row in the Customers table.

In the following example, be sure to change the **CustomerID** from **21** to whatever CustomerID your individual John Smith was given.

Type the following at the MySQL prompt:

```
mysql> UPDATE Customers
SET EmailAddress='john@gmail.com'
WHERE CustomerID=21;
```

If everything went OK, you should see:

OBSERVE:

```
mysql> UPDATE Customers
-> SET EmailAddress='john@gmail.com'
-> WHERE CustomerID=21;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql>
```

Update statements have two important parts - the **column updates**, and the **WHERE** clause. The **column updates** specify the new values for the given columns. The **WHERE** clause tells the database which rows in the database you want to update.

Let's check to make sure the data was updated.

Type the following at the MySQL prompt:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress FROM Customers;
```

Yes, it has been updated.

OBSERVE:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress FROM Customers;
+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | EmailAddress |
+-----+-----+-----+-----+
|          21 | John     | Smith   | john@gmail.com |
|          31 | Jane     | Adams   | jane@hotmail.com |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

You don't need a `WHERE` clause, but if one is not provided, every row in the database will be updated. If you want to update only one row, you'll need to specify enough conditions in your `WHERE` clause to limit your change to one row.

Tip If you're not sure how to limit your `UPDATE` statement correctly, write it as a `SELECT` statement first. If your `SELECT` statement returns the intended rows, chances are your `UPDATE` statement will only change the intended rows as well.

You can change multiple columns as well. Before we do, let's write a `SELECT` statement to make sure our `WHERE` clause is correct. Once again, be sure to change the `CustomerID` from `21` to whatever `CustomerID` your individual John Smith was given.

Type the following at the MySQL prompt:

```
mysql> SELECT *
FROM Customers
WHERE CustomerID=21;
```

One row was returned, with the correct customer.

OBSERVE:

```
mysql> SELECT *
-> FROM Customers
-> WHERE CustomerID=21;
+-----+-----+-----+-----+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | AddressLine1 | AddressLine2 | City | State | PostalCode | E
+-----+-----+-----+-----+-----+-----+-----+-----+
|          21 | John     | Smith   | 123 4th Street | NULL         | Chicago | IL | 60606 | j
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Now that we're confident that we've written a correct `WHERE` clause, let's rewrite the `SELECT` statement as an `UPDATE` statement.

Type the following at the MySQL prompt:

```
mysql> UPDATE Customers
SET EmailAddress='johnsmith@gmail.com', City='Evanston'
WHERE CustomerID=21;
```

Your results show that one row was matched, and one row was updated.

OBSERVE:

```
mysql> UPDATE Customers
-> SET EmailAddress='johnsmith@gmail.com', City='Evanston'
-> WHERE CustomerID=21;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql>
```

If you need additional confirmation that the change took place, write another `SELECT` query to check for yourself!

Delete

Now you have data in your tables and have altered that data, how do you get rid of it? Perhaps your customer Ms. Adams wants to be removed from your system. You may recall that in our database Ms. Adams has a `CustomerID` of `31`. Double check though - chances

are her CustomerID is different in your system.

Type the following at the MySQL prompt:

```
mysql> DELETE FROM Customers
WHERE CustomerID=31;
```

If everything went OK, you should see:

OBSERVE:

```
mysql> DELETE FROM Customers
-> WHERE CustomerID=31;
Query OK, 1 row affected (0.00 sec)

mysql>
```

Is the data really gone? Let's find out.

Type the following at the MySQL prompt:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress FROM Customers;
```

Yep, it's gone.

OBSERVE:

```
mysql> SELECT CustomerID, FirstName, LastName, EmailAddress FROM Customers;
+-----+-----+-----+-----+
| CustomerID | FirstName | LastName | EmailAddress |
+-----+-----+-----+-----+
|          21 | John      | Smith    | johnsmith@gmail.com |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Take a moment to reflect on the work you've just done. You added data to your tables, retrieved it, filtered it, updated some rows, and deleted some old records. In the next lab you'll learn how to keep your data secure and consistent through the use of transactions. See you in the next lesson!

Transactions

DBA 1: Introduction to Database Administration Lesson 5

Making sure your commands behave

Welcome back! In the last lesson you learned how to store and retrieve data from your database. you also learned to create tables in your database to track customers, inventory, and sales. In this lesson we'll discuss the steps you can take to make sure the data in your database is always correct and up-to-date, even if many people are working with the data at the same time.

Note

For this lesson you'll need to have two Unix Terminals open at the same time. In addition to position of the tabs to tell them apart, each will have it's own name and number, such as "Terminal1" or "Terminal2". Log into MySQL and be sure to set the 'TRADITIONAL' mode in each Unix Terminal! When you switch to a new terminal you will need to click in it before you can type in it.

Let's go back to our bookstore. When a book is sold, several things need to happen:

1. Check product inventory to make sure the item is in stock.
2. Create an order.
3. Add the product to the order item table.
4. Update inventory to reflect the new quantity in stock.
5. Check the inventory level again so the sales staff is alerted when stock gets low.

But what happens if two people try to purchase the same book at the same time? Who gets the book? What happens if you add inventory to the database when someone purchases a book?



In the database world, **transactions** are used to group related commands so all commands execute or all commands do not execute. There is no in-between.

ACID

Databases must maintain four properties: **Atomicity**, **Consistency**, **Isolation**, and **Durability**. To help you remember this, you can use the acronym **ACID**.

Atomicity: Transactions cannot be divided or split. Transactions either succeed or fail as a single unit.



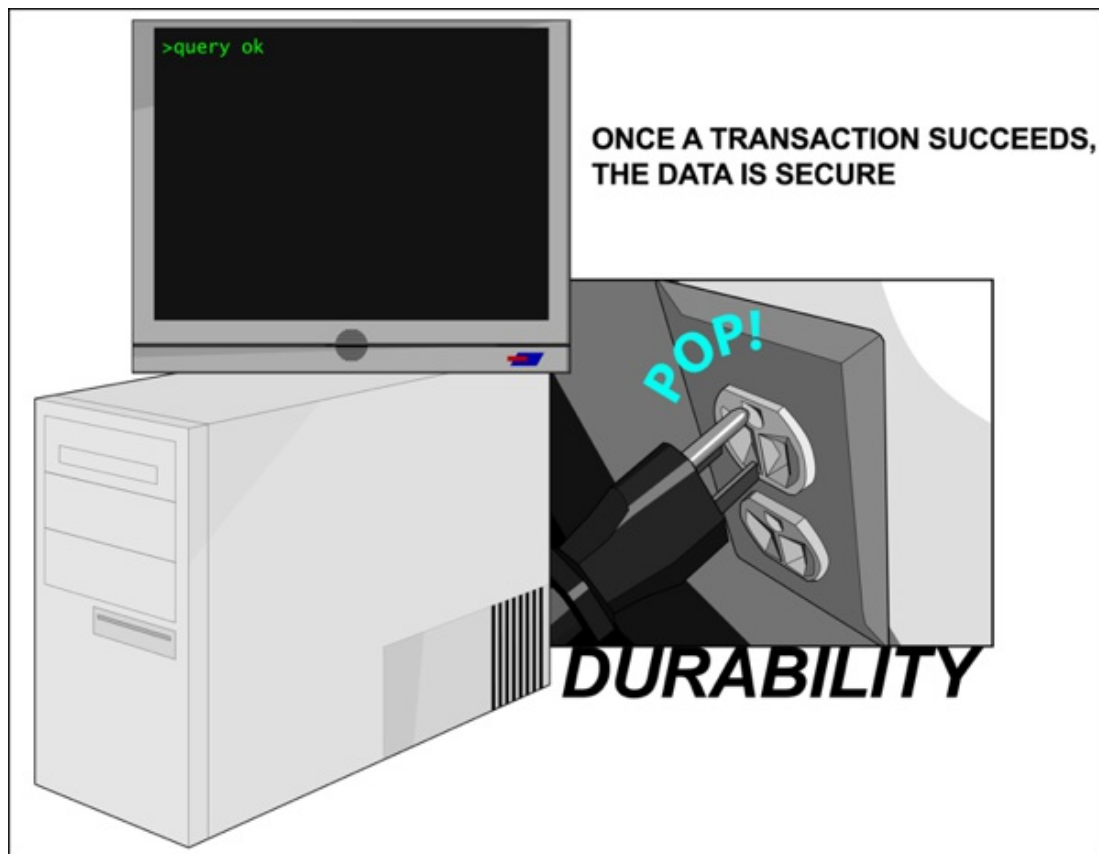
Consistency: Databases only store data that is consistent with the rules defined for storing data. For instance, if an inventory table isn't allowed to have negative quantities, then no rows will be allowed to have negative quantities.



Isolation: Transactions are independent of the outside world. No outside change can cause an unexpected change within our transaction. When making a sale, no other sales can change the quantity in inventory until the sale has been completed.



Durability: Once a database transaction commits, the data is secure in the database.



All databases handle Atomicity, Consistency, and Durability internally, automatically. The Isolation property is dependent on the user - it usually defaults to a secure level, but depending on your requirements, you can raise or lower it.

Note

MySQL is a unique database because it doesn't always support ACID properties. Depending on the version of MySQL being used and the way your database is setup, transactions may not secure your data. To ensure MySQL will secure your data, always use the **InnoDB** table type.

Transaction Isolation levels

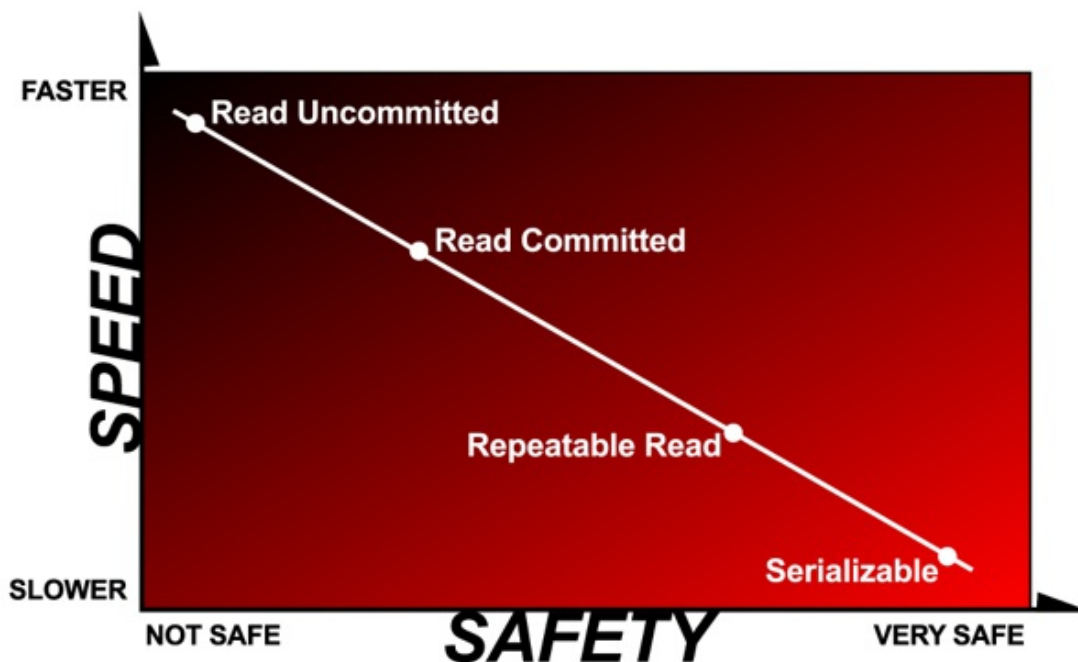
There are four primary transaction isolation levels supported by databases.

The first is called *read uncommitted*. It is the fastest, but least safe level. Transactions run with this isolation level are not guaranteed to occur independently of other transactions. This means that one sale transaction is allowed to "dirty read" the results of other incomplete sale transactions. (A "dirty read" is when the database gives you data that has not yet been committed to disk.) This isolation level is only used in certain circumstances where you don't necessarily care.

The next transaction level, called *read committed*, is slightly safer than read uncommitted. Dirty reads are not possible under this isolation level. If you run a query at the beginning of your transaction, and run the same query at the end of the transaction, the results won't be the same if another transaction begins and commits in the middle of your transaction.

Repeatable read is usually the default transaction isolation level. In this isolation level you cannot get dirty reads, and no other transactions can change rows during your transaction.

The highest level of safety, and the slowest transaction level, is *serializable*. Under this isolation level, each 'read' and 'write' to the database occurs in sequence. This isolation level can cause performance issues, since your transaction might have to wait for someone else's transaction to complete.



Using Transactions

Before you query the database, you should specify your transaction isolation level. In MySQL this is done via the `SET` command.

Be sure you are in the first of your two Unix Terminals before executing these first commands.

Type the following at the MySQL prompt in the first Unix Terminal:

```
mysql> SET transaction isolation level read committed;
```

This command doesn't return any rows, so MySQL will respond:

OBSERVE:

```
mysql> SET transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Most database servers allow you to process many SQL statements together as a single *batch*. In MySQL you begin a batch with the `START TRANSACTION` statement, end it with `COMMIT`, or undo it with `ROLLBACK`. As always, individual SQL statements must be terminated by a semicolon (;).

Let's create a simple transaction.

Type the following at the MySQL prompt in first Unix Terminal:

```
mysql> START TRANSACTION;
SELECT LastName FROM Customers;
COMMIT;
```

First, you `START TRANSACTION`. In this batch you are only executing one statement, to return the `count` from the `Customers` table. However, you could do additional work in your batch by adding more SQL statements. Finally, you end your batch with `COMMIT`;

If you typed everything correctly, you should see the following:

OBSERVE:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT LastName from Customers;
+-----+
| LastName |
+-----+
| Smith    |
+-----+
```

```
1 row in set (0.01 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

Note It's okay if the results returned from your query are different from this example.

Of course, this example only selects data from the database - it doesn't update or add data. What happens when you use a transaction with an `INSERT` statement? To find out, you'll use both Unix Terminals. Make sure you type these commands in the correct mode!

Type the following at the MySQL prompt in the first Unix Terminal:

```
mysql>
START TRANSACTION;
INSERT INTO Customers
(FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
VALUES ('John', 'Doe', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'johndoe@hotmail.com');
```

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

OBSERVE:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO Customers
-> (FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
-> VALUES ('John', 'Doe', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'johndoe@hotmail.com')
Query OK, 1 row affected (0.00 sec)

mysql>
```

Next, switch to the second Unix Terminal.

Type the following at the MySQL prompt in the second Unix Terminal:

```
mysql> SELECT FirstName, LastName from Customers;
```

Your table may have more rows than the table shown below. However you should see that **John Doe** is not there:

OBSERVE:

```
mysql> SELECT FirstName, LastName from Customers;
+-----+-----+
| FirstName | LastName |
+-----+-----+
| John      | Smith    |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

If your transaction isolation level was set correctly, you *won't* see any rows for John Doe. This is the correct result, since we set the **read committed** isolation level before we started the batch.

Note If you **do** see John Doe, you probably missed the step where you entered in the `SET transaction isolation level read committed;` Delete John Doe, and try again from the beginning.

Now switch back to the first Unix Terminal - where you typed in the `INSERT` statement.

Type the following at the MySQL prompt in the first Unix Terminal:

```
mysql> COMMIT;
```


MySQL will respond:

OBSERVE:

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Switch back to the second Unix Terminal - where you typed in the SELECT statement. Let's run it again.

Type the following at the MySQL prompt in the second Unix Terminal:

```
mysql> SELECT FirstName, LastName from Customers;
```

Now you see John Doe, since that transaction has committed.

OBSERVE:

```
mysql> SELECT FirstName, LastName from Customers;
+-----+-----+
| FirstName | LastName |
+-----+-----+
| John      | Smith    |
| John      | Doe      |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

There is one other way to end your batch - you can undo your changes by using the **ROLLBACK** statement.

Make sure you are in the first Unix Terminal.

Type the following at the MySQL prompt in the first Unix Terminal:

```
mysql>
START TRANSACTION;
INSERT INTO Customers
(FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
VALUES ('Becky', 'Stein', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'bstein@hotmail.com');
ROLLBACK;
```

You should see this:

OBSERVE:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO Customers
-> (FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAddress)
-> VALUES ('Becky', 'Stein', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'bstein@hotmail.co
Query OK, 1 row affected (0.01 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Now, switch to the second Unix Terminal.

Type the following at the MySQL prompt in the second Unix Terminal:

```
mysql> SELECT FirstName, LastName from Customers;
```

Since you **rolled back** the transaction, you won't see any rows for Becky Stein. You might see something similar to this:

OBSERVE:

```
mysql> SELECT FirstName, LastName from Customers;
+-----+-----+
| FirstName | LastName |
+-----+-----+
| John      | Smith    |
| John      | Doe      |
+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Committing and rolling back

You might be asking, "so when do I use COMMIT and ROLLBACK?"

Transactions are usually used in applications or in stored procedures. Your application will begin a transaction, then execute a query. If that query was successful, the application executes the next query. If any query fails, the transaction is rolled back. If all queries succeed, the transaction is committed.

Using transactions for this means your application doesn't have to try to undo any database work if there is a problem. The database does the hard work for you!

You've learned a very powerful way to make sure your data stays secure. In the next lesson you'll learn how to combine data from all of your tables in powerful and meaningful ways. **See you then!**

Joins

DBA 1: Introduction to Database Administration Lesson 6

Combining Tables with a Join

In the last lesson you learned about transactions. Now you'll learn how to combine data from many tables into a single query.

Wouldn't it be nice to be able to write a query that would allow you to see the current inventory of the book store? Until now the only way you had to find this information was by writing two queries: one to list the products in the store, and another to list the quantities in the store.

Before we begin this lesson, let's clear the products and inventory table and start fresh. You can do this by using the `DELETE` keyword you learned about earlier.

Type the following at the MySQL prompt:

```
mysql> DELETE FROM Inventory;  
mysql> DELETE FROM Products;
```

Depending on how many products you have, you might see the results below:

OBSERVE:

```
mysql> DELETE FROM Inventory;  
Query OK, 15 rows affected (0.07 sec)  
  
mysql> DELETE FROM Products;  
Query OK, 15 rows affected (0.00 sec)
```

Now that our tables are empty, let's create some new data. O'Reilly has a very nice book catalog, so let's borrow a few descriptions from that. We'll download this `sql` file to save us some time. This text file contains many `INSERT` statements that we'll submit to the database server - instead of typing each and every `INSERT` command ourselves.

First, we need to get back to the Unix prompt.

Type the following at the MySQL prompt:

```
mysql> exit;
```

Next, we'll grab the file from [O'Reilly's servers](http://courses.oreillyschool.com/dbal/downloads/products.sql) using the `curl` command.

Type the following at the Unix prompt:

```
cold:~$ curl -L http://courses.oreillyschool.com/dbal/downloads/products.sql > products.sql
```

Once that file has downloaded, we'll use the `mysql` command to import the products.

Type the following at the Unix prompt:

```
cold:~$ mysql -h sql -p -u certjosh certjosh < products.sql
```

If everything went correctly, you'll see no results:

OBSERVE:

```
cold:~$ mysql -h sql -p -u certjosh certjosh < products.sql  
Enter password:  
cold:~$
```

Next, insert the data for inventories. Log back into MySQL, and be sure to set your `sql_mode`!

Type the following at the MySQL prompt:

```
mysql> INSERT INTO Inventory values ('artofsql', 52);  
mysql> INSERT INTO Inventory values ('databaseid', 0);  
mysql> INSERT INTO Inventory values ('mysqlspp', 5);
```

```
mysql> INSERT INTO Inventory values ('sqlhks', 32);
mysql> INSERT INTO Inventory values ('sqltuning', 105);
```

If you typed everything correctly you should see this:

OBSERVE:

```
mysql> INSERT INTO Inventory values ('artofsql', 52);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO Inventory values ('databaseid', 0);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Inventory values ('mysqlspp', 5);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Inventory values ('sqlhks', 32);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO Inventory values ('sqltuning', 105);
Query OK, 1 row affected (0.01 sec)

mysql>
```

To combine two tables in one query, you use a **JOIN**. A join between the Products table and Inventory table will match every row in Products with every row in Inventory. Let's try it out!

Type the following at the MySQL prompt:

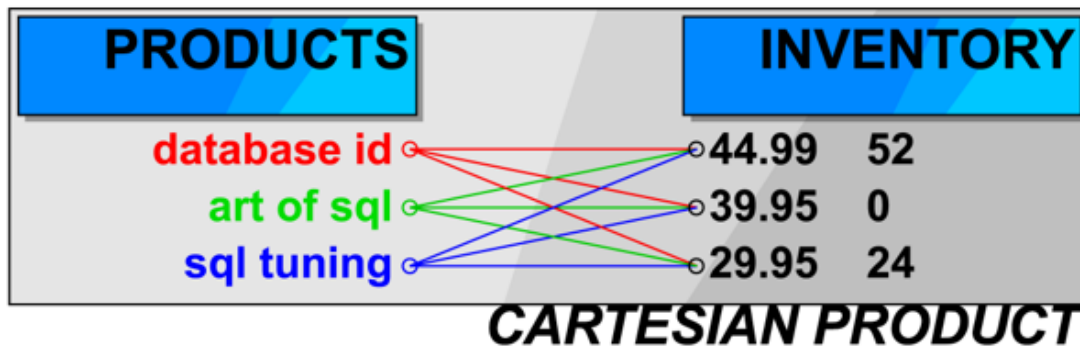
```
SELECT * FROM Products JOIN Inventory;
```

See the **JOIN** keyword? Run this query - the results may not be what you expected.

OBSERVE:

```
mysql> SELECT * FROM Products JOIN Inventory;
+-----+-----+-----+-----+
| ProductCode | Title | Category | Description |
+-----+-----+-----+-----+
| databaseid | Database in Depth | Database Theory | This book sheds light on the pri |
| sqlhks | SQL Hacks | SQL | Whether you're running Access, M |
| artofsql | The Art of SQL | Database Theory | For all the buzz about trendy IT |
| sqltuning | SQL Tuning | Database Theory | A poorly performing database app |
| mysqlspp | MySQL Stored Procedure Programming | MySQL | The implementation of stored pro |
| databaseid | Database in Depth | Database Theory | This book sheds light on the pri |
| sqlhks | SQL Hacks | SQL | Whether you're running Access, M |
| artofsql | The Art of SQL | Database Theory | For all the buzz about trendy IT |
| sqltuning | SQL Tuning | Database Theory | A poorly performing database app |
| mysqlspp | MySQL Stored Procedure Programming | MySQL | The implementation of stored pro |
| databaseid | Database in Depth | Database Theory | This book sheds light on the pri |
| sqlhks | SQL Hacks | SQL | Whether you're running Access, M |
| artofsql | The Art of SQL | Database Theory | For all the buzz about trendy IT |
| sqltuning | SQL Tuning | Database Theory | A poorly performing database app |
| mysqlspp | MySQL Stored Procedure Programming | MySQL | The implementation of stored pro |
| databaseid | Database in Depth | Database Theory | This book sheds light on the pri |
| sqlhks | SQL Hacks | SQL | Whether you're running Access, M |
| artofsql | The Art of SQL | Database Theory | For all the buzz about trendy IT |
| sqltuning | SQL Tuning | Database Theory | A poorly performing database app |
| mysqlspp | MySQL Stored Procedure Programming | MySQL | The implementation of stored pro |
+-----+-----+-----+-----+
25 rows in set (0.00 sec)
```

The database gave you the exact information you requested - it matched every row in the Products table with every row in the Inventory table. Five rows in Products times five rows in Inventory (5 x 5) = **25** rows. This is the *cartesian product* of those two tables. Graphically, a cartesian product may look something like the following:



For joins to be effective you need to specify how to link tables together. You do this by specifying the *join columns*.

Note You can join nearly any column, with nearly any data type. Usually we only want to join on our primary key (or another key) for performance reasons.

We know that the Products and Inventory tables have a common column: **ProductCode**. We'll join on this column. We'll also specify the columns we want to view instead of requesting all columns.

Type the following at the MySQL prompt:

```
mysql> SELECT Products.ProductCode, Products.Title, Products.Price, I.QuantityInStock as Qty
FROM Products
JOIN Inventory as I on (Products.ProductCode = I.ProductCode);
```

Notice that we listed the ProductCode column as **Products.ProductCode**? Different tables might have columns with the same names, however SQL requires us to reference a specific table and column. We do this by using the *TableName.ColumnName* syntax.

There is a shortcut available. You can use a *table alias* to specify a shorter name for a table. In our example we created an alias for Inventory called **I** by typing **as I**.

You can also specify a *column alias* using the same syntax. See how we renamed the **QuantityInStock** column?

Finally, we specify the columns we want to join upon by using the *on* keyword: **on (Products.ProductCode = I.ProductCode)**. Graphically, a join could look something like this:



If you typed everything correctly, you'll see the five rows from the Inventory and Products tables that have the same ProductCode. Your results should contain the same data, but it might be in different order. This is because we didn't ask the database to return the results in any particular order.

OBSERVE:

```
mysql> SELECT Products.ProductCode, Products.Title, Products.Price, I.QuantityInStock as Qty
-> FROM Products
-> JOIN Inventory as I on (Products.ProductCode = I.ProductCode);
+-----+-----+-----+-----+
| ProductCode | Title | Price | Qty |
+-----+-----+-----+-----+
| artofsql | The Art of SQL | 44.99 | 52 |
| databaseid | Database in Depth | 29.95 | 0 |
| mysqlspp | MySQL Stored Procedure Programming | 44.99 | 5 |
| sqlhks | SQL Hacks | 29.99 | 32 |
| sqltuning | SQL Tuning | 39.95 | 105 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Congratulations, you've written one query to get your product and inventory information!

Other Joins

Left Joins

What happens if someone at the bookstore has created a row in the Products table for a new book, but hasn't entered a corresponding row in Inventory?

Let's see for ourselves by adding one more book to the Product table. We'll download this one as well.

Type the following at the MySQL prompt:

```
mysql> exit;
```

Grab the file from [O'Reilly's servers](http://oreil.ly/servers) using the **curl** command.

Type the following at the Unix prompt in Unix:

```
cold:~$ curl -L http://courses.oreillyschool.com/dba1/downloads/products-1.sql > products-1.sql
```

Once that file has downloaded, use the **mysql** command to import the products.

Type the following at the Unix prompt:

```
cold:~$ mysql -h sql -p -u certjosh certjosh < products-1.sql
```

If everything went correctly, you'll see no results:

OBSERVE:

```
cold:~$ mysql -h sql -p -u certjosh certjosh < products-1.sql
Enter password:
cold:~$
```

Once you have that data imported, reconnect to MySQL. Be sure to set your `sql_mode` again.

Now run your previous query to retrieve the product quantities. Here it is again (using table aliases this time):

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
FROM Products as P
JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

You ran the insert statement, but your results are the same as before!

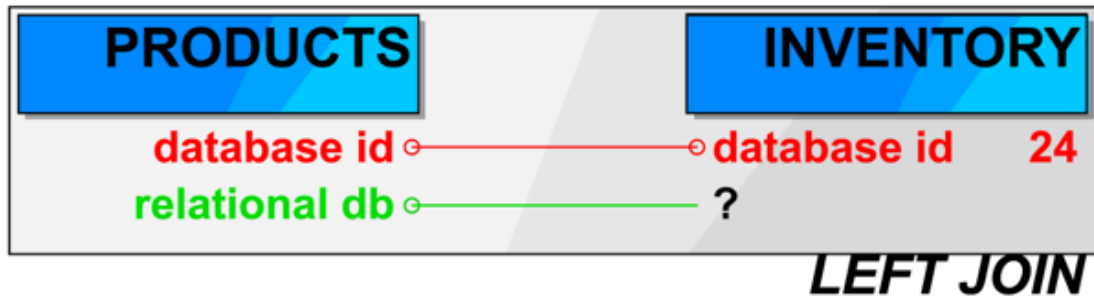
OBSERVE:

```
mysql> SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
-> FROM Products as P
-> JOIN Inventory as I on (P.ProductCode = I.ProductCode);
+-----+-----+-----+-----+
| ProductCode | Title                                | Price | Qty |
+-----+-----+-----+-----+
| artofsql    | The Art of SQL                      | 44.99 | 52 |
| databaseid  | Database in Depth                   | 29.95 | 0 |
| mysqlspp    | MySQL Stored Procedure Programming | 44.99 | 5 |
| sqlhks      | SQL Hacks                          | 29.99 | 32 |
| sqltuning   | SQL Tuning                          | 39.95 | 105 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

The database is doing exactly what you asked it to do - it's showing you every row from Inventory and Products where the product code is the same. Since the inventory table doesn't have a row with **ProductCode='relationaldb'**, no row is returned for that book.

It would be nice to see such a row though. Fortunately there is a way we can make this happen - the *LEFT JOIN*. The left join tells the database to return all rows from the first or LEFT table, even if a corresponding row doesn't exist in the second or RIGHT table. Graphically, a left join looks something like this:



Let's try a left join.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

Now the missing row from Products is returned, along with a NULL quantity since no corresponding row exists in the Inventory table. Your results may be sorted differently - that is OK.

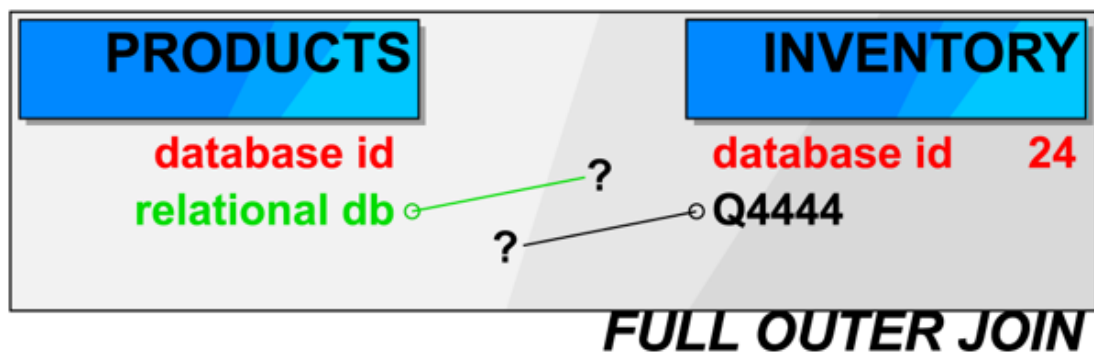
OBSERVE:

```
mysql> SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
+-----+-----+-----+-----+
| ProductCode | Title | Price | Qty |
+-----+-----+-----+-----+
| artofsql | The Art of SQL | 44.99 | 52 |
| databaseid | Database in Depth | 29.95 | 0 |
| mysqlspp | MySQL Stored Procedure Programming | 44.99 | 5 |
| relationaldb | The Relational Database Dictionary | 14.99 | NULL |
| sqlhks | SQL Hacks | 29.99 | 32 |
| sqltuning | SQL Tuning | 39.95 | 105 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>
```

Full Outer Joins / Union

What if you wanted to join the Products and Inventory tables, showing rows from Products that don't match rows in Inventory, and rows in Inventory that don't match rows in Products? This is called a *FULL OUTER JOIN*. Graphically, a full outer join looks something like this:



Before we try out a full outer join, let's create an entry in Inventory that doesn't match anything in Products.

Type the following at the MySQL prompt:

```
mysql> INSERT INTO Inventory VALUES ('Q4440', 14);
```

Note

Usually you wouldn't want a row to be in inventory unless there was already a corresponding row in Products. The database has a way you can enforce this rule, called a *foreign key constraint*. Stay tuned, we'll discuss this in a future course.

Make sure the query is successful - you should see something similar to Query OK, 1 row affected (0.05 sec).

Now let's try to query our tables.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, I.QuantityInStock, P.Title, P.Price
FROM Products as P
FULL OUTER JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

Try to run this query. Even if you typed everything correctly you'll get the message below.

OBSERVE:

```
mysql> SELECT P.ProductCode, I.QuantityInStock, P.Title, P.Price
-> from Products as P
-> FULL OUTER JOIN Inventory as I on (P.ProductCode = I.ProductCode);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
mysql>
```

Unfortunately MySQL doesn't support full outer joins. As with most programming languages, there is another way to perform this query. Fortunately MySQL does support the *UNION* keyword.

A *union* will combine two select statements into one resulting data set. The only restriction is that the two queries must return the same number of columns of the same data type. For our full outer join problem, we can use our previous LEFT JOIN query to retrieve Products and Inventory, and union those results with a new query that is just the opposite: a LEFT JOIN of Inventory on Products.

First, let's write a LEFT JOIN query of Inventory on Products. We'll make sure to specify exactly the same columns as our previous query, except *ProductCode*. We'll reference the *ProductCode* from Inventory instead of Products this time.

Type the following at the MySQL prompt:

```
mysql> SELECT I.ProductCode, I.QuantityInStock, P.Title, P.Price
FROM Inventory as I
LEFT JOIN Products as P on (P.ProductCode = I.ProductCode)
WHERE P.ProductCode IS NULL;
```

This query is nearly the same as one we previously wrote, with one major difference. For this query we have a **WHERE** clause that limits our data set to rows from Inventory that don't have corresponding rows in Products. If we didn't include this WHERE clause, we would get duplicates in our UNIONed data set because our previous query includes matching rows.

If you typed the query correctly, you should see this data set:

OBSERVE:

```
mysql> SELECT I.ProductCode, I.QuantityInStock, P.Title, P.Price
-> FROM Inventory as I
-> LEFT JOIN Products as P on (P.ProductCode = I.ProductCode)
-> WHERE P.ProductCode IS NULL;
+-----+-----+-----+-----+
| ProductCode | QuantityInStock | Title | Price |
+-----+-----+-----+-----+
| Q4440      | 14              | NULL  | NULL  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

This is exactly what we want - only the rows from Inventory that don't correspond to rows in Products. Now we can combine our two queries.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, I.QuantityInStock, P.Title, P.Price
```



```

FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
UNION ALL
SELECT I.ProductCode, I.QuantityInStock, P.Title, P.Price
FROM Inventory as I
LEFT JOIN Products as P on (P.ProductCode = I.ProductCode)
WHERE P.ProductCode IS NULL;

```

Our previous query is in **green**, the **UNION ALL** keyword is next, and our new query is in **blue**. If you typed everything correctly you'll see these results:

OBSERVE:			
ProductCode	QuantityInStock	Title	Price
artofsql	52	The Art of SQL	44.99
databaseid	0	Database in Depth	29.95
mysqlspp	5	MySQL Stored Procedure Programming	44.99
relationaldb	NULL	The Relational Database Dictionary	14.99
sqlhks	32	SQL Hacks	29.99
sqltuning	105	SQL Tuning	39.95
Q4440	14	NULL	NULL

7 rows in set (0.02 sec)

This is our full outer join - rows in Products that don't match rows in Inventory, and rows in Inventory that don't match rows in Products. The first six rows (in **green**) are from the first query, and the last row (in **blue**) is from the second query.

You've learned a lot in this lesson! In the next lesson you'll learn how to further combine your data in meaningful ways to answer many new business questions. See you there!

Aggregates, Functions and Conditionals

DBA 1: Introduction to Database Administration Lesson 7

In the previous lesson you learned new ways to combine your data. In this lesson you'll learn several more ways you can query your data.

Aggregating Data

Aggregate functions create summaries from your rows and columns of data. The simplest aggregate function is *COUNT*, which is a row count. Let's try it out by retrieving the number of products (rows) in the Products table.

Type the following at the MySQL prompt:

```
mysql> SELECT COUNT(*) FROM Products;
```

If you haven't changed the Products table from the last lesson, you should see this:

OBSERVE:

```
mysql> SELECT COUNT(*) FROM Products;
+-----+
| COUNT(*) |
+-----+
|         6 |
+-----+
1 row in set (0.00 sec)

mysql>
```

Here the **COUNT** aggregate has given us the number of rows in the Products table. It would be nice to be able to answer more complex questions, such as "How many products are in each category?" To answer that question, we must *GROUP BY* the Category column on the Products table.

Type the following at the MySQL prompt:

```
mysql> SELECT Category, COUNT(*) as NumberOfProducts
FROM Products
GROUP BY Category;
```

When you run the query you will see the results below.

OBSERVE:

```
mysql> SELECT Category, COUNT(*) as NumberOfProducts
-> FROM Products
-> GROUP BY Category;
+-----+-----+
| Category          | NumberOfProducts |
+-----+-----+
| Database Theory   | 3               |
| MySQL             | 1               |
| SQL               | 2               |
+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

There are three books in the "Database Theory" category, one in "MySQL," and two in "SQL."

What if you wanted to find the average price *and* the maximum price of the products in each category? You can use multiple aggregates in the same query.

Type the following at the MySQL prompt:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
FROM Products
GROUP BY Category;
```

In this query we've specified the columns on which we would like to apply the aggregate. The **AVG** aggregate is over the Price column, and so is the **MAX** aggregate. Your results should look like this:

OBSERVE:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
-> FROM Products
-> GROUP BY Category;
+-----+-----+-----+
| Category      | AveragePrice | MaximumPrice |
+-----+-----+-----+
| Database Theory | 38.296667    | 44.99        |
| MySQL         | 44.990000    | 44.99        |
| SQL           | 22.490000    | 29.99        |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

MySQL allows you to omit the GROUP BY clause in some queries. If you omit GROUP BY your query may not return the same results however. Try it:

Type the following at the MySQL prompt:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
FROM Products;
```

This time you will only see one result row:

OBSERVE:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
-> FROM Products;
+-----+-----+-----+
| Category      | AveragePrice | MaximumPrice |
+-----+-----+-----+
| Database Theory | 37.974000    | 44.99        |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

If you want, you can disable this MySQL-specific behavior by changing the `SQL_MODE`.

Type the following at the MySQL prompt:

```
mysql> set SQL_MODE='ONLY_FULL_GROUP_BY';
```

If you typed the prior line correctly you will see Query OK, 0 rows affected (0.00 sec).

Now when you run a query without a GROUP BY you will see an error.

Type the following at the MySQL prompt:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
FROM Products;
```

The error will look something like:

OBSERVE:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
-> FROM Products;
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP columns is illegal
```

For more information about MySQL's use of GROUP BY, see [MySQL's web site](#).

There are many more useful aggregate functions, such as `SUM` and `MIN`. See [MySQL's web site](#) for additional aggregate functions. Be sure to take some time and experiment with these!

Functions

While aggregates allow you to summarize many rows of information, functions allow you to create brand new columns of data. The store keeps track of products, and the number of products in inventory. Suppose your manager wants to know the dollar value of the products in stock.

In the last lesson we used a join to show the products along with the quantity in stock. Let's write that query again.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, P.Price, I.QuantityInStock as Qty
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

Run this query, you'll see familiar results:

OBSERVE:

```
SELECT P.ProductCode, P.Price, I.QuantityInStock as Qty
-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
+-----+-----+-----+
| ProductCode | Price | Qty |
+-----+-----+-----+
| artofsql   | 44.99 | 52 |
| databaseid | 29.95 | 0 |
| mysqlspp   | 44.99 | 5 |
| relationaldb | 14.99 | NULL |
| sqlhks     | 29.99 | 32 |
| sqltuning  | 39.95 | 105 |
+-----+-----+-----+
6 rows in set (0.00 sec)
```

The dollar value for a product is the Price times the Quantity. Let's add this "Dollar Value" to our query.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, P.Price, I.QuantityInStock as Qty, P.Price * I.QuantityInStock as DollarValue
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

This query has a new **DollarValue** column which is the product of Price and Quantity. The results should look like this:

OBSERVE:

```
SELECT P.ProductCode, P.Price, I.QuantityInStock as Qty, P.Price * I.QuantityInStock as DollarValue
-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
+-----+-----+-----+-----+
| ProductCode | Price | Qty | DollarValue |
+-----+-----+-----+-----+
| artofsql   | 44.99 | 52 | 2339.48 |
| databaseid | 29.95 | 0 | 0.00 |
| mysqlspp   | 44.99 | 5 | 224.95 |
| relationaldb | 14.99 | NULL | NULL |
| sqlhks     | 29.99 | 32 | 959.68 |
| sqltuning  | 39.95 | 105 | 4194.75 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

It probably looks strange to have a NULL DollarValue. But it is the desired result, since Qty is NULL for that row.

So how can we find the total dollar value for our products? Let's combine our function with the SUM aggregate!

Type the following at the MySQL prompt:

```
mysql> select P.ProductCode, P.Price, I.QuantityInStock as Qty, SUM(P.Price * I.QuantityInStock) as Do
from Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

This query isn't correct. We are asking for a summary of all rows in the Products and Inventory tables, along with the ProductCode, Price, and QuantityInStock. This doesn't make sense - what is the ProductCode for the **total sum**? The database cannot give you an

answer, so if you try to run the query you'll see an error such as:

```
OBSERVE:

mysql> SELECT P.ProductCode, P.Price, I.QuantityInStock as Qty, SUM(P.Price * I.QuantityInStock) as Do
-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP columns is illegal
mysql>
```

To correct this error you have two options - you can remove the extra columns from your query, or you can GROUP BY those columns. Since we are looking for the total of all rows, let's remove the columns.

```
Type the following at the MySQL prompt:

mysql> SELECT SUM(P.Price * I.QuantityInStock) as TotalDollarValue
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

This query is much better, and gives us the desired result.

```
OBSERVE:

mysql> SELECT SUM(P.Price * I.QuantityInStock) as TotalDollarValue
-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
+-----+
| TotalDollarValue |
+-----+
|          7718.86 |
+-----+
1 row in set (0.00 sec)

mysql>
```

MySQL has many functions that operate on strings, numbers, and dates. You can find more information about these functions at [the MySQL web site](#).

Conditionals

Now that you know about aggregates and functions you're able to answer some interesting questions! Suppose your boss has a new inventory problem for you to solve. If there are less than 10 products in the store, she wants the store manager to reorder. If there are between 10 and 50 products, the inventory level is fine. If there are 50 or more products, the manager needs to send the extra stock to the warehouse.

How do we query the database to find the details about the inventory? One way is to use a CASE statement.

```
Type the following at the MySQL prompt:

mysql> SELECT P.ProductCode, P.Title, I.QuantityInStock as Qty,
CASE WHEN I.QuantityInStock < 10 THEN 'Reorder'
      WHEN I.QuantityInStock >= 10 AND I.QuantityInStock < 50 THEN 'In Stock'
      WHEN I.QuantityInStock >= 50 THEN 'Extra to warehouse'
END as Action
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

CASE statements have two parts -- **comparisons (in BLUE)** and **results (in GREEN)**. **Comparisons** are checked in order. As soon as a TRUE comparison is found, the corresponding **result** is returned.

Try this query. As long as you didn't make any typing mistakes, you should see these results:

```
OBSERVE:

+-----+-----+-----+-----+
| ProductCode | Title | Qty | Action |
+-----+-----+-----+-----+
| artofsql    | The Art of SQL | 52 | Extra to warehouse |
| databaseid  | Database in Depth | 0 | Reorder |
| mysqlspp    | MySQL Stored Procedure Programming | 5 | Reorder |
| relationaldb | The Relational Database Dictionary | NULL | NULL |
```

sqlhks	SQL Hacks	32	In Stock	
sqltuning	SQL Tuning	105	Extra to warehouse	
+-----+-----+-----+-----+				
6 rows in set (0.01 sec)				

This is almost the result we are looking for, however that NULL for "The Relational Database Dictionary" is troubling. Since we have never had inventory for that book, we should set the action to be "Place initial order." But remember, we're dealing with NULLs so we have to be careful.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, P.Title, IFNULL(I.QuantityInStock, 0) as Qty,
CASE WHEN I.QuantityInStock IS NULL THEN 'Place Initial Order'
      WHEN I.QuantityInStock < 10 THEN 'Reorder'
      WHEN I.QuantityInStock >= 10 AND I.QuantityInStock < 50 THEN 'In Stock'
      WHEN I.QuantityInStock >= 50 THEN 'Extra to warehouse'
END as Action
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

In this query we use the **IFNULL** conditional to check the QuantityInStock column. NULLs in that column are replaced by zero (0). We also added a comparison to the CASE statement - **I.QuantityInStock IS NULL** - to display the action "Place Initial Order."

Run this query, your results will look something like this:

OBSERVE:

+-----+-----+-----+-----+				
ProductCode	Title	Qty	Action	
+-----+-----+-----+-----+				
artofsql	The Art of SQL	52	Extra to warehouse	
databaseid	Database in Depth	0	Reorder	
mysqlspp	MySQL Stored Procedure Programming	5	Reorder	
relationaldb	The Relational Database Dictionary	0	Place Initial Order	
sqlhks	SQL Hacks	32	In Stock	
sqltuning	SQL Tuning	105	Extra to warehouse	
+-----+-----+-----+-----+				
6 rows in set (0.00 sec)				

In this lesson you learned how to combine data from your columns in meaningful ways to create new columns of data and summaries of rows. In the next lesson you'll learn one way to store your new queries in the database for future use. See you then!

Sub Queries and Views

DBA 1: Introduction to Database Administration Lesson 8

Over the past few lessons we've learned more complex ways to query our databases. In this lesson we'll examine one more way we can query our databases, as well as a way to save our complex queries.

Querying Queries

Your small book store doesn't have a lot of storage room. You occasionally need to query the database to find out which products you have the highest quantity of in inventory. Remember when we wrote a query to return the inventory levels for each product in our catalog? Let's try it again, just in case you forgot.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, I.QuantityInStock as Qty, P.Title, P.Price
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

If you haven't changed your database too much you might see results like this:

OBSERVE:

```
+-----+-----+-----+-----+
| ProductCode | Qty | Title | Price |
+-----+-----+-----+-----+
| artofsql | 52 | The Art of SQL | 44.99 |
| databaseid | 0 | Database in Depth | 29.95 |
| mysqlspp | 5 | MySQL Stored Procedure Programming | 44.99 |
| relationaldb | NULL | The Relational Database Dictionary | 14.99 |
| sqlhks | 32 | SQL Hacks | 29.99 |
| sqltuning | 105 | SQL Tuning | 39.95 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

You could scan the returned items to find the one with the largest quantity. But it would be even better to return a single row, like this:

OBSERVE:

```
+-----+-----+-----+-----+
| ProductCode | Qty | Title | Price |
+-----+-----+-----+-----+
| sqltuning | 105 | SQL Tuning | 39.95 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

We learned earlier about aggregate functions such as MAX. Let's try using it to return the desired result.

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, MAX(I.QuantityInStock) as Qty, P.Title, P.Price
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
GROUP BY I.ProductCode, P.Title, P.Price;
```

Since we're using the **MAX** aggregate with non-aggregate columns, we need to use a corresponding **GROUP BY** clause. In English, the **GROUP BY** would roughly translate to "for each ProductCode, Title and Price combination GROUP, show me the MAX(Price)."

Try running the query.

OBSERVE:

```
+-----+-----+-----+-----+
| ProductCode | Qty | Title | Price |
+-----+-----+-----+-----+
| artofsql | 52 | The Art of SQL | 44.99 |
| databaseid | 0 | Database in Depth | 29.95 |
| mysqlspp | 5 | MySQL Stored Procedure Programming | 44.99 |
```

```
| relationaldb| NULL | The Relational Database Dictionary | 14.99 |
| sqlhks     | 32  | SQL Hacks                          | 29.99 |
| sqltuning  | 105 | SQL Tuning                         | 39.95 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

The order may not be the same, but the same six rows are returned. And as it turns out we answered a different question - for each row in Products, we returned the corresponding MAX quantity from Inventory. We really just wanted the product with the maximum quantity in Inventory.

In order to return this information, let's try to restrict our query in the WHERE clause - something like this:

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, I.QuantityInStock as Qty, P.Title, P.Price
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
WHERE I.QuantityInStock = MAX(I.QuantityInStock);
```

This query isn't quite right either. If you try to run it you'll see the error ERROR 1111 (HY000): Invalid use of group function. Fear not, there is hope. We can correct this error if we use a *sub query*.

If you think about tables and query results, you'll realize that they are essentially the same. Both contain rows and columns of data. It would seem logical to be able to query the results from a query. And as the programming gods would have it, you can! You do this using a *sub query*.

In order to return the product with the greatest quantity, let's query the results of a query. But before we do that, let's make sure that the sub query correct. First, find the maximum QuantityInStock in Inventory.

Type the following at the MySQL prompt:

```
mysql> SELECT MAX(QuantityInStock) FROM Inventory;
```

You should see one row, something similar to this:

OBSERVE:

```
mysql> SELECT MAX(QuantityInStock) FROM Inventory;
+-----+
| MAX(QuantityInStock) |
+-----+
|                    105 |
+-----+
1 row in set (0.00 sec)

mysql>
```

That looks good! Now that we know the MAX(QuantityInStock) from Inventory, the query could essentially be rewritten like so:

OBSERVE:

```
SELECT P.ProductCode, I.QuantityInStock as Qty, P.Title, P.Price
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
WHERE I.QuantityInStock = 105;
```

While this query returns the desired result, in real life this might not work. After all, the QuantityInStock might change before we get a chance to run the query. Instead let's use a sub query. Type the following query into the editor:

Type the following at the MySQL prompt:

```
mysql> SELECT P.ProductCode, I.QuantityInStock as Qty, P.Title, P.Price
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
WHERE I.QuantityInStock = (SELECT MAX(QuantityInStock) FROM Inventory);
```

Run this query. You should see something similar to the results below:

OBSERVE:

```
mysql> SELECT P.ProductCode, I.QuantityInStock as Qty, P.Title, P.Price
```



```

-> FROM Products as P
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode)
-> WHERE I.QuantityInStock = (SELECT MAX(QuantityInStock) FROM Inventory);
+-----+-----+-----+-----+
| ProductCode | Qty | Title | Price |
+-----+-----+-----+-----+
| sqltuning | 105 | SQL Tuning | 39.95 |
+-----+-----+-----+-----+
1 row in set (0.02 sec)

mysql>

```

There you have it! Exactly the results we wanted.

Views

We've learned a lot of complex ways to write queries. It can be cumbersome to rewrite these queries. Since query results look just like a table, wouldn't it be nice to be able to store a query definition in the database, and access it just like a table?

Views solve our problem. *Views* are a great way to tuck away complex query logic into the database. Database Administrators love views because they can be used to hide sensitive columns of data (such as social security numbers) from certain users.

Creating a view

Remember back in lesson five when you used a join to query the products and inventory tables? Let's run that query again.

Type the following at the MySQL prompt:

```

mysql> SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);

```

The data in your tables may have changed, but the results should look similar to this:

OBSERVE:

```

+-----+-----+-----+-----+
| ProductCode | Title | Price | Qty |
+-----+-----+-----+-----+
| artofsql | The Art of SQL | 44.99 | 52 |
| databaseid | Database in Depth | 29.95 | 0 |
| mysqlspp | MySQL Stored Procedure Programming | 44.99 | 5 |
| relationaldb | The Relational Database Dictionary | 14.99 | NULL |
| sqlhks | SQL Hacks | 29.99 | 32 |
| sqltuning | SQL Tuning | 39.95 | 105 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

This data is very useful, but typing that query is cumbersome. We'll store the query in a new view called *ProdInventory*. The syntax for creating a view is pretty straightforward: `CREATE VIEW name AS query`.

Type the following at the MySQL prompt:

```

mysql> CREATE VIEW ProdInventory AS
SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
from Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);

```

If you typed everything correctly, you'll see something like this:

OBSERVE:

```

mysql> CREATE VIEW ProdInventory AS
SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
FROM Products as P LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
Query OK, 0 rows affected (0.07 sec)

mysql>

```

That's it! You created the view.

Since a view is more or less like a table, you can query it like a table.

Type the following at the MySQL prompt:

```
mysql> SELECT * FROM ProdInventory;
```

The results should be just as before:

OBSERVE:

```
mysql> SELECT * FROM ProdInventory;
+-----+-----+-----+-----+
| ProductCode | Title | Price | Qty |
+-----+-----+-----+-----+
| artofsql    | The Art of SQL | 44.99 | 52 |
| databaseid  | Database in Depth | 29.95 | 0 |
| mysqlspp    | MySQL Stored Procedure Programming | 44.99 | 5 |
| relationaldb | The Relational Database Dictionary | 14.99 | NULL |
| sqlhks      | SQL Hacks | 29.99 | 32 |
| sqltuning   | SQL Tuning | 39.95 | 105 |
+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>
```

Restrictions on views

Views are nearly identical to tables, however, there are some important differences.

First, you cannot always `INSERT` into a view. If your view contains joins, you won't be able to insert data into all of the joined tables. You may be able to insert into one of the tables, but it's usually best to avoid `INSERTs` to views.

Second, views are sensitive to changes in the database. For example, in the `ProdInventory` view, if the `Title` column is changed to "ProductTitle," the view will give an error.

Dropping a view

The view we created before is nice, however, the name is a little confusing. Instead of calling it "ProdInventory" (which might be interpreted as "Production Inventory"), let's call it "ProductInventory." Removing a view from the database is similar to removing a table. You use the **`DROP VIEW IF EXISTS`** command.

Type the following at the MySQL prompt:

```
mysql> DROP VIEW IF EXISTS ProdInventory;
```

When the view is dropped you will see this:

OBSERVE:

```
mysql> DROP VIEW IF EXISTS ProdInventory;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

We are now free to add our view called "ProductInventory."

Type the following at the MySQL prompt:

```
mysql> CREATE VIEW ProductInventory AS
SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
FROM Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

As long as you typed everything correctly you'll see the standard `Query OK, 0 rows affected (0.00 sec)` response.

Now you've learned another way to query your data. You've also learned about the power of views - another way the database works for you instead of you working for your database. In the next lesson we'll expand on that idea and learn how to store complex logic in the database. Stay tuned!

Stored Procedures

DBA 1: Introduction to Database Administration Lesson 9

In the last lesson we learned how to use a view to save complex query logic in the database. While views are really useful, they can't be used to store a sequence of steps that sometimes need to be performed with data. In this lesson you'll learn how to use *stored procedures*.

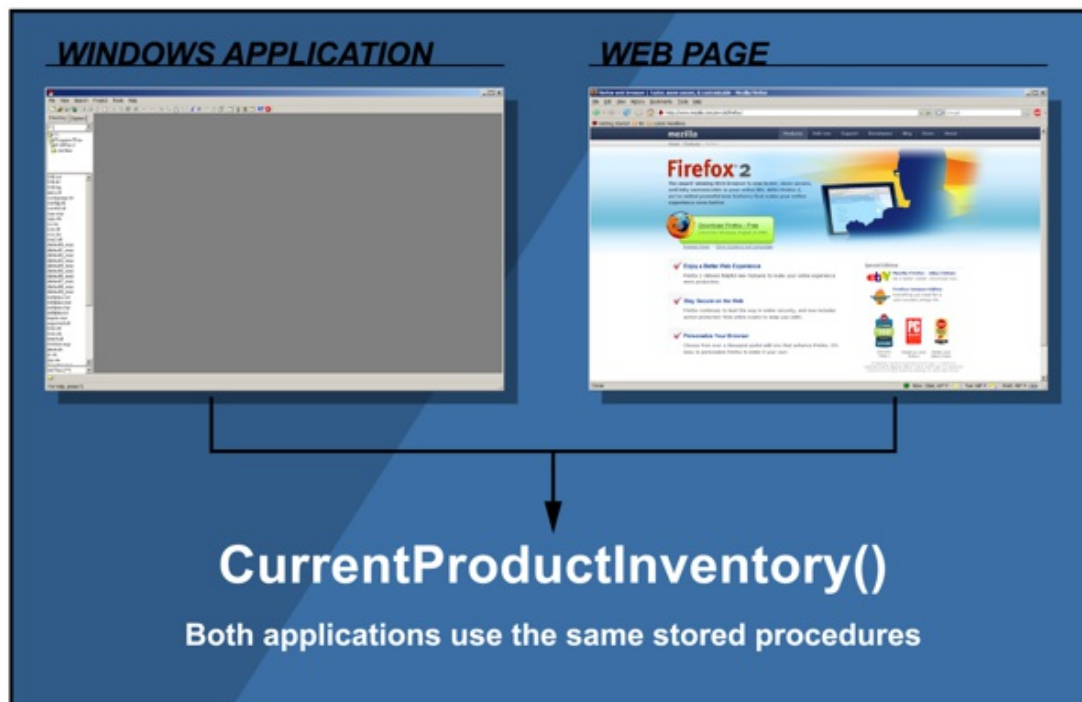
Using Stored Procedures

Motivations

Stored procedures are like functions for the database. They have a set of parameters, perform a set of tasks, and may return a set of results. Code reuse is important, and stored procedures provide a kind of code reuse that is similar to what functions and classes do in other languages.

Technologies (especially web technologies) come and go, but data doesn't change much. The process of tracking store inventory is essentially the same today as it was in 1997, 1987, or 1977. But the technology has changed pretty dramatically - inventory tracking used to be done on paper in 1978, then maybe on a green-screen computer in 1988, and then on a windows application in 1998, but in 2008 inventory tracking is likely to be done using a web browser *and* a windows application. Stored procedures created back in 1988 would still function in 2008, twenty years and three user interfaces later because data itself hasn't change much, but the front-end technology sure did!

Stored procedures, in addition to helping us track data, are also used to secure data. In business, for example, access to database tables could be disallowed specifically to make sure that access is gained only through a stored procedures. Since the database user is known to the server, sensitive information (such as hourly salary rates) wouldn't be returned to users outside of the human resources staff. This security applies to all access through a windows application, through a web site, or through command line tools.



Stored procedures have yet another advantage - speed. By keeping logic next to the data, database servers don't have to send massive amounts of information to applications (and back) for processing. This might not matter for small databases with only a few hundred rows, but it is critical for databases that contain millions or billions of rows.

Creating Stored Procedures

Setup

Before we can create our procedure we'll have to make a small change to our MySQL environment. MySQL usually expects a semicolon at the end of a statement. Since stored procedures can encapsulate many statements, we'll have to tell MySQL to use something else to separate statements as we enter them. To do this we'll use the special keyword *delimiter*.

Type the following at the MySQL prompt:

```
mysql> DELIMITER //
```

Run this command. If everything went well, you won't notice any difference right away.

OBSERVE:

```
mysql> DELIMITER //  
mysql>
```

You'll notice a difference after you run a query though.

Type the following at the MySQL prompt:

```
mysql> SELECT ProductCode,Title FROM Products;
```

After you press enter, not much will happen. Your screen will likely look like the following:

OBSERVE:

```
mysql> SELECT ProductCode,Title FROM Products;  
->
```

To submit the query to the server you need to enter the delimiter, *//*. Once you do your query will be evaluated, and you'll see some results.

Type the following at the MySQL prompt:

```
mysql> SELECT ProductCode,Title FROM Products;  
//
```

OBSERVE:

```
mysql> SELECT ProductCode,Title FROM Products;  
-> //  
+-----+-----+  
| ProductCode | Title |  
+-----+-----+  
| artofsql    | The Art of SQL |  
| databaseid  | Database in Depth |  
| mysqlspp    | MySQL Stored Procedure Programming |  
| relationaldb | The Relational Database Dictionary |  
| sqlhks      | SQL Hacks |  
| sqltuning   | SQL Tuning |  
+-----+-----+  
6 rows in set (0.00 sec)  
  
mysql>
```

Duplicating our View

A perfectly valid use of a stored procedure is to store a query for repeated use. A view could be used for this use as well, however most databases can optimize stored procedures in ways that are not possible with view or ad-hoc queries.

Let's take our ProductInventory view and turn it into a basic stored procedure. Below is the query behind that view. Try it out to refresh your memory.

OBSERVE:

```
SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty  
FROM Products as P  
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

Let's dive right in and create our procedure. We'll do so by using the *CREATE PROCEDURE* keyword.

Type the following at the MySQL prompt:

```
mysql> CREATE PROCEDURE CurrentProductInventory ()  
BEGIN  
SELECT P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty  
FROM Products AS P  
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
```

```
END;  
//
```

If everything was typed in correctly, you will see the following response:

OBSERVE:

```
mysql> CREATE PROCEDURE CurrentProductInventory ()  
-> BEGIN  
-> select P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty  
-> from Products as P  
-> LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);  
-> END;  
-> //  
Query OK, 0 rows affected (0.00 sec)  
  
mysql>
```

First we used the **CREATE PROCEDURE** keyword, followed by the name of our procedure - **CurrentProductInventory**. A pair of empty parentheses **()** tells MySQL that our procedure won't have any parameters.

Next, the **BEGIN** keyword tells MySQL that we are going to write one or more statements that should be treated as a block. The **END;** keyword marks the end of that block. Finally, we end our entry with the **//** delimiter.

Now let's run the procedure, using the **CALL** keyword. We still need to type in the delimiters, since we haven't reset that MySQL option.

Type the following at the MySQL prompt:

```
mysql> CALL CurrentProductInventory ();  
//
```

Make sure you type the procedure name - **CurrentProductInventory** - correctly, and that you type in an empty set of parentheses **()** to tell MySQL that our procedure doesn't take any parameters.

If everything is typed in correctly, you'll see these results:

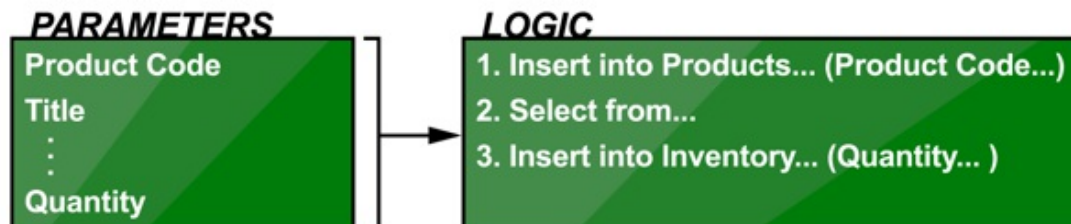
OBSERVE:

```
mysql> CALL CurrentProductInventory ();  
-> //  
  
+-----+-----+-----+-----+  
| ProductCode | Title | Price | Qty |  
+-----+-----+-----+-----+  
| artofsql | The Art of SQL | 44.99 | 52 |  
| databaseid | Database in Depth | 29.95 | 0 |  
| mysqlspp | MySQL Stored Procedure Programming | 44.99 | 5 |  
| relationaldb | The Relational Database Dictionary | 14.99 | NULL |  
| sqlhks | SQL Hacks | 29.99 | 32 |  
| sqltuning | SQL Tuning | 39.95 | 105 |  
+-----+-----+-----+-----+  
6 rows in set (0.00 sec)  
  
Query OK, 0 rows affected (0.00 sec)  
  
mysql>
```

Looks good!

Parameters

Using a stored procedure to return results like a view can be helpful, however, stored procedures are much more powerful if they accept parameters.



Back in the first few lessons we designed a store structure to keep track of products and inventory in two tables, then we entered data into those two tables manually. While this process will work, it's clunky and error prone. Instead of adding products to the system manually into two tables, we could use a stored procedure.

Adding a product requires inserting data into two tables - Products and Inventory. Done separately, the queries look like this:

```
OBSERVE:

INSERT INTO Products (ProductCode, Title, Category, Description, Price)
VALUES ('mysqlian', 'MySQL in a Nutshell', 'MySQL', 'MySQL in a Nutshell covers all MySQL funct
administration.', 39.95);

INSERT INTO Inventory (ProductCode, QuantityInStock)
VALUES ('mysqlian', 52);
```

In order to add a product, we need to know the following information:

- Product Code
- Title
- Category
- Description
- Price
- Quantity

These bits of information will become parameters for our stored procedure.

```
OBSERVE:

CREATE PROCEDURE CreateProduct (
  ProductCode varchar(20),
  Title varchar(50),
  Category varchar(30),
  Description text,
  Price decimal (9,2),
  Quantity int
)
BEGIN
END;
//
```

In the previous listing we created a skeleton for our stored procedure. We have each of our **parameters**, followed by a **data type** that matches our table definition. Now let's complete this procedure by filling in our two queries.

```
Type the following at the MySQL prompt:

mysql> CREATE PROCEDURE CreateProduct (
  ProductCode varchar(20),
  Title varchar(50),
  Category varchar(30),
  Description text,
  Price decimal (9,2),
  Quantity int
)
BEGIN
  INSERT INTO Products (ProductCode, Title, Category, Description, Price)
  VALUES (ProductCode, Title, Category, Description, Price);

  INSERT INTO Inventory (ProductCode, QuantityInStock)
  VALUES (ProductCode, Quantity);
END;
//
```

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

```
OBSERVE:

mysql> CREATE PROCEDURE CreateProduct (
-> ProductCode varchar(20),
```

```

-> Title varchar(50),
-> Category varchar(30),
-> Description text,
-> Price decimal (9,2),
-> Quantity int
-> )
-> BEGIN
-> INSERT INTO Products (ProductCode, Title, Category, Description, Price)
-> VALUES (ProductCode, Title, Category, Description, Price);
->
-> INSERT INTO Inventory (ProductCode, QuantityInStock)
-> VALUES (ProductCode, Quantity);
-> END;
-> //
Query OK, 0 rows affected (0.00 sec)

mysql>

```

Calling this procedure is similar to calling the last one. Try it without any parameters:

Type the following at the MySQL prompt:

```

mysql> CALL CreateProduct ();
//

```

This procedure has six parameters, and requires all six. MySQL will return a message:

OBSERVE:

```

mysql> CALL CreateProduct ();
-> //
ERROR 1318 (42000): Incorrect number of arguments for PROCEDURE certjosh.CreateProduct; expected 6
mysql>

```

Note

We just created the procedure, so we know the names and data types of the parameters for our procedure. If you need to check the definition of a procedure, you can do so by using:
 show create procedure *procedure name*
 statement.

Let's call the procedure with parameters this time. The example below will only have one parameter per line, but we're only doing to facilitate our discussion.

Type the following at the MySQL prompt:

```

mysql> CALL CreateProduct (
  "mysqlian",
  "MySQL in a Nutshell",
  "MySQL",
  "MySQL in a Nutshell covers all MySQL functions, as well as MySQL administration.",
  39.95,
  52
);
//

```

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

OBSERVE:

```

mysql> CALL CreateProduct (
-> "mysqlian",
-> "MySQL in a Nutshell",
-> "MySQL",
-> "MySQL in a Nutshell covers all MySQL functions, as well as MySQL administration.",
-> 39.95,
-> 52
-> );
-> //
Query OK, 1 row affected (0.00 sec)

mysql>

```

But wait! Didn't the stored procedure create *two* rows - one in Product, and one in Inventory? Check by running the previous CurrentProductInventory stored procedure.

Type the following at the MySQL prompt:

```
mysql> CALL CurrentProductInventory ();  
//
```

Your results should be similar to this:

OBSERVE:

```
mysql> CALL CurrentProductInventory ();  
-> //  
+-----+-----+-----+-----+  
| ProductCode | Title | Price | Qty |  
+-----+-----+-----+-----+  
| artofsql | The Art of SQL | 44.99 | 52 |  
| databaseid | Database in Depth | 29.95 | 0 |  
| mysqlian | MySQL in a Nutshell | 39.95 | 52 |  
| mysqlspp | MySQL Stored Procedure Programming | 44.99 | 5 |  
| relationaldb | The Relational Database Dictionary | 14.99 | NULL |  
| sqlhks | SQL Hacks | 29.99 | 32 |  
| sqltuning | SQL Tuning | 39.95 | 105 |  
+-----+-----+-----+-----+  
7 rows in set (0.00 sec)  
  
Query OK, 0 rows affected (0.00 sec)  
  
mysql>
```

It seems that two rows were inserted after all. MySQL's responses can sometimes be misleading.

Variables

Most programming languages have some concept of a variable. SQL is not any different. Variables can be very useful in stored procedures, where they can be used to store intermediate results. Variables in MySQL are *not* case sensitive, so a variable named `myVariable` is the same as a variable named `MyVARIABLE`.

Our **Inventory** table currently has two columns - ProductCode and QuantityInStock. Suppose your manager wants to update inventory by title instead of by ProductCode. We could make a stored procedure to make this task easier.

First, let's write some code to capture a ProductCode for a product.

Type the following at the MySQL prompt:

```
mysql> CREATE PROCEDURE GetProductCode(  
    product_title varchar(50)  
)  
BEGIN  
    DECLARE product_code varchar(20);  
  
    SELECT ProductCode into product_code  
    FROM Products  
    WHERE Title = product_title;  
  
    SELECT product_code as ProductCode;  
END;  
//
```

If you typed everything correctly you should see Query OK, 0 rows affected (0.00 sec).

The line with `DECLARE product_code varchar(20)` in our procedure tells MySQL that we are using a variable called `product_code`, whose type is `varchar(20)`. Later we use a `SELECT` statement to view the results.

Let's try the new procedure. We'll lookup the title **SQL Hacks**.

Type the following at the MySQL prompt:

```
mysql> call GetProductCode('SQL Hacks');  
//
```


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

OBSERVE:

```
mysql> call GetProductCode('SQL Hacks');
-> //
+-----+
| ProductCode |
+-----+
| sqlhks      |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>
```

Looking good! The product code is correct, so let's rewrite our procedure to accept one new parameter called **NewQuantityInStock** and update our inventory table.

Let's remove our old procedure before we continue.

Type the following at the MySQL prompt:

```
mysql> DROP PROCEDURE IF EXISTS GetProductCode;
//
```

If everything is OK you'll see the standard message: Query OK, 0 rows affected (0.00 sec).

Let's create our new procedure.

Type the following at the MySQL prompt:

```
mysql> CREATE PROCEDURE UpdateInventory(
    product_title varchar(50),
    NewQuantityInStock int
)
BEGIN
    DECLARE product_code varchar(20);

    SELECT ProductCode into product_code
    FROM Products
    WHERE Title = product_title;

    UPDATE Inventory SET QuantityInStock=NewQuantityInStock
    WHERE ProductCode = product_code;
END;
//
```

There are two new bits: the **NewQuantityInStock** parameter, and the **UPDATE** statement that sets the new quantity in stock.

Before we try the procedure, let's manually reset the QuantityInStock for "sqlhks."

Type the following at the MySQL prompt:

```
mysql> UPDATE Inventory SET QuantityInStock=32
WHERE ProductCode = 'sqlhks';
//
```

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

OBSERVE:

```
mysql> UPDATE Inventory SET QuantityInStock=32
-> WHERE ProductCode = 'sqlhks';
-> //
Query OK, 0 rows affected (0.01 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql>
```

With that done, we can test our procedure. Let's try to set our inventory to **99**.

Type the following at the MySQL prompt:

```
mysql> call UpdateInventory('SQL Hacks',99);  
//
```

If everything went as planned, you'll see:

OBSERVE:

```
mysql> call UpdateInventory('SQL Hacks',99);  
-> //  
Query OK, 1 row affected (0.01 sec)  
  
mysql>
```

Was the inventory updated? Let's check it out.

Type the following at the MySQL prompt:

```
mysql> SELECT * FROM Inventory WHERE ProductCode='sqlhks';  
//
```

Sure enough, it worked perfectly!

OBSERVE:

```
mysql> SELECT * FROM Inventory WHERE ProductCode='sqlhks';  
-> //  
+-----+-----+  
| ProductCode | QuantityInStock |  
+-----+-----+  
| sqlhks      |          99     |  
+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

Looks good! You may be asking yourself, "what happens if there are two books with the same title?" Since the product title isn't the primary key, that could happen. Add a product to your inventory, and run `UpdateInventory` to see what happens.

You've learned a lot in this lesson! You're on your way to creating powerful and reliable database applications. In the next lesson we'll learn about two concepts important to many end users: how to *PIVOT* and *UNPIVOT* your data.

PIVOT and UNPIVOT

DBA 1: Introduction to Database Administration Lesson 10

Welcome back! In the last few lessons we've been looking into the different ways we can write queries and procedures to interact with our data. In this lesson we'll learn about two data manipulation techniques - *pivot* and *unpivot*.

PIVOTing data

Pivoting data is the process of aggregating or moving rows of data into columns of data. Suppose our store keeps track of sales data. The boss only cares about **Units Sold**, and he really wants to know how many units were sold in the **North** and how many were sold in the **South**. Business users and programmers often view data in drastically different ways. Programmers might think in rows and columns of data, such as the following table with sales data by region and date:

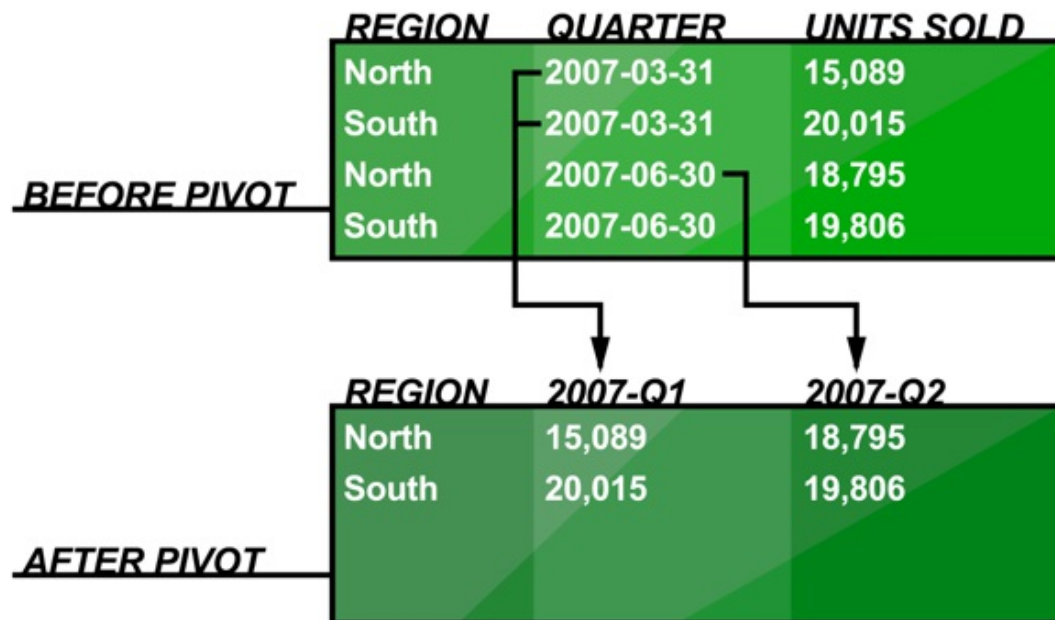
Region	Quarter	Units Sold
North	2007-03-31	15,089
North	2007-06-30	18,795
North	2007-09-30	19,065
North	2007-12-31	19,987
South	2007-03-31	20,015
South	2007-06-30	19,806
South	2007-09-30	21,053
South	2007-12-31	23,068

Showing the data in this way is useful, especially to programmers, however, many business users may want to compare the units sold by region and by date. It's difficult to look at the previous table and compare second quarter 2007 sales for the North and South regions.

Business users would probably prefer to see the data this way:

Region	Units Sold 2007-Q1	Units Sold 2007-Q2	Units Sold 2007-Q3	Units Sold 2007-Q4
North	15,089	18,795	19,065	19,987
South	20,015	19,806	21,053	23,068

This representation shows exactly the same data, but now it's easy for business users to see that for 2007-Q2 the South outsold the North by 1,011 units. Graphically, a pivot may look like this:



Major databases such as Oracle and SQL Server now have `PIVOT` keywords added to their SQL dialect. And while MySQL doesn't have that keyword currently, you can still pivot your data without much trouble.

Let's start by adding a simple table for our demonstrations. We'll use the same data and same basic structure as the first table.

Type the following at the MySQL prompt:

```
mysql> CREATE TABLE SalesAnalysis
(
  Region varchar(10) NOT NULL,
  Quarter date NOT NULL,
  UnitsSold integer NOT NULL
) ENGINE=INNODB;
```

If you typed everything correctly you'll see the familiar Query OK, 0 rows affected (0.01 sec) message.

Once you have the table created, populate it with our sample data.

Type the following at the MySQL prompt:

```
mysql> INSERT INTO SalesAnalysis values ('NORTH', '2007-03-31',15089);
INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-06-30',18795);
INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-09-30',19065);
INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-12-31',19987);

INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-03-31',20015);
INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-06-30',19806);
INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-09-30',21053);
INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-12-31',23068);
```

Make sure you entered everything correctly:

Type the following at the MySQL prompt:

```
mysql> SELECT * FROM SalesAnalysis;
```

OBSERVE:

```
mysql> SELECT * FROM SalesAnalysis;
+-----+-----+-----+
| Region | Quarter      | UnitsSold |
+-----+-----+-----+
| NORTH  | 2007-03-31   | 15089     |
| NORTH  | 2007-06-30   | 18795     |
| NORTH  | 2007-09-30   | 19065     |
| NORTH  | 2007-12-31   | 19987     |
| SOUTH  | 2007-03-31   | 20015     |
| SOUTH  | 2007-06-30   | 19806     |
| SOUTH  | 2007-09-30   | 21053     |
| SOUTH  | 2007-12-31   | 23068     |
+-----+-----+-----+
8 rows in set (0.00 sec)

mysql>
```

Now that we have some sample data, how do we go about pivoting the rows into columns? If you recall from the previous lessons, we've already learned how to use the CASE statement and aggregates. We'll use both of those features to come up with our PIVOT.

The column we want to pivot is **UnitsSold**, and we want to pivot that column by the **Quarter** column. We'll add four new columns called **2007-Q1**, **2007-Q2**, **2007-Q3**, and **2007-Q4**, and use a CASE statement to allocate UnitsSold to each of those new columns. Let's try it!

Type the following at the MySQL prompt:

```
mysql> SELECT Region,
CASE WHEN Quarter='2007-03-31' THEN UnitsSold END AS '2007-Q1',
CASE WHEN Quarter='2007-06-30' THEN UnitsSold END AS '2007-Q2',
CASE WHEN Quarter='2007-09-30' THEN UnitsSold END AS '2007-Q3',
CASE WHEN Quarter='2007-12-31' THEN UnitsSold END AS '2007-Q4'
FROM SalesAnalysis;
```

In the previous listing, the UnitsSold in the **first quarter** are entered into the new **'2007-Q1'** column, the UnitsSold in the **second quarter** are put into the new **'2007-06-30'** column, and so on. But the results are not exactly correct - they still contain as many rows as the original table, with a bunch of NULL values in between them.

OBSERVE:

```
mysql> SELECT Region,
-> CASE WHEN Quarter='2007-03-31' THEN UnitsSold END AS '2007-Q1',
-> CASE WHEN Quarter='2007-06-30' THEN UnitsSold END AS '2007-Q2',
-> CASE WHEN Quarter='2007-09-30' THEN UnitsSold END AS '2007-Q3',
-> CASE WHEN Quarter='2007-12-31' THEN UnitsSold END AS '2007-Q4'
-> FROM SalesAnalysis;
+-----+-----+-----+-----+
| Region | 2007-Q1 | 2007-Q2 | 2007-Q3 | 2007-Q4 |
+-----+-----+-----+-----+
| NORTH | 15089 | NULL | NULL | NULL |
| NORTH | NULL | 18795 | NULL | NULL |
| NORTH | NULL | NULL | 19065 | NULL |
| NORTH | NULL | NULL | NULL | 19987 |
| SOUTH | 20015 | NULL | NULL | NULL |
| SOUTH | NULL | 19806 | NULL | NULL |
| SOUTH | NULL | NULL | 21053 | NULL |
| SOUTH | NULL | NULL | NULL | 23068 |
+-----+-----+-----+-----+
8 rows in set (0.00 sec)

mysql>
```

How are we going to collapse those extra rows? We'll use an *aggregate*! In this example we can use `SUM` to add up our new columns, grouped by the region. For good practice, we should also add an `ELSE` clause to the `CASE` statement. Instead of using `NULL` we'll use zero.

Let's try our updated pivot!

Type the following at the MySQL prompt:

```
mysql> SELECT Region,
SUM(CASE WHEN Quarter='2007-03-31' THEN UnitsSold ELSE 0 END) AS '2007-Q1',
SUM(CASE WHEN Quarter='2007-06-30' THEN UnitsSold ELSE 0 END) AS '2007-Q2',
SUM(CASE WHEN Quarter='2007-09-30' THEN UnitsSold ELSE 0 END) AS '2007-Q3',
SUM(CASE WHEN Quarter='2007-12-31' THEN UnitsSold ELSE 0 END) AS '2007-Q4'
FROM SalesAnalysis
GROUP BY Region;
```

This one looks much better, and returns the correct results.

OBSERVE:

```
mysql> SELECT Region,
-> SUM(CASE WHEN Quarter='2007-03-31' THEN UnitsSold ELSE 0 END) AS '2007-Q1',
-> SUM(CASE WHEN Quarter='2007-06-30' THEN UnitsSold ELSE 0 END) AS '2007-Q2',
-> SUM(CASE WHEN Quarter='2007-09-30' THEN UnitsSold ELSE 0 END) AS '2007-Q3',
-> SUM(CASE WHEN Quarter='2007-12-31' THEN UnitsSold ELSE 0 END) AS '2007-Q4'
-> FROM SalesAnalysis
-> GROUP BY Region;
+-----+-----+-----+-----+
| Region | 2007-Q1 | 2007-Q2 | 2007-Q3 | 2007-Q4 |
+-----+-----+-----+-----+
| NORTH | 15089 | 18795 | 19065 | 19987 |
| SOUTH | 20015 | 19806 | 21053 | 23068 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Our business users will be very happy.

UNPIVOTing Data

As you just saw, pivoting data changes rows into columns. *Unpivoting* data changes columns into rows. This is very useful when your business users give you a set of data and ask you to import it into a traditional database format.

Unpivoting looks something like this:

	REGION	2007-Q1	2007-Q2
<i>BEFORE UNPIVOT</i>	North	15,089	19,065
	South	18,795	21,053
	REGION	QUARTER	UNITS SOLD
<i>AFTER UNPIVOT</i>	North	2007-03-31	15,089
	South	2007-03-31	18,795
	North	2007-06-30	19,065

Note

If you pivot data, you cannot necessarily unpivot the result to get back to the original data. Remember that when you pivot data, you aggregate rows into a single column. It may not be possible to deaggregate that column back to several rows.

In the pivot example from earlier in the lesson, we worked with sales data by region and quarter. Suppose this data was imported from a system directly in that format. What would that look like? Let's save our previous query as a view in order to observe this condition.

Type the following at the MySQL prompt:

```
mysql> CREATE VIEW SalesDataPivot AS
SELECT Region,
SUM(CASE WHEN Quarter='2007-03-31' THEN UnitsSold ELSE 0 END) AS '2007-Q1',
SUM(CASE WHEN Quarter='2007-06-30' THEN UnitsSold ELSE 0 END) AS '2007-Q2',
SUM(CASE WHEN Quarter='2007-09-30' THEN UnitsSold ELSE 0 END) AS '2007-Q3',
SUM(CASE WHEN Quarter='2007-12-31' THEN UnitsSold ELSE 0 END) AS '2007-Q4'
FROM SalesAnalysis
GROUP BY Region;
```

If you typed it in correctly you'll see the standard message `Query OK, 0 rows affected (0.00 sec)`. Run a quick query using this new view to make sure you see the same results.

Now that we have our view in place, we can construct our query to unpivot the data. Since MySQL doesn't have an UNPIVOT operator, we'll have to do it the old-fashioned way by using a few queries and UNION statements. We'll write four queries - one for each of the columns to be unpivoted. Then we'll add a new column called **Quarter** to each query.

First let's try to unpivot the first quarter of data - '2007-Q1'.

Type the following at the MySQL prompt:

```
mysql> SELECT Region,
'2007-03-31' AS Quarter,
`2007-Q1` AS UnitsSold
FROM SalesDataPivot;
```

In this listing we added a new column called **Quarter** and renamed the column ``2007-Q1`` "UnitsSold".

WARNING

Be sure to type the correct character for the column `2007-Q1`. MySQL requires column names to be backwards apostrophe (') quoted because the column name starts with a number and contains a dash. Don't confuse the backwards apostrophe (') character with the plain apostrophe (') character. The backwards apostrophe (') is usually located close to the escape key on your keyboard.

If you typed everything correctly, you'll see something like this:

OBSERVE:

```
mysql> SELECT Region,
-> '2007-03-31' AS Quarter,
```

```

-> `2007-Q1` AS UnitsSold
-> FROM SalesDataPivot;
+-----+-----+-----+
| Region | Quarter | UnitsSold |
+-----+-----+-----+
| NORTH  | 2007-03-31 | 15089 |
| SOUTH  | 2007-03-31 | 20015 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>

```

Looks great! Now we can combine this with three more queries, and a **UNION** statement to collect all of the results.

Type the following at the MySQL prompt:

```

mysql> SELECT Region, '2007-03-31' AS Quarter, `2007-Q1` AS UnitsSold
FROM SalesDataPivot
UNION
SELECT Region, '2007-06-30' AS Quarter, `2007-Q2` AS UnitsSold
FROM SalesDataPivot
UNION
SELECT Region, '2007-09-30' AS Quarter, `2007-Q3` AS UnitsSold
FROM SalesDataPivot
UNION
SELECT Region, '2007-12-31' AS Quarter, `2007-Q4` AS UnitsSold
FROM SalesDataPivot;

```

As long as you typed everything correctly, you'll see exactly the same data that is stored in the SalesAnalysis table!

OBSERVE:

```

mysql> SELECT Region, '2007-03-31' AS Quarter, `2007-Q1` AS UnitsSold
-> FROM SalesDataPivot
-> UNION
-> SELECT Region, '2007-06-30' AS Quarter, `2007-Q2` AS UnitsSold
-> FROM SalesDataPivot
-> UNION
-> SELECT Region, '2007-09-30' AS Quarter, `2007-Q3` AS UnitsSold
-> FROM SalesDataPivot
-> UNION
-> SELECT Region, '2007-12-31' AS Quarter, `2007-Q4` AS UnitsSold
-> FROM SalesDataPivot;
+-----+-----+-----+
| Region | Quarter | UnitsSold |
+-----+-----+-----+
| NORTH  | 2007-03-31 | 15089 |
| SOUTH  | 2007-03-31 | 20015 |
| NORTH  | 2007-06-30 | 18795 |
| SOUTH  | 2007-06-30 | 19806 |
| NORTH  | 2007-09-30 | 19065 |
| SOUTH  | 2007-09-30 | 21053 |
| NORTH  | 2007-12-31 | 19987 |
| SOUTH  | 2007-12-31 | 23068 |
+-----+-----+-----+
8 rows in set (0.00 sec)

mysql>

```

We did it! We successfully pivoted the data, then unpivoted it to its original form.

You've learned yet another way to query the data in your database. MySQL offers additional ways to facilitate your database applications - user defined types, functions, and extensions are all ways to expand the ways you store and interact with your data. See you in the next lesson!

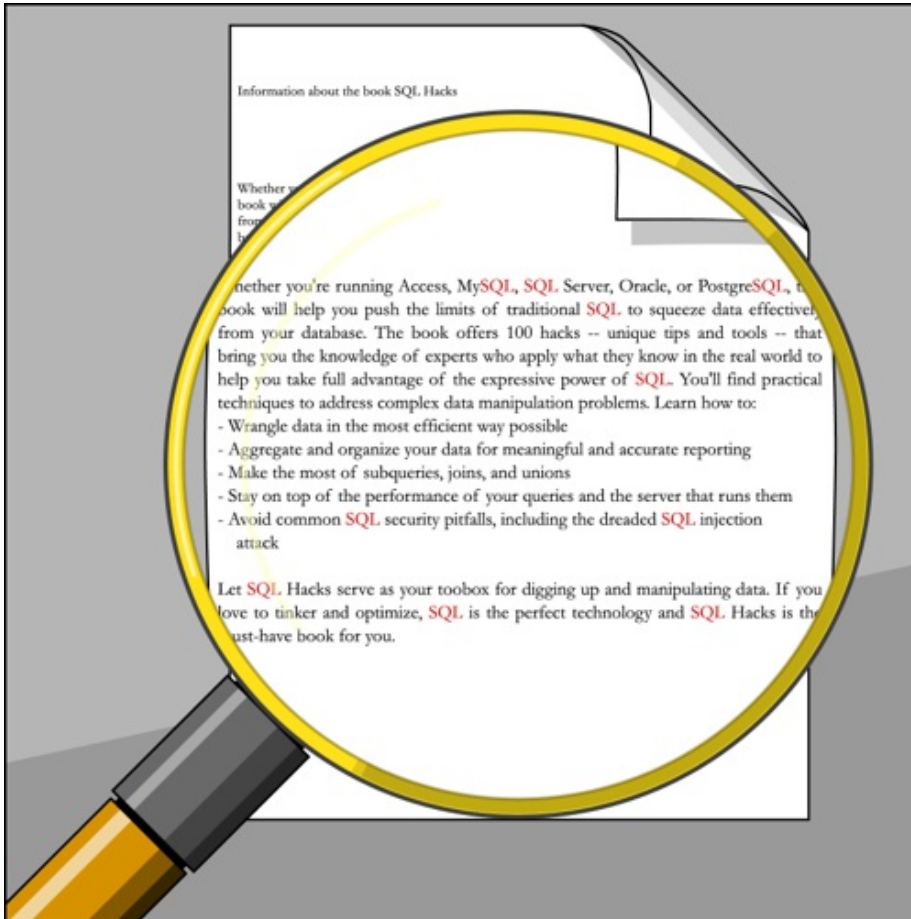
Full Text

DBA 1: Introduction to Database Administration Lesson 11

We've covered a lot of information in the course so far, and learned many ways to store and interact with your data. In this lesson we'll cover one more tool in the SQL tool box: full-text indexes.

Creating Full-Text Indexes

You may want to create a full-text index to enable quick searches through large amounts of text. Staff working in a bookstore are usually knowledgeable about many authors and topics, but they can benefit from the use of an efficient search tool. We can help them by providing a text search of all information associated with each book including the title, author, book description, and category.



Instead of creating a complex search on all of the database fields, a full-text index lets you query all included fields at once.

Note

Full-text indexes are not a replacement for normal indexes and queries. Like the name suggests, 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. Before we can use this index on our *Products* table, we'll have to make sure the table type is *MyISAM*.

Type the following at the MySQL prompt:

```
mysql> ALTER TABLE Products ENGINE=MyISAM;
```

If you typed everything correctly you'll see a message similar to the following. (It's OK if you get a warning.)

OBSERVE:

```
mysql> ALTER TABLE Products ENGINE=MyISAM;
Query OK, 7 rows affected (0.01 sec)
Records: 7 Duplicates: 0 Warnings: 0

mysql>
```


Before we start experimenting with full-text indexes, make sure you've included some products with long descriptions. Let's start by cleaning up the tables a bit. First, delete existing rows from the Products and Inventory tables.

Type the following at the MySQL prompt:

```
mysql> DELETE FROM Products;
DELETE FROM Inventory;
```

You should see something similar to this:

OBSERVE:

```
mysql> DELETE FROM Products;
Query OK, 7 rows affected (0.00 sec)

mysql> DELETE FROM Inventory;
Query OK, 7 rows affected (0.01 sec)

mysql>
```

To avoid a lot of typing, we'll download an SQL file that contains new products with long descriptions. Open a second Unix Terminal to download a special SQL file for this lesson.

We'll grab the file from [O'Reilly's servers](http://courses.oreillyschool.com/dba1/downloads/products-10.sql) using the **curl** command.

Type the following at the Unix prompt in the second Unix Terminal:

```
cold:~$ curl http://courses.oreillyschool.com/dba1/downloads/products-10.sql > products-10.sql
```

Once you have downloaded the file, use the **mysql** command to import the products.

Type the following at the Unix prompt in the second Unix Terminal:

```
cold:~$ mysql -h sql -p -u username username < products-10.sql
```

If everything went well, you'll see no results:

OBSERVE:

```
cold:~$ mysql -h sql -p -u username username < products-10.sql
Enter password:
cold:~$
```

Now that we have some data, let's create our full-text index. We need to tell the database server which columns we are interested in searching. We could search through all char, varchar and text columns, however our users will primarily search by title and description.

Let's add **FULLTEXT** to the **Title** and **Description** columns. Be sure to switch back to **Unix** mode.

Type the following at the MySQL prompt:

```
mysql> ALTER TABLE Products ADD FULLTEXT (Title, Description);
```

You should see a message similar to this:

OBSERVE:

```
mysql> ALTER TABLE Products ADD FULLTEXT (Title, Description);
Query OK, 7 rows affected (0.01 sec)
Records: 7 Duplicates: 0 Warnings: 0

mysql>
```

That's all there is to it! Now you're ready to write some queries.

Querying full-text Indexes

The syntax to use a full-text index in a query is straightforward. The magic is in the **WHERE** clause.

OBSERVE:

```
WHERE MATCH (column 1, column 2...column n)
AGAINST ('comma separated list of search criteria');
```

Since the books in the Products table are all on the topic of databases, try searching for "relational database."

Type the following at the MySQL prompt:

```
mysql> SELECT ProductCode, Title, Price FROM Products
WHERE MATCH (Title,Description)
AGAINST ('relational databases');
```

Unless you have additional database related books in your table, you should see three rows of data. Take a peek at the descriptions for these books - all have some mention of the phrase "relational database."

OBSERVE:

```
mysql> SELECT ProductCode, Title, Price FROM Products
-> WHERE MATCH(Title,Description)
-> AGAINST('relational databases');
+-----+-----+-----+
| ProductCode | Title                                     | Price |
+-----+-----+-----+
| databaseid  | Database in Depth                       | 29.95 |
| relationaldb | The Relational Database Dictionary     | 14.99 |
| artofsql    | The Art of SQL                         | 44.99 |
+-----+-----+-----+
3 rows in set (0.02 sec)

mysql>
```

Let's compare the query we just executed to the other way of searching for this information - using the LIKE operator. You probably remember that the % percent symbol matches anything when using the LIKE operator.

Type the following at the MySQL prompt:

```
mysql> select ProductCode, Title, Price FROM Products
WHERE title like '%relational%databases%' OR description like '%relational%databases%';
```

Run this query. You might be surprised by the results:

OBSERVE:

```
mysql> select ProductCode, Title, Price FROM Products
-> WHERE title like '%relational%databases%' OR description like '%relational%databases%';
+-----+-----+-----+
| ProductCode | Title                                     | Price |
+-----+-----+-----+
| relationaldb | The Relational Database Dictionary     | 14.99 |
| databaseid  | Database in Depth                       | 29.95 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Only two rows were returned! If you suspect that the full-text index is doing some extra work, your suspicions would be correct. Full-text indexes take into account the language being indexed in order to return relevant results. By default a natural language search is performed, meaning similar words such as "database" and "databases" would be considered a match. While that type of match is possible when using LIKE, it leads to some tricky SQL code!

The full-text index has a few optimizations to help performance:

- Words less than five characters in length are ignored. (This minimum length can be changed.)
- Stop words such as "the" and even "considering" are ignored by default. (The list of stop words can be overridden.)

Check out [MySQL's web site](#) for a list of default stop words.

Note

Since our Products table is small, using full-text index probably doesn't affect speed and efficiency much. However, on a large table with a properly planned full-text index, the impact on performance can be significant.

Try expanding the criteria - add "oracle" to the search.

Type the following at the MySQL prompt:

```
mysql> SELECT ProductCode, Title, Price FROM Products
WHERE MATCH(Title,Description)
AGAINST('relational databases, oracle');
```

This time five books matched the criteria.

OBSERVE:

```
mysql> SELECT ProductCode, Title, Price FROM Products
-> WHERE MATCH(Title,Description)
-> AGAINST('relational databases, oracle');
+-----+-----+-----+
| ProductCode | Title | Price |
+-----+-----+-----+
| databaseid | Database in Depth | 29.95 |
| relationaldb | The Relational Database Dictionary | 14.99 |
| mysqlspp | MySQL Stored Procedure Programming | 44.99 |
| sqlhks | SQL Hacks | 29.99 |
| artofsql | The Art of SQL | 44.99 |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

The next question you might ask is **how can I rank my matches in order of relevance?**

The answer: include MATCH...AGAINST in the selected columns:

Type the following at the MySQL prompt:

```
mysql> SELECT ProductCode, Title, Price,
MATCH(Title,Description) AGAINST('relational databases') as Relevance
FROM Products
WHERE MATCH(Title,Description) AGAINST('relational databases');
```

Note

It may seem like MySQL would execute the full-text search twice in the previous query. As long as both MATCH...AGAINST sections are exactly the same, only one query will execute.

Now you can see how well your search criteria matches the full-text index. For MySQL, relevance numbers are always floating point numbers, greater than or equal to zero. A relevance of zero indicates no match.

OBSERVE:

```
mysql> SELECT ProductCode, Title, Price,
-> MATCH(Title,Description) AGAINST('relational databases') as Relevance
-> FROM Products
-> WHERE MATCH(Title,Description) AGAINST('relational databases');
+-----+-----+-----+-----+
| ProductCode | Title | Price | Relevance |
+-----+-----+-----+-----+
| databaseid | Database in Depth | 29.95 | 1.313608288765 |
| relationaldb | The Relational Database Dictionary | 14.99 | 0.81241208314896 |
| artofsql | The Art of SQL | 44.99 | 0.10763692110777 |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

You've just learned how you can use full-text indexes to provide a fast and powerful way to search your data. In the next lesson you'll learn how to peek into the INFORMATION_SCHEMA tables MySQL maintains to learn more about your tables and columns. See you then!

INFORMATION_SCHEMA

DBA 1: Introduction to Database Administration Lesson 12

Welcome back! So far we've learned how to create and populate tables, use views, and create stored procedures. Tables are made up of rows and columns. A view is a stored version of a query, and a stored procedure is a sequence of steps kept in the database. All of this information is *metadata* - extra information that describes our database.

INFORMATION_SCHEMA

A description of our database could be written in English like this: "tables and procedures used to keep track of book inventory in our store." It also could be composed in the form of a list:

- Inventory
- Products
- CreateProduct
- ...

When you think about it, this data could be organized in rows and columns, just like a database table itself! This is a useful way to access database information since we can use our standard query tools to reveal all aspects of our database.

MySQL and other databases (at least those that claim to support some SQL:2003 features) provide a window into database metadata through the **INFORMATION_SCHEMA**.

A schema is a collection of tables, fields, and the relationships between fields and tables. In MySQL, an **INFORMATION_SCHEMA** is a pseudo-database. It doesn't physically exist, however you are able to connect to it. You can query any table in the database, however you cannot update, alter, or delete anything.

Tables

Suppose your boss stops by your desk one afternoon. He heard about the fancy full-text indexes you used in the previous lesson, and he wants to know which tables use the MyISAM engine and which ones use the InnoDB engine. **INFORMATION_SCHEMA** will enable you to answer this question quickly.

Let's take a peek at our tables by querying the **TABLES** table in **INFORMATION_SCHEMA**. In the following list be sure to replace **username** with your own username.

Type the following at the MySQL prompt:

```
mysql> SELECT table_name, table_type, engine
FROM INFORMATION_SCHEMA.TABLES
WHERE table_schema='username';
```

Depending on how many tables you've created in your database, you should see something similar to the following (some of the rows have been omitted for clarity):

OBSERVE:

```
mysql> SELECT table_name, table_type, engine
-> FROM INFORMATION_SCHEMA.TABLES
-> WHERE table_schema='username';
+-----+-----+-----+
| table_name | table_type | engine |
+-----+-----+-----+
| Customers | BASE TABLE | InnoDB |
| Inventory | BASE TABLE | InnoDB |
| OrderLine | BASE TABLE | InnoDB |
| Orders | BASE TABLE | InnoDB |
| ProductInventory | VIEW | NULL |
| Products | BASE TABLE | MyISAM |
| SalesAnalysis | BASE TABLE | MyISAM |
| SalesDataPivot | VIEW | NULL |
...
| test | BASE TABLE | MyISAM |
+-----+-----+-----+
31 rows in set (1.29 sec)

mysql>
```

With one short query we were able to get the exact information we needed. Now, suppose your boss only wants the names of the tables that *do not* support full-text indexes. We can easily add to the **WHERE** clause to our previous query to answer that

question:

Type the following at the MySQL prompt:

```
mysql> SELECT table_name, table_type, engine
FROM INFORMATION_SCHEMA.TABLES
WHERE table_schema='certjosh' AND engine='InnoDB';
```

It looks like we have four tables that do not support full-text indexing:

OBSERVE:

```
mysql> SELECT table_name, table_type, engine
-> FROM INFORMATION_SCHEMA.TABLES
-> WHERE table_schema='certjosh' AND engine='InnoDB';
+-----+-----+-----+
| table_name | table_type | engine |
+-----+-----+-----+
| Customers  | BASE TABLE | InnoDB |
| Inventory  | BASE TABLE | InnoDB |
| OrderLine  | BASE TABLE | InnoDB |
| Orders     | BASE TABLE | InnoDB |
+-----+-----+-----+
4 rows in set (1.00 sec)
```

Columns

Well, your boss was very impressed with how quickly you were able to provide the information he requested, so he's come directly to you with his next query request. It seems he is worried that certain columns in the Products table are not large enough to hold all of the data for a new book - **Steal This Computer Book 4.0: What They Won't Tell You About the Internet**. He may be right - that title is very long!

To find this information fast, we'll peek into the **COLUMNS** table of **INFORMATION_SCHEMA**.

Type the following at the MySQL prompt:

```
mysql> SELECT column_name, data_type, character_maximum_length
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name='Products' AND table_schema='username';
```

If you typed everything correctly, you'll see results similar to this:

OBSERVE:

```
mysql> SELECT column_name, data_type, character_maximum_length
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE table_name='Products' AND table_schema='username';
+-----+-----+-----+
| column_name | data_type | character_maximum_length |
+-----+-----+-----+
| ProductCode | char      | 20                        |
| Title       | varchar   | 50                        |
| Category    | varchar   | 30                        |
| Description | text      | 65535                     |
| Price       | decimal   | NULL                      |
+-----+-----+-----+
5 rows in set (1.36 sec)

mysql>
```

A quick visual inspection shows that the **Title** column may not be long enough. Let's refine our query to return all columns that aren't long enough to contain the title. To do this we can use the **length** function - it returns an integer count of a string length.

Type the following at the MySQL prompt:

```
mysql> SELECT column_name, data_type, character_maximum_length
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name='Products'
AND character_maximum_length < length('Steal This Computer Book 4.0: What They Won't Tell You
```

Note You'll need to be sure to include a backslash (\) in the title, since the title itself contains an apostrophe ('). The backslash "escapes" the quote so MySQL won't think it is the end of the title.

Try it out - the results should look something like this:

```
OBSERVE:

mysql> SELECT column_name, data_type, character_maximum_length
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE table_name='Products'
-> AND character_maximum_length < length('Steal This Computer Book 4.0: What They Won\'t Te

+-----+-----+-----+
| column_name | data_type | character_maximum_length |
+-----+-----+-----+
| ProductCode | char      | 20                       |
| Title       | varchar   | 50                       |
| Category    | varchar   | 30                       |
+-----+-----+-----+
3 rows in set (0.99 sec)

mysql>
```

The title column was way too short. The ProductCode and Category may also need to be increased. You give this information to your boss who is extremely happy with your work!

VIEWS

Later in the day your boss stops by yet again. This time he is concerned about the views being used in the application. He wants to know exactly which views have been created, and which views allow UPDATE, DELETE, and perhaps INSERT queries. Once again you turn to INFORMATION_SCHEMA - this time to the **VIEWS** table.

Type the following at the MySQL prompt:

```
mysql> SELECT table_name, is_updatable, view_definition
FROM INFORMATION_SCHEMA.VIEWS;
```

If you typed everything correctly you'll see a ton of output:

```
OBSERVE:

mysql> SELECT table_name, is_updatable, view_definition
-> FROM INFORMATION_SCHEMA.VIEWS;

+-----+-----+-----+
| table_name      | is_updatable | view_definition
+-----+-----+-----+
| ProductInventory | NO           | /* ALGORITHM=UNDEFINED */ select `P`.`ProductCode` AS `Prod
| SalesDataPivot  | NO           | /* ALGORITHM=UNDEFINED */ select `certjosh`.`SalesAnalysis`
+-----+-----+-----+
2 rows in set (1.02 sec)

mysql>
```

You know you'll have to make this output more readable before you hand it over to your boss. Fortunately, he doesn't need to see the whole view definition, just the first several characters. We can use **substring** to skip the first few characters and limit the description to a total of 50 characters.

Type the following at the MySQL prompt:

```
mysql> SELECT table_name, is_updatable,
substring(view_definition,27,50) AS definition
FROM INFORMATION_SCHEMA.VIEWS;
```

This is much easier on the eyes!

```
OBSERVE:

mysql> SELECT table_name, is_updatable,
-> substring(view_definition,27,50) AS definition
```

```

-> FROM INFORMATION_SCHEMA.VIEWS;
+-----+-----+-----+
| table_name      | is_updatable | definition |
+-----+-----+-----+
| ProductInventory | NO          | select `P`.`ProductCode` AS `ProductCode`, `P`.`Tit |
| SalesDataPivot  | NO          | select `certjosh`.`SalesAnalysis`.`Region` AS `Reg |
+-----+-----+-----+
2 rows in set (0.99 sec)

mysql>

```

You give the results to your boss, who is still very impressed with your work.

ROUTINES

It's nearing 5:00 PM, and your boss comes by your desk for one last piece of information. A lazy developer really wants to rename the "QuantityInStock" column to "Qty" to save typing. Your boss wants to know how many procedures will have to be changed if the column is renamed.

You tell him you'll look into it, and will give him an answer by the end of the day. For this task you'll use the **ROUTINES** table from INFORMATION_SCHEMA. Some of the procedures can be quite long, so let's end our query with \G to request the "long" output from MySQL instead of the "wide" output.

Type the following at the MySQL prompt:

```

mysql> SELECT routine_name, routine_type, routine_definition
FROM INFORMATION_SCHEMA.ROUTINES \G

```

Your results might be slightly different, but they should look similar to this:

OBSERVE:

```

mysql> SELECT routine_name, routine_type, routine_definition FROM INFORMATION_SCHEMA.ROUTINES \
***** 1. row *****
routine_name: CreateProduct
routine_type: PROCEDURE
routine_definition: BEGIN
insert into Products (ProductCode, Title, Category, Description, Price)
values (ProductCode, Title, Category, Description, Price);

insert into Inventory (ProductCode, QuantityInStock)
values (ProductCode, Quantity);
END
***** 2. row *****
routine_name: CurrentProductInventory
routine_type: PROCEDURE
routine_definition: BEGIN
select P.ProductCode, P.Title, P.Price, I.QuantityInStock as Qty
from Products as P
LEFT JOIN Inventory as I on (P.ProductCode = I.ProductCode);
END
2 rows in set (0.00 sec)

mysql>

```

These are exactly the results we want. Our boss is busy, so he doesn't want to spend a lot of time reading through the procedure definitions to see where "QuantityInStock" is being used, so we'll add a calculated column to do that search for him.

Type the following at the MySQL prompt:

```

mysql> SELECT routine_name, routine_type,
CASE WHEN routine_definition LIKE '%QuantityInStock%' THEN 1 ELSE 0 END
AS Uses_QuantityInStock
FROM INFORMATION_SCHEMA.ROUTINES;

```

It looks like we have some work to do if we decide to rename the column.

OBSERVE:

```

mysql> SELECT routine_name, routine_type,
-> CASE WHEN routine_definition LIKE '%QuantityInStock%' THEN 1 ELSE 0 END
-> AS Uses_QuantityInStock

```

```
-> FROM INFORMATION_SCHEMA.ROUTINES ;
+-----+-----+-----+
| routine_name      | routine_type | Uses_QuantityInStock |
+-----+-----+-----+
| CreateProduct     | PROCEDURE    | 1 |
| CurrentProductInventory | PROCEDURE    | 1 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Once again you've given your boss the information he needs. It might be time to ask for a raise!

For additional information on the INFORMATION_SCHEMA tables in MySQL, check out [MySQL's web site](#).

Thats it! You've gotten a good start on your database applications. But you probably don't want to stop now. This course is just the first in a series designed to give you the confidence and knowledge you need to use databases effectively. The next lesson will be a description of your final project. See you there!

Final Project

DBA 1: Introduction to Database Administration Lesson 13

Wow! You've come a long way. We've covered a lot, and now it's time for you to apply your new knowledge to the final project.

Final Project

You've done a good job at ATunes, and learned a lot. In order to continue your learning, and to share tips and experiences with databases, you decide to create a blog just for databases. You'll publish articles written by many different people, and allow visitors to comment on those articles.

You have a friend who will make a web site for you, so the only aspect of the blog you'll have to worry about is the database. You need to create all of the database objects needed to support the blog. To complete the project, you'll need to create the following tables:

1. ArticleAuthors - names, email addresses, some biographical information
2. Articles - title, short description (blurb), article text, date (use a full text index on the blurb)
3. ArticleCategories - categories are like MySQL, Oracle, SQL Server, Queries, Stored Procedures, etc.
4. Commenters - names, email addresses of commenters
5. Comments - Notes left by commenters regarding an article
6. ArticleViews - Logs the date and time an article was viewed

Feel free to add additional columns as you see necessary. Make sure you use appropriate data types.

After you create the tables, populate them all with sample data. Create at least five articles. You don't have to write all of the content yourself - test data is fine.

Note Make sure you created a full text index on the Article blurb.

Next, write a stored procedure named *AddComment* to add a comment on an article, creating an entry in Commenters if it doesn't exist.

Create an view named *ArticleDisplay* that displays author, the article, and category for the article. This view will be used to display the article on the web page.

Write the following queries:

1. Demonstrate the use of the full text index on the Articles table. One query should show successful matches with a ranking, another query should show no matches.
2. Pivot ArticleViews to display the number of times an article was viewed in the last 10, 20 and 30 days. The output should look something like the following:

OBSERVE:			
Article	D10	D20	D30
New Features in MySQL	5	10	7
Views in Oracle	2	4	1

3. Write a query that shows the most popular article - the article that has the most rows in ArticleViews.

Make sure you name your objects carefully - your mentor will be looking for specific names.

Store all of your CREATE, INSERT and SELECT statements in a file called **dba1finalproject.sql**. When you are finished, hand in your SQL file.

GOOD LUCK!