DBA 1: Introduction to Database Administration

Introduction

Welcome to the O'Reilly DBA Series!

In this course, you will learn the basics to create a well-designed database using basic database commands. You will also learn how to manipulate the data stored in these tables and to return meaningful results to help analyze the data stored.

Course Objectives

When you complete this course, you will be able to:

- create, read, update, and delete data using basic SQL commands.
- perform relational joins and transactions on SQL tables.
- aggregate data through functions and conditionals.
- perform subqueries and manipulate views.
- create stored procedures within SQL.
- summarize data using PIVOT and UNPIVOT.
- demonstrate knowledge of data indices and information schemas.
- create a full-fledged blog database.

From beginning to end, you will learn by doing projects in your own Unix and MySQL environments, and then handing them in for instructor feedback. These projects, as well as the final project—developing a complete database and demonstrating administrative tasks—will add to your portfolio and will contribute to certificate completion. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

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* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

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

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet,

reference books, and online help are all fair game.

- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! 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 you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

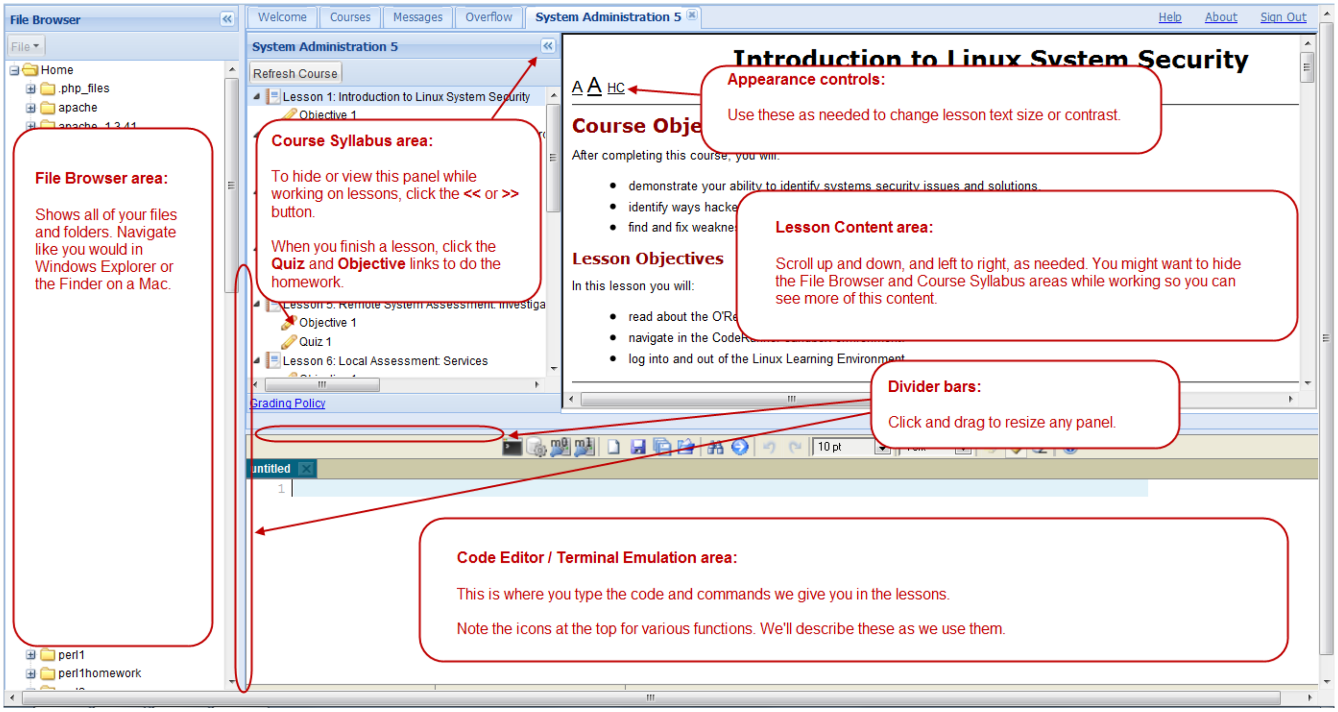
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

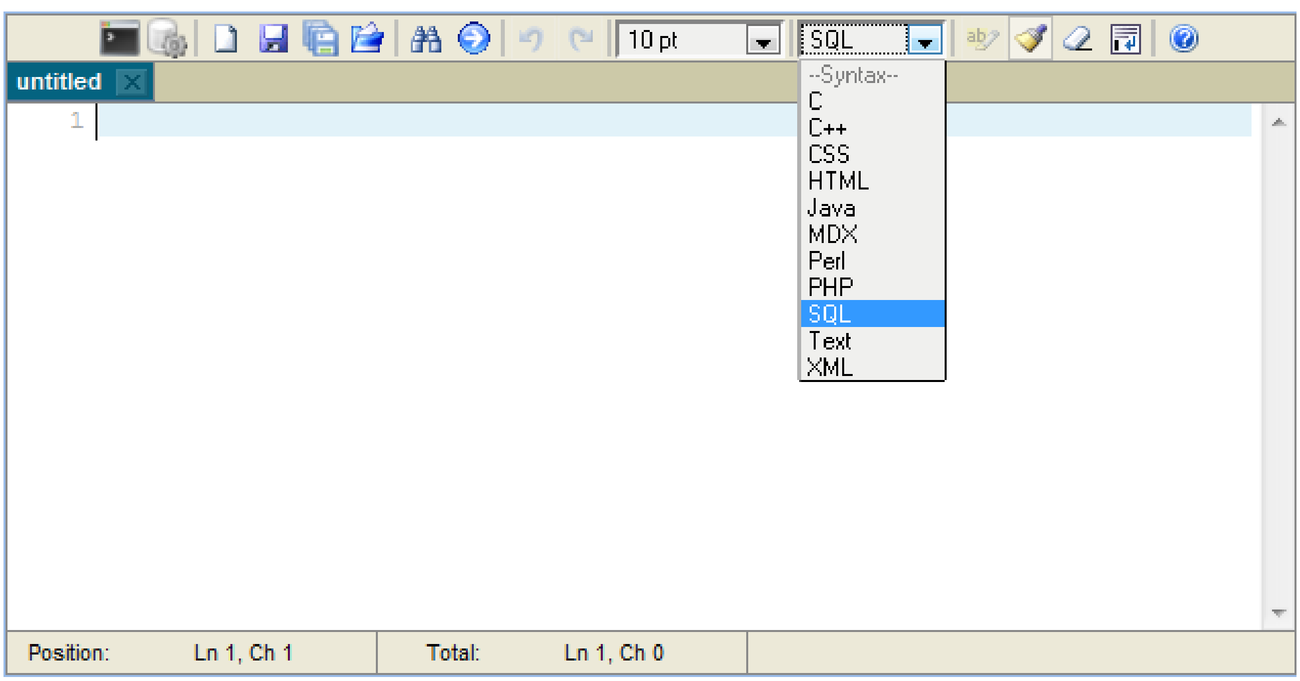
Code Editor Demo

Coursework Demo

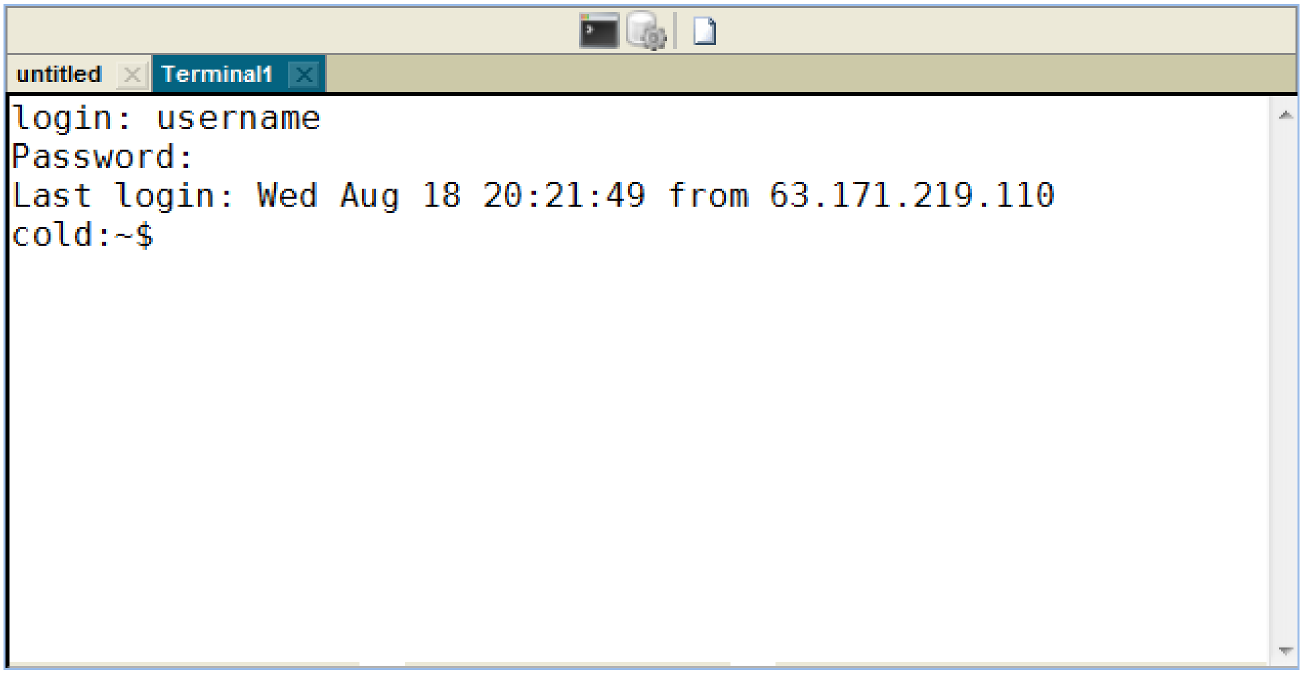
Getting Started

In this course, you will be creating SQL files. SQL files are made up of SQL statements and have an .sql extension.

Click the **New Terminal** button to connect to Unix or click the **Connect to MySQL** button to connect directly to the MySQL server. In this course we'll primarily use 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.



Click 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 both occurrences of *username* with your own username.

Type the following at the UNIX prompt:

```
cold1:~$ mysql -h sql -p -u username username
```

Your screen will look like this:

OBSERVE:

```
cold1:~$ 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: 5.1.69-log Source distribution

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

To connect to a mysql database, we run the program **mysql**. To specify which database server we want to connect to, we type **-h sql**. We want to be prompted to provide a password, so 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:

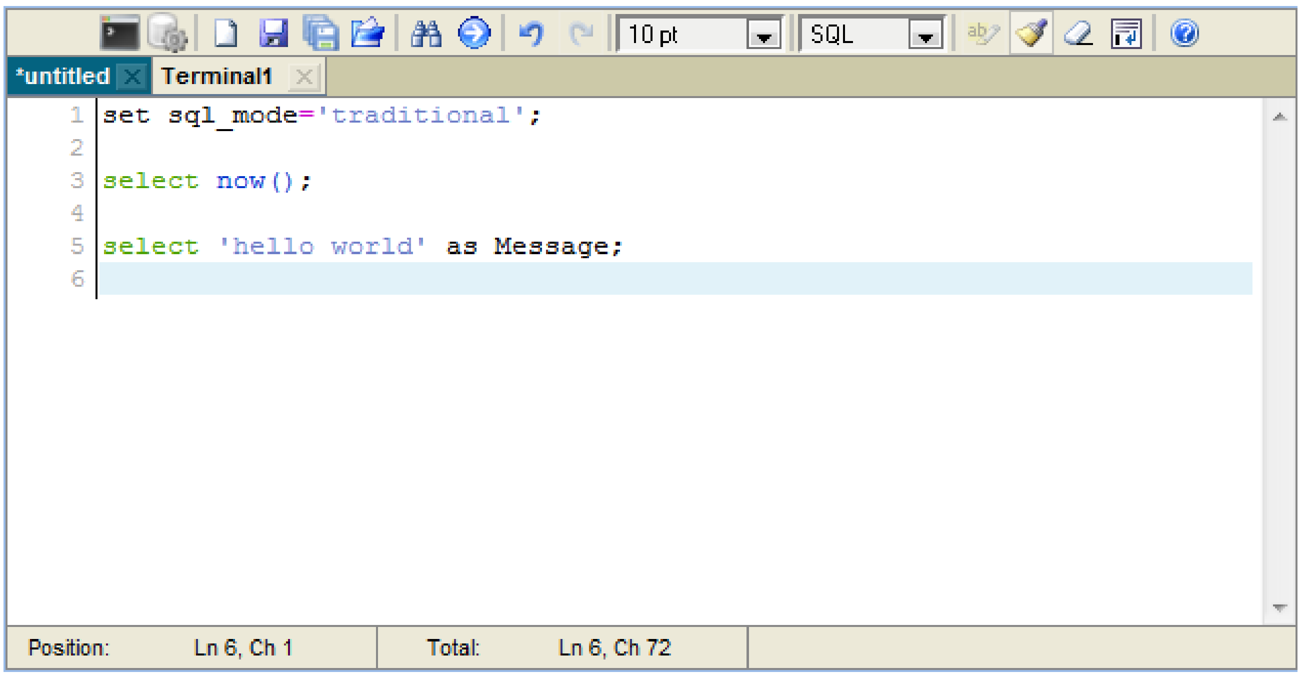
```
mysql> set sql_mode='traditional';
Query OK, 0 rows affected (0.00 sec)

mysql>
```

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

INTERACTIVE SESSION:

```
mysql> exit
Bye
cold1:~$
```

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. Now, click the **New File** button in the menu bar to create a new "untitled" tab.

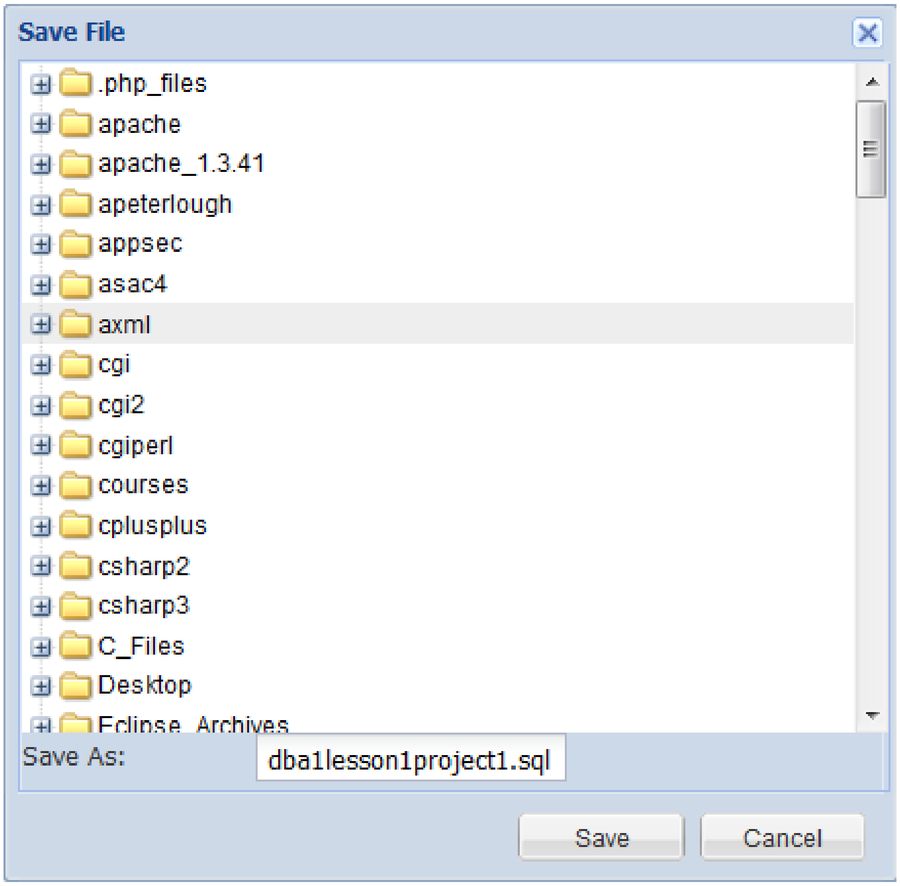


Enter the following code as shown:

Type the following into CodeRunner:

```
set sql_mode='traditional';

select now();

select 'hello world' as Message;
```

To save your file, click the **Save** button. For this example, name the file **dba1lesson1project1.sql**:



Note When naming files and creating tables, please be sure to name them exactly as requested. The .sql extension is important for MySQL files, and in a Unix system, case is significant: 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 the **Load** button and select the file from the **Load File** window, or double-click the file in the File Browser. Note that when you save a file, the word "untitled" is replaced by the file name in the code editor tab. When you open a file, the file opens in a new tab.

Test it in the Unix Terminal using the following command:

Type the following at the UNIX prompt:

```
cold1:~$ mysql -t -h sql -p -u username username < dba1lesson1project1.sql
Enter password:
+---------------------+
| now()               |
+---------------------+
| 2013-10-16 15:56:48 |
+---------------------+
+-------------+
| Message     |
+-------------+
| hello world |
+-------------+
cold1:~$
```

OBSERVE:

```
cold1:~$ mysql -t -h sql -p -u username username < dba1lesson1project1.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 **dba1lesson1project1.sql** with your project filename.

You've finished this lesson! When you finish a lesson, complete the homework objectives and quizzes. To get started on the first objective, click **Objective 1** in the curriculum area in the upper left. For this objective, you'll submit the **dba1lesson1project1.sql** file we created here.

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

Storing Data

Storing Data

Before we get started, make sure you are in the Unix Terminal and then connect to MySQL. Click the **New Terminal** button  and log in. 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* with your actual username, and be sure to type your username twice!

Type the following at the UNIX prompt:

```
cold1:~$ 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**pdating data, and **D**eleting data; we refer to these operations collectively as *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

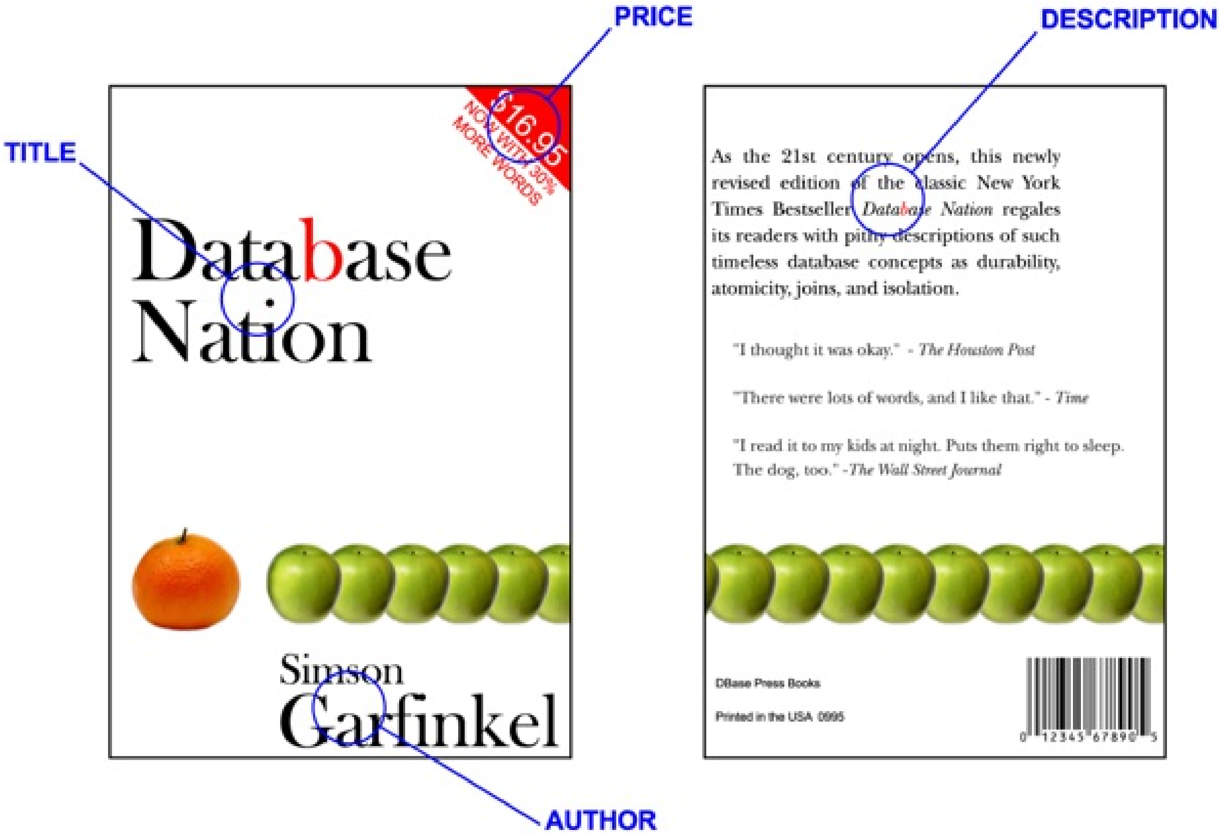
Before you can build the walls of a skyscraper, you must decide which material you will use to build your walls. Database administrators must also choose the material with which to construct their database. Let's work with an example.

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 their 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 are 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 textual. The date published is exactly that: a date. The number of pages is a non-negative integer. The cost is a decimal, with two decimal places. The description, if one exists, is also textual.

As a database administrator (DBA), your job is 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, because 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 classified into three groups: text data types, numeric data types, and date/time data types. Here are some of the data types 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	An approximate decimal point value stored in 4 bytes.	5.15

Numeric	DOUBLE	An approximate decimal point value stored in 8 bytes.	7.25
Numeric	DECIMAL	An exact decimal point value. The preferred data type for storing monetary values such as prices or salaries.	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 a 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 "" or 0. 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 to interact quickly 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, the book might come in hardcover and paperback editions, and there might someday be another book with the same title. If our book store stocks both editions, or the other book with the same title, the title "Database Nation" alone won't be enough for us to distinguish between the different books. Thus we won't be able to use the title as our primary key.

Among other aspects of a book that we might store in a database, the price, the author, the description, and the category are all potentially not unique.

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 CodeRunner is 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;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Let's take a closer look:

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;
```

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 (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 use the **INNODB** engine (**ENGINE=INNODB**) for most of the tables we create. This engine 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'll create additional tables, and see how all of the tables relate to each other. See you there!

Relationships Between Tables

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 customers 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! Also, we designed the Products table to contain products, and a customer isn't a product!

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 provides an **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;
 Query OK, 0 rows affected (0.01 sec)



mysql>
```

Let's discuss some important parts of this command:

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;
```

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

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

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.

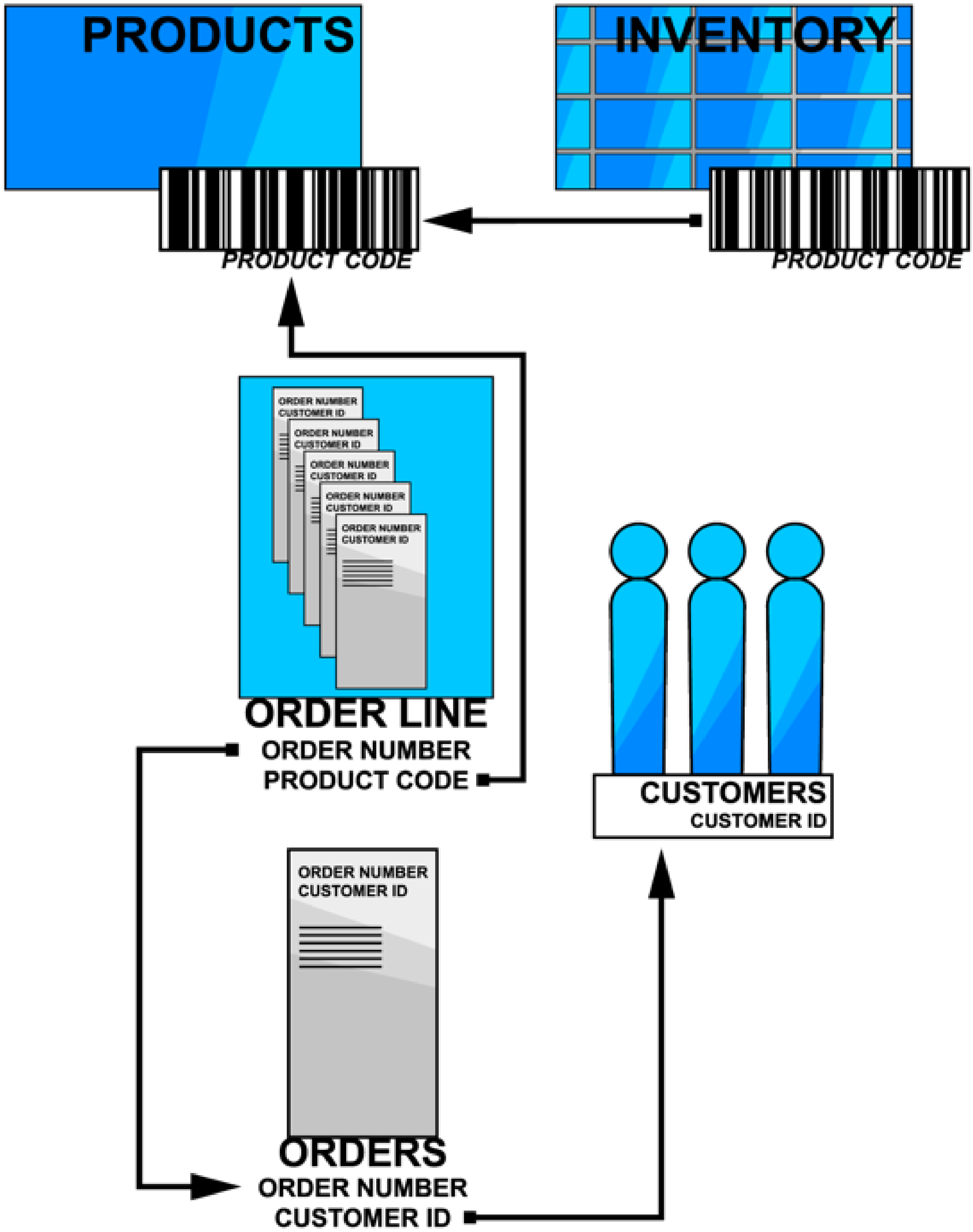
Next, we defined the name column. **FirstName** will be a **varchar** with a maximum length of **50** characters. This should be a required value for all customers, so it is defined as **NOT NULL**. The columns for LastName and AddressLine1 are defined similarly.

AddressLine2 is also a **varchar** of maximum length **50** characters, but since some people 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**, 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 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;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Next, create a table to hold **Orders**. This table will contain an auto-increment column to uniquely identify each order.

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;
Query OK, 0 rows affected (0.01 sec)



mysql>
```

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;
Query OK, 0 rows affected (0.01 sec)



mysql>
```

This table has something new:

OBSERVE:

```
PRIMARY KEY(OrderNumber, ProductCode)
```

The **PRIMARY KEY** is defined on two columns. A PRIMARY KEY uniquely identifies a row in a table. Often 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 there!

INSERT, SELECT, UPDATE, DELETE

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;
+--------------+--------------+------+-----+---------+----------------+
| 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

To add data to a table, use the **INSERT** keyword.

Type the following at the MySQL prompt:

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

Congratulations, you just added a row of data into your table! Let's look closer.

OBSERVE:

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

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 same order you specified the **columns**.

Now let's insert another record.

Type the following at the MySQL prompt:

```
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. When we created the table, we defined the column "AddressLine1" to be NOT NULL. We didn't provide a value for AddressLine1 in our insert statement, the database set it to NULL by default, resulting in this error.

> **WARNING** If you see something like **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—to do this, type **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'
);
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;
+------------+-----------+----------+-----------------+--------------+---------+-------
+------------+------------------+---------------------+
| CustomerID | FirstName | LastName | AddressLine1    | AddressLine2 | City    | State
| PostalCode | EmailAddress     | DateAdded           |
+------------+-----------+----------+-----------------+--------------+---------+-------
+------------+------------------+---------------------+
|         21 | John      | Smith    | 123 4th Street  | NULL         | Chicago | IL
| 60606      | john@hotmail.com | 2007-09-19 20:36:38 |
|         31 | Jane      | Adams    | 300 N. Michigan | NULL         | Chicago | NULL
| NULL       | jane@hotmail.com | 2007-09-19 20:52:36 |
+------------+-----------+----------+-----------------+--------------+---------+-------
+------------+------------------+---------------------+
2 rows in set (0.00 sec)

mysql>
```

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

Your output 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. As you learn more about MySQL, you'll discover other ways to present the information. For example, try **SELECT * FROM Customers\G**. We won't get into this in detail here, but for an idea of what's available, see the MySQL developer documentation.

In a SELECT statement (as in many computer commands), the asterisk (*) means "all columns." Thus the SQL statement you entered could be translated to English as "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;
+------------+-----------+----------+------------------+
| CustomerID | FirstName | LastName | EmailAddress     |
+------------+-----------+----------+------------------+
|         21 | John      | Smith    | john@hotmail.com |
|         31 | Jane      | Adams    | jane@hotmail.com |
+------------+-----------+----------+------------------+
2 rows in set (0.00 sec)

mysql>
```

OBSERVE:

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

This time, MySQL only retrieves the columns you specified (**CustomerID, FirstName, LastName, and EmailAddress**).

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';
+------------+-----------+----------+-----------------+-------+
| CustomerID | FirstName | LastName | EmailAddress    | State |
+------------+-----------+----------+-----------------+-------+
|         21 | John      | Smith    | john@hotmail.com | IL    |
+------------+-----------+----------+-----------------+-------+
1 row in set (0.01 sec)

mysql>
```

OBSERVE:

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

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

This time Ms. Adams isn't included because we didn't specify a state for her. We know Chicago is in Illinois, but SQL doesn't!

What if you only want to see your customers from Illinois who have a gmail address? 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';
Empty set (0.00 sec)

mysql>
```

We don't have any customers in Illinois with gmail accounts, so no rows were returned. This query shows something new.

OBSERVE:

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

AND is a conjunction that connects the two queries, 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 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 *id* 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=id;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

OBSERVE:

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

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

mysql>
```

Yes, it has been updated.

The **WHERE** clause is not required, but if you don't provide one, 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 that 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 *id* to whatever CustomerID your individual John Smith was given.

Type the following at the MySQL prompt:

```
mysql> SELECT *
    -> FROM Customers
    -> WHERE CustomerID=21;
+------------+-----------+----------+----------------+--------------+---------+-------+
------------+---------------+---------------------+
| CustomerID | FirstName | LastName | AddressLine1   | AddressLine2 | City    | State |
 PostalCode | EmailAddress   | DateAdded           |
+------------+-----------+----------+----------------+--------------+---------+-------+
------------+---------------+---------------------+
|         21 | John      | Smith    | 123 4th Street | NULL         | Chicago | IL    |
 60606      | john@gmail.com | 2007-09-19 20:36:38 |
+------------+-----------+----------+----------------+--------------+---------+-------+
------------+---------------+---------------------+
1 row in set (0.00 sec)

mysql>
```

One row was returned, with the correct customer.

Now that we're confident that we've written the 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=id;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

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

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

Delete

Now that 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. of **31**. Double-check her CustomerID with SELECT and replace *id* in the command with her CustomerID from your system.

Type the following at the MySQL prompt:

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

mysql>
```

Yep, it's gone!

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 there!

Transactions

Making Sure Your Commands Behave

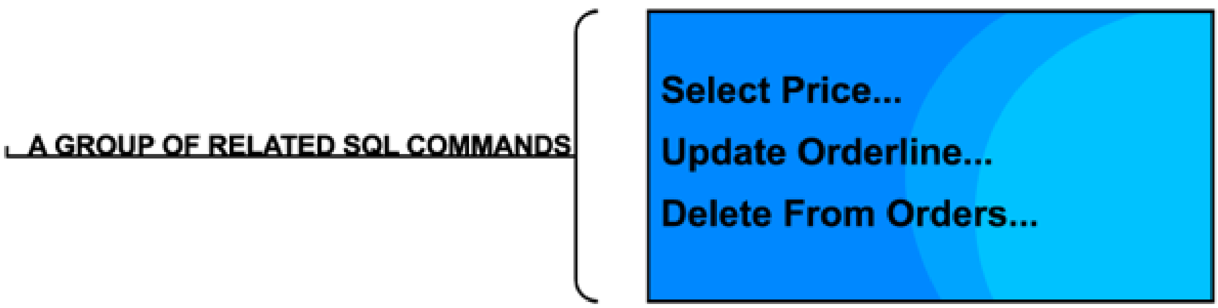
Welcome back! In the last lesson you learned how to store and retrieve data from your database. You also learned to create tables in your database to track customers, inventory, and sales. In this lesson we'll discuss the steps you can take to make sure the data in your database is always correct and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

For this lesson, you'll need to have two Unix terminals open at the same time. The two sessions will be open in different tabs. To tell them apart, use the position of the tabs, and the tabs' names ("Terminal1" and "Terminal2," or similar). 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?

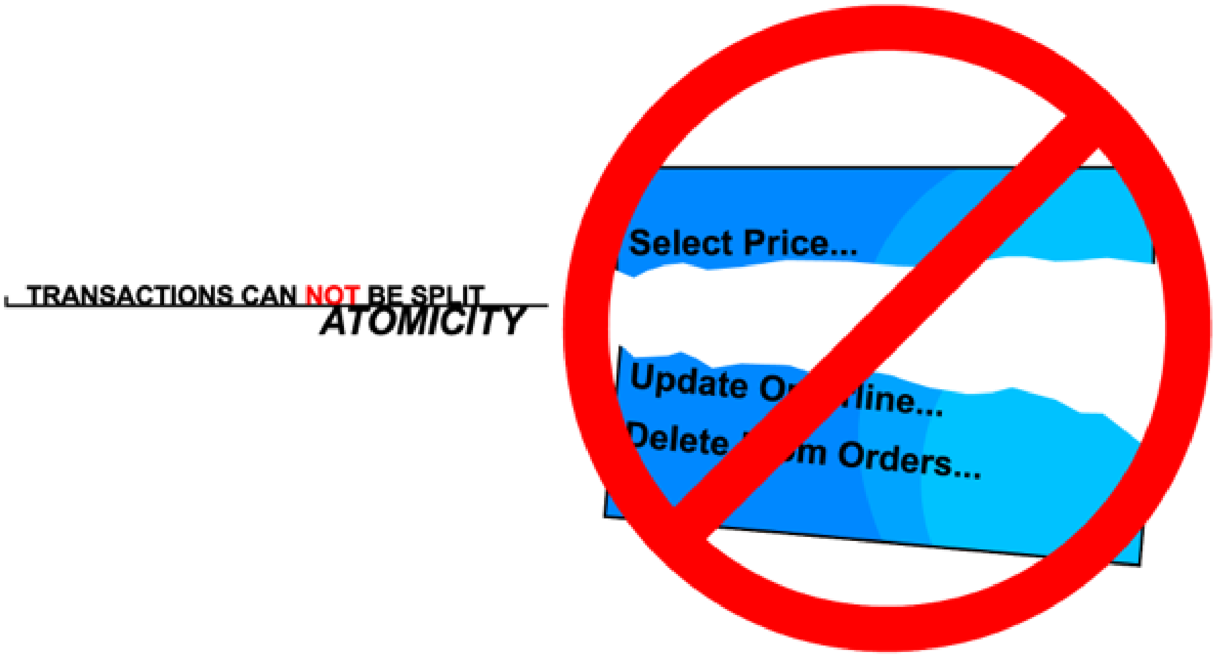


In the database world, *transactions* are used to group related commands so all commands execute or all commands do not execute. There is no in-between.

ACID

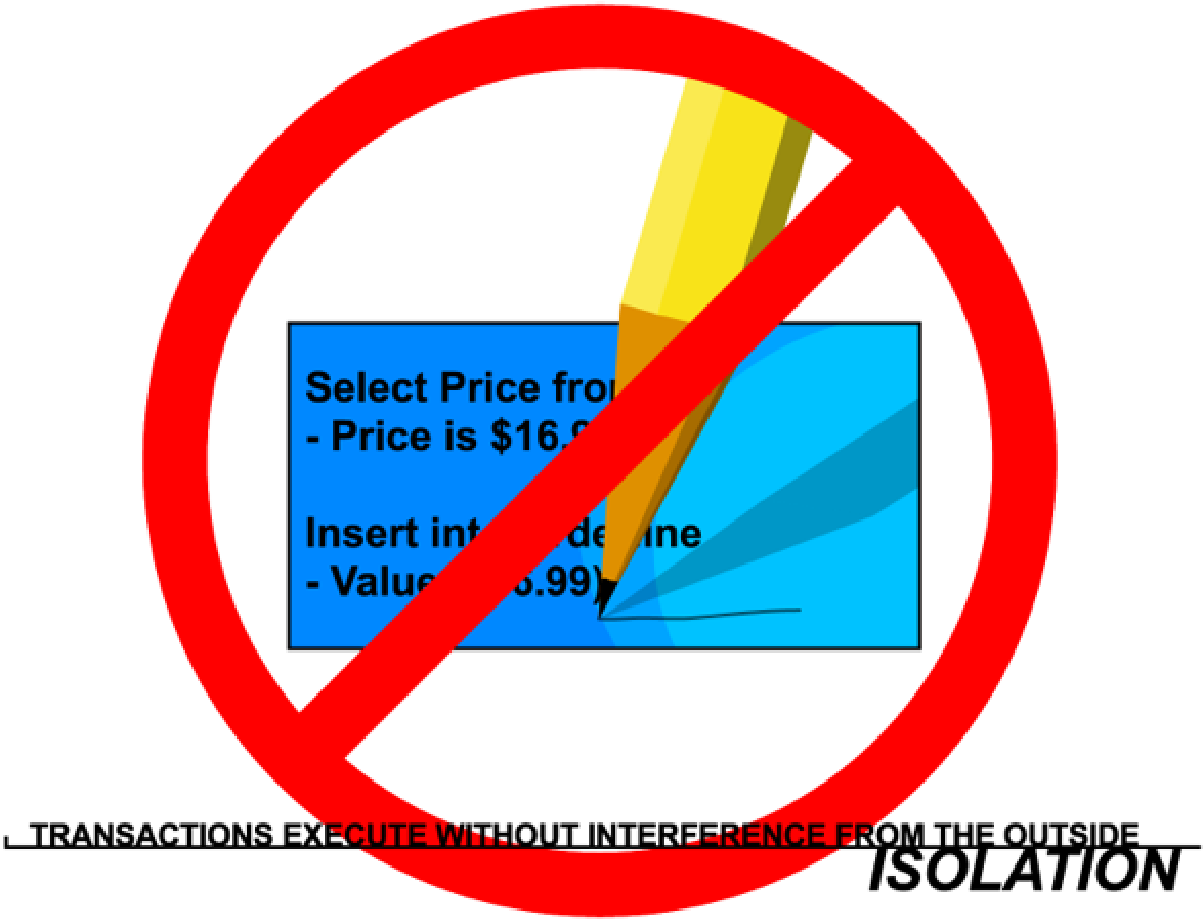
Databases must maintain four properties: **Atomicity**, **Consistency**, **Isolation**, and **Durability**. To help you remember this, use the acronym **ACID**.

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

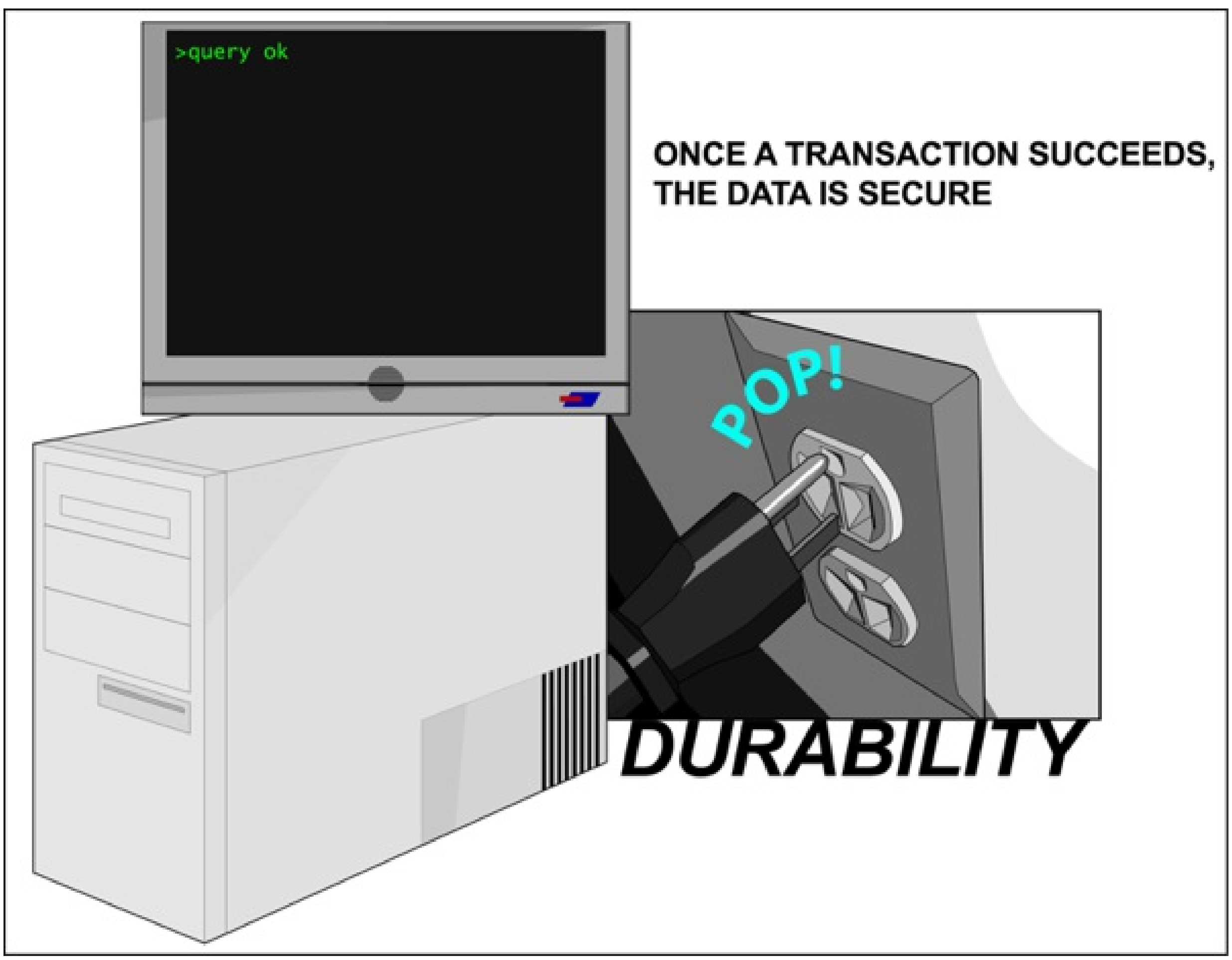


Consistency: Databases only store data that is consistent with the rules defined for storing data. For instance, if an inventory table isn't allowed to have negative quantities, then no rows will be allowed to have negative quantities.

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 this 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 and automatically. The Isolation property depends on the user—it's usually set to a secure level by default, 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 set up, transactions may not secure your data. To ensure MySQL will secure your data, always use the **InnoDB** engine.

Transaction Isolation Levels

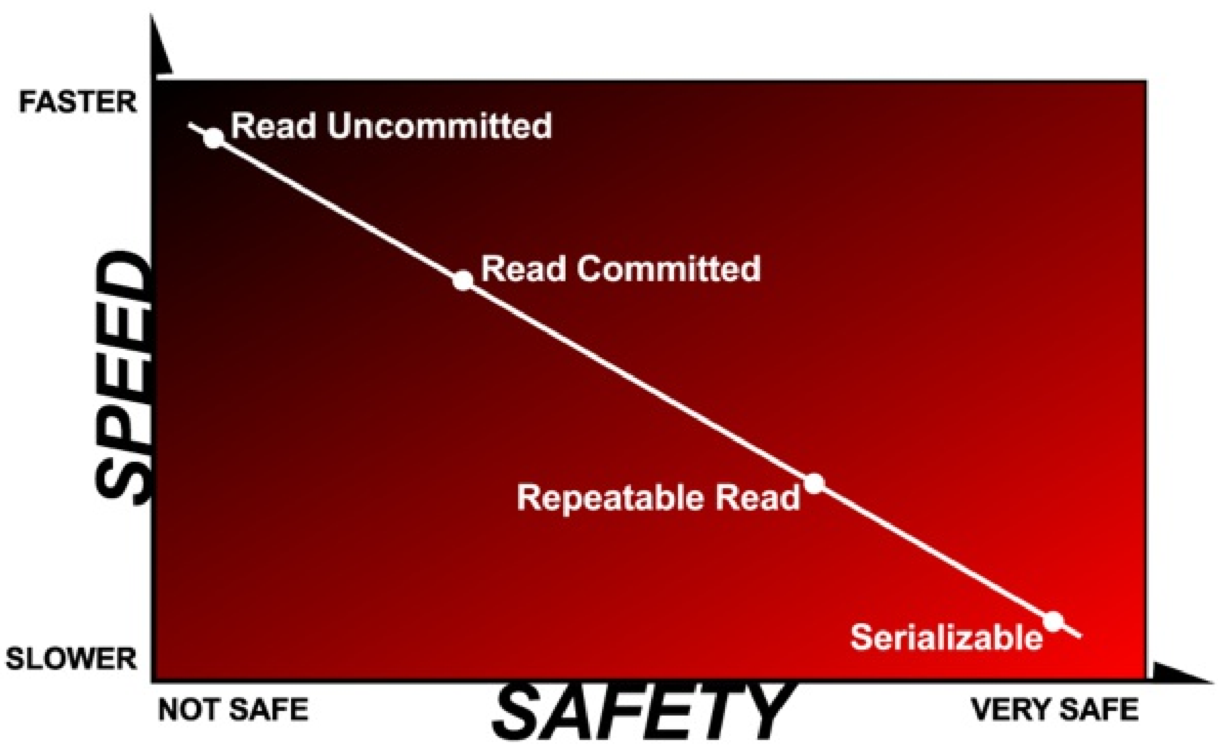
There are four primary transaction isolation levels supported by databases.

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 if the data is incorrect.

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;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

This command doesn't return any rows.

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, and end it with **COMMIT**, or undo it with **ROLLBACK**. As always, you must end each individual SQL statement with a semicolon (;).

Let's create a simple transaction.

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

```
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.

OBSERVE:

```
mysql> START TRANSACTION;
mysql> SELECT LastName FROM Customers;
mysql> 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;**.

Of course, this example only selected data from the database—it didn'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.

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

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO Customers
    -> (FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAdd
ress)
    -> VALUES ('John', 'Doe', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'johnd
oe@hotmail.com');
Query OK, 1 row affected (0.00 sec)

mysql>
```

Next, open a second Unix terminal session, start mysql, and set sql_mode='traditional'.

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

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

mysql>
```

Your table may have more rows than the table shown above, but you should see that **John Doe** is not there.

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 **SET transaction isolation level read committed;** on the first terminal. Switch back to the first terminal, 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;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Switch back to the second Unix terminal, where you typed in the SELECT statement and run it again (you can use the up arrow on your keyboard to repeat the command).

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

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

mysql> INSERT INTO Customers
    -> (FirstName, LastName, AddressLine1, City, State, PostalCode, DateAdded, EmailAdd
ress)
    -> VALUES ('Becky', 'Stein', '123 4th Street', 'Chicago', 'IL', '60606', NOW(), 'bs
tein@hotmail.com');
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 and repeat the SELECT:

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

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

mysql>
```

Your output may be different from the above, but since you *rolled back* the transaction, you don't see any rows for Becky Stein.

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

Combining Tables with a Join

In this lesson, 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. We haven't "officially" added any records to these tables, but if you've added any, you can delete them using the **DELETE** keyword you learned about earlier.

Type the following at the MySQL prompt:

```
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 an 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, get back to the Unix prompt.

Type the following at the MySQL prompt:

```
mysql> exit;
Bye
cold1:~$
```

Next, we'll grab the file from O'Reilly's servers using the **curl** command.

Type the following at the Unix prompt:

```
cold1:~$ curl -L https://courses.oreillyschool.com/dba1/downloads/products.sql > produc
ts.sql
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  9520  100  9520    0     0  53816      0 --:--:-- --:--:-- --:--:-- 178k
cold1:~$
```

Once that file has downloaded, we'll use the **mysql** command to import the products (replace *username* with your username). If the data is imported successfully, you'll see no results.

Type the following at the Unix prompt:

```
cold1:~$ mysql -h sql -p -u username username < products.sql
Enter password:
cold1:~$
```

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);
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!

INTERACTIVE SESSION:

```
mysql> SELECT * FROM Products JOIN Inventory;
+-------------+------------------------------------+-----------------+-----------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
----------------------------------------+-------+-------------+-----------------+
| ProductCode | Title                              | Category        | Description











                                                   | Price | productcode | QuantityInSt
ock |
+-------------+------------------------------------+-----------------+-----------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
```

```
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------+---------+-----------+-----------------+
| databaseid  | Database in Depth                   | Database Theory |This book sheds l
ight on the principles behind the relational model, which is fundamental to all databas
e-backed applications--and, consequently, most of the work that goes on in the computin
g world today. Database in Depth: The Relational Model for Practitioners goes beyond th
e hype and gets to the heart of how relational databases actually work. Ideal for exper
ienced database developers and designers, this concise guide gives you a clear view of 
the technology--a view that's not influenced by any vendor or product. Featuring an ext
ensive set of exercises, it will help you: * understand why and how the relational mode
l is still directly relevant to modern database technology (and will remain so for the 
foreseeable future) * see why and how the SQL standard is seriously deficient * use the
 best current theoretical knowledge in the design of their databases and database appli
cations * make informed decisions in their daily database professional activities Datab
ase in Depth will appeal not only to database developers and designers, but also to a d
iverse field of professionals and academics, including database administrators (DBAs),
information modelers, database consultants, and more. Virtually everyone who deals with
 relational databases should have at least a passing understanding of the fundamentals
of working with relational models. Author C.J. Date has been involved with the relation
al model from its earliest days. An exceptionally clear-thinking writer, Date lays out 
principle and theory in a manner that is easily understood. Few others can speak as aut
horitatively the topic of relational databases as Date can. | 29.95 | artofsql    |
          52 |
| sqlhks      | SQL Hacks                           | SQL             |Whether you're ru
nning Access, MySQL, SQL Server, Oracle, or PostgreSQL, this book will help you push th
e limits of traditional SQL to squeeze data effectively from your database. The book of
fers 100 hacks -- unique tips and tools -- that bring you the knowledge of experts who 
apply what they know in the real world to help you take full advantage of the expressiv
e power of SQL. You'll find practical techniques to address complex data manipulation p
roblems. Learn how to: * Wrangle data in the most efficient way possible * Aggregate an
d organize your data for meaningful and accurate reporting * Make the most of subquerie
s, joins, and unions * Stay on top of the performance of your queries and the server th
at runs them * Avoid common SQL security pitfalls, including the dreaded SQL injection 
attack Let SQL Hacks serve as your toolbox for digging up and manipulating data. If you
 love to tinker and optimize, SQL is the perfect technology and SQL Hacks is the must-h
ave book for you. | 29.99 | artofsql    |          52 |
| artofsql    | The Art of SQL                      | Database Theory |For all the buzz 
about trendy IT techniques, data processing is still at the core of our systems, especi
ally now that enterprises all over the world are confronted with exploding volumes of d
ata. Database performance has become a major headache, and most IT departments believe 
that developers should provide simple SQL code to solve immediate problems and let DBAs
 tune any "bad SQL" later. In The Art of SQL, author and SQL expert Stephane Faroult ar
gues that this "safe approach" only leads to disaster. His insightful book, named after
 Art of War by Sun Tzu, contends that writing quick inefficient code is sweeping the di
rt under the rug. SQL code may run for 5 to 10 years, surviving several major releases 
of the database management system and on several generations of hardware. The code must
 be fast and sound from the start, and that requires a firm understanding of SQL and re
lational theory. The Art of SQL offers best practices that teach experienced SQL users 
to focus on strategy rather than specifics. Faroult's approach takes a page from Sun Tz
u's classic treatise by viewing database design as a military campaign. You need knowle
dge, skills, and talent. Talent can't be taught, but every strategist from Sun Tzu to m
odern-day generals believed that it can be nurtured through the experience of others. T
hey passed on their experience acquired in the field through basic principles that serv
ed as guiding stars amid the sound and fury of battle. This is what Faroult does with S
QL. Like a successful battle plan, good architectural choices are based on contingencie
s. What if the volume of this or that table increases unexpectedly? What if, following 
a merger, the number of users doubles? What if you want to keep several years of data o
nline? Faroult's way of looking at SQL performance may be unconventional and unique, bu
```

```
t he's deadly serious about writing good SQL and using SQL well. The Art of SQL is not
a cookbook, listing problems and giving recipes. The aim is to get you-and your manager
-to raise good questions. | 44.99 | artofsql    |             52 |
| sqltuning   | SQL Tuning                          | Database Theory |A poorly performi
ng database application not only costs users time, but also has an impact on other appl
ications running on the same computer or the same network. SQL Tuning provides an essen
tial next step for SQL developers and database administrators who want to extend their
SQL tuning expertise and get the most from their database applications. There are two b
asic issues to focus on when tuning SQL: how to find and interpret the execution plan o
f an SQL statement and how to change SQL to get a specific alternate execution plan. SQ
L Tuning provides answers to these questions and addresses a third issue that's even mo
re important: how to find the optimal execution plan for the query to use. Author Dan T
ow outlines a timesaving method he's developed for finding the optimum execution plan--
rapidly and systematically--regardless of the complexity of the SQL or the database pla
tform being used. You'll learn how to understand and control SQL execution plans and ho
w to diagram SQL queries to deduce the best execution plan for a query. Key chapters in
 the book include exercises to reinforce the concepts you've learned. SQL Tuning conclu
des by addressing special concerns and unique solutions to "unsolvable problems." Wheth
er you are a programmer who develops SQL-based applications or a database administrator
 or other who troubleshoots poorly tuned applications, SQL Tuning will arm you with a r
eliable and deterministic method for tuning your SQL queries to gain optimal performanc
e. | 39.95 | artofsql    |             52 |
| mysqlspp    | MySQL Stored Procedure Programming | MySQL           |The implementatio
n of stored procedures in MySQL 5.0 a huge milestone -- one that is expected to lead to
 widespread enterprise adoption of the already extremely popular MySQL database. If you
 are serious about building the web-based database applications of the future, you need
 to get up to speed quickly on how stored procedures work -- and how to build them the
right way. This book, destined to be the bible of stored procedure development, is a re
source that no real MySQL programmer can afford to do without. In the decade since MySQ
L burst on the scene, it has become the dominant open source database, with capabilitie
s and performance rivaling those of commercial RDBMS offerings like Oracle and SQL Serv
er. Along with Linux and PHP, MySQL is at the heart of millions of applications. And no
w, with support for stored procedures, functions, and triggers in MySQL 5.0, MySQL offe
rs the programming power needed for true enterprise use. MySQL's new procedural languag
e has a straightforward syntax, making it easy to write simple programs. But it's not s
o easy to write secure, easily maintained, high-performance, and bug-free programs. Few
 in the MySQL world have substantial experience yet with stored procedures, but Guy Har
rison and Steven Feuerstein have decades of combined expertise. In MySQL Stored Procedu
re Programming, they put that hard-won experience to good use. Packed with code example
s and covering everything from language basics to application building to advanced tuni
ng and best practices, this highly readable book is the one-stop guide to MySQL develop
ment. It consists of four major sections: * MySQL stored programming fundamentals -- tu
torial, basic statements, SQL in stored programs, and error handling * Building MySQL s
tored programs -- transaction handling, built-in functions, stored functions, and trigg
ers * MySQL stored programs in applications -- using stored programs with PHP, Java, Pe
rl, Python, and .NET (C# and VB.NET) * Optimizing MySQL stored programs -- security, ba
sic and advanced SQL tuning, optimizing stored program code, and programming best pract
ices A companion web site contains many thousands of lines of code, that you can put to
 use immediately. Guy Harrison is Chief Architect of Database Solutions at Quest Softwa
re and a frequent speaker and writer on MySQL topics. Steven Feuerstein is the author o
f Oracle PL/SQL Programming, the classic reference for Oracle stored programming for mo
re than ten years. Both have decades of experience as database developers, and between
them they have authored a dozen books. | 44.99 | artofsql    |             52 |
| databaseid  | Database in Depth                   | Database Theory |This book sheds l
ight on the principles behind the relational model, which is fundamental to all databas
e-backed applications--and, consequently, most of the work that goes on in the computin
g world today. Database in Depth: The Relational Model for Practitioners goes beyond th
e hype and gets to the heart of how relational databases actually work. Ideal for exper
ienced database developers and designers, this concise guide gives you a clear view of
the technology--a view that's not influenced by any vendor or product. Featuring an ext
ensive set of exercises, it will help you: * understand why and how the relational mode
l is still directly relevant to modern database technology (and will remain so for the
foreseeable future) * see why and how the SQL standard is seriously deficient * use the
 best current theoretical knowledge in the design of their databases and database appli
cations * make informed decisions in their daily database professional activities Datab
ase in Depth will appeal not only to database developers and designers, but also to a d
iverse field of professionals and academics, including database administrators (DBAs),
information modelers, database consultants, and more. Virtually everyone who deals with
```

```
 relational databases should have at least a passing understanding of the fundamentals
of working with relational models. Author C.J. Date has been involved with the relation
al model from its earliest days. An exceptionally clear-thinking writer, Date lays out
principle and theory in a manner that is easily understood. Few others can speak as aut
horitatively the topic of relational databases as Date can. | 29.95 | databaseid  |
           0 |
| sqlhks      | SQL Hacks                          | SQL             |Whether you're ru
nning Access, MySQL, SQL Server, Oracle, or PostgreSQL, this book will help you push th
e limits of traditional SQL to squeeze data effectively from your database. The book of
fers 100 hacks -- unique tips and tools -- that bring you the knowledge of experts who
apply what they know in the real world to help you take full advantage of the expressiv
e power of SQL. You'll find practical techniques to address complex data manipulation p
roblems. Learn how to: * Wrangle data in the most efficient way possible * Aggregate an
d organize your data for meaningful and accurate reporting * Make the most of subquerie
s, joins, and unions * Stay on top of the performance of your queries and the server th
at runs them * Avoid common SQL security pitfalls, including the dreaded SQL injection
attack Let SQL Hacks serve as your toolbox for digging up and manipulating data. If you
 love to tinker and optimize, SQL is the perfect technology and SQL Hacks is the must-h
ave book for you. | 29.99 | databaseid  |                0 |
| artofsql    | The Art of SQL                     | Database Theory |For all the buzz
about trendy IT techniques, data processing is still at the core of our systems, especi
ally now that enterprises all over the world are confronted with exploding volumes of d
ata. Database performance has become a major headache, and most IT departments believe
that developers should provide simple SQL code to solve immediate problems and let DBAs
 tune any "bad SQL" later. In The Art of SQL, author and SQL expert Stephane Faroult ar
gues that this "safe approach" only leads to disaster. His insightful book, named after
 Art of War by Sun Tzu, contends that writing quick inefficient code is sweeping the di
rt under the rug. SQL code may run for 5 to 10 years, surviving several major releases
of the database management system and on several generations of hardware. The code must
 be fast and sound from the start, and that requires a firm understanding of SQL and re
lational theory. The Art of SQL offers best practices that teach experienced SQL users
to focus on strategy rather than specifics. Faroult's approach takes a page from Sun Tz
u's classic treatise by viewing database design as a military campaign. You need knowle
dge, skills, and talent. Talent can't be taught, but every strategist from Sun Tzu to m
odern-day generals believed that it can be nurtured through the experience of others. T
hey passed on their experience acquired in the field through basic principles that serv
ed as guiding stars amid the sound and fury of battle. This is what Faroult does with S
QL. Like a successful battle plan, good architectural choices are based on contingencie
s. What if the volume of this or that table increases unexpectedly? What if, following
a merger, the number of users doubles? What if you want to keep several years of data o
nline? Faroult's way of looking at SQL performance may be unconventional and unique, bu
t he's deadly serious about writing good SQL and using SQL well. The Art of SQL is not
a cookbook, listing problems and giving recipes. The aim is to get you-and your manager
-to raise good questions. | 44.99 | databaseid  |                0 |
| sqltuning   | SQL Tuning                         | Database Theory |A poorly performi
ng database application not only costs users time, but also has an impact on other appl
ications running on the same computer or the same network. SQL Tuning provides an essen
tial next step for SQL developers and database administrators who want to extend their
SQL tuning expertise and get the most from their database applications. There are two b
asic issues to focus on when tuning SQL: how to find and interpret the execution plan o
f an SQL statement and how to change SQL to get a specific alternate execution plan. SQ
L Tuning provides answers to these questions and addresses a third issue that's even mo
re important: how to find the optimal execution plan for the query to use. Author Dan T
ow outlines a timesaving method he's developed for finding the optimum execution plan--
rapidly and systematically--regardless of the complexity of the SQL or the database pla
tform being used. You'll learn how to understand and control SQL execution plans and ho
w to diagram SQL queries to deduce the best execution plan for a query. Key chapters in
 the book include exercises to reinforce the concepts you've learned. SQL Tuning conclu
des by addressing special concerns and unique solutions to "unsolvable problems." Wheth
er you are a programmer who develops SQL-based applications or a database administrator
 or other who troubleshoots poorly tuned applications, SQL Tuning will arm you with a r
eliable and deterministic method for tuning your SQL queries to gain optimal performanc
e. | 39.95 | databaseid  |                0 |
| mysqlspp    | MySQL Stored Procedure Programming | MySQL           |The implementatio
n of stored procedures in MySQL 5.0 a huge milestone -- one that is expected to lead to
 widespread enterprise adoption of the already extremely popular MySQL database. If you
 are serious about building the web-based database applications of the future, you need
 to get up to speed quickly on how stored procedures work -- and how to build them the
```

```
right way. This book, destined to be the bible of stored procedure development, is a re
source that no real MySQL programmer can afford to do without. In the decade since MySQ
L burst on the scene, it has become the dominant open source database, with capabilitie
s and performance rivaling those of commercial RDBMS offerings like Oracle and SQL Serv
er. Along with Linux and PHP, MySQL is at the heart of millions of applications. And no
w, with support for stored procedures, functions, and triggers in MySQL 5.0, MySQL offe
rs the programming power needed for true enterprise use. MySQL's new procedural languag
e has a straightforward syntax, making it easy to write simple programs. But it's not s
o easy to write secure, easily maintained, high-performance, and bug-free programs. Few
 in the MySQL world have substantial experience yet with stored procedures, but Guy Har
rison and Steven Feuerstein have decades of combined expertise. In MySQL Stored Procedu
re Programming, they put that hard-won experience to good use. Packed with code example
s and covering everything from language basics to application building to advanced tuni
ng and best practices, this highly readable book is the one-stop guide to MySQL develop
ment. It consists of four major sections: * MySQL stored programming fundamentals -- tu
torial, basic statements, SQL in stored programs, and error handling * Building MySQL s
tored programs -- transaction handling, built-in functions, stored functions, and trigg
ers * MySQL stored programs in applications -- using stored programs with PHP, Java, Pe
rl, Python, and .NET (C# and VB.NET) * Optimizing MySQL stored programs -- security, ba
sic and advanced SQL tuning, optimizing stored program code, and programming best pract
ices A companion web site contains many thousands of lines of code, that you can put to
 use immediately. Guy Harrison is Chief Architect of Database Solutions at Quest Softwa
re and a frequent speaker and writer on MySQL topics. Steven Feuerstein is the author o
f Oracle PL/SQL Programming, the classic reference for Oracle stored programming for mo
re than ten years. Both have decades of experience as database developers, and between
them they have authored a dozen books. | 44.99 | databaseid  |              0 |
| databaseid  | Database in Depth                    | Database Theory |This book sheds l
ight on the principles behind the relational model, which is fundamental to all databas
e-backed applications--and, consequently, most of the work that goes on in the computin
g world today. Database in Depth: The Relational Model for Practitioners goes beyond th
e hype and gets to the heart of how relational databases actually work. Ideal for exper
ienced database developers and designers, this concise guide gives you a clear view of
the technology--a view that's not influenced by any vendor or product. Featuring an ext
ensive set of exercises, it will help you: * understand why and how the relational mode
l is still directly relevant to modern database technology (and will remain so for the
foreseeable future) * see why and how the SQL standard is seriously deficient * use the
 best current theoretical knowledge in the design of their databases and database appli
cations * make informed decisions in their daily database professional activities Datab
ase in Depth will appeal not only to database developers and designers, but also to a d
iverse field of professionals and academics, including database administrators (DBAs),
information modelers, database consultants, and more. Virtually everyone who deals with
 relational databases should have at least a passing understanding of the fundamentals
of working with relational models. Author C.J. Date has been involved with the relation
al model from its earliest days. An exceptionally clear-thinking writer, Date lays out
principle and theory in a manner that is easily understood. Few others can speak as aut
horitatively the topic of relational databases as Date can. | 29.95 | mysqlspp    |
           5 |
| sqlhks      | SQL Hacks                            | SQL             |Whether you're ru
nning Access, MySQL, SQL Server, Oracle, or PostgreSQL, this book will help you push th
e limits of traditional SQL to squeeze data effectively from your database. The book of
fers 100 hacks -- unique tips and tools -- that bring you the knowledge of experts who
apply what they know in the real world to help you take full advantage of the expressiv
e power of SQL. You'll find practical techniques to address complex data manipulation p
roblems. Learn how to: * Wrangle data in the most efficient way possible * Aggregate an
d organize your data for meaningful and accurate reporting * Make the most of subquerie
s, joins, and unions * Stay on top of the performance of your queries and the server th
at runs them * Avoid common SQL security pitfalls, including the dreaded SQL injection
attack Let SQL Hacks serve as your toolbox for digging up and manipulating data. If you
 love to tinker and optimize, SQL is the perfect technology and SQL Hacks is the must-h
ave book for you. | 29.99 | mysqlspp    |              5 |
| artofsql    | The Art of SQL                       | Database Theory |For all the buzz
about trendy IT techniques, data processing is still at the core of our systems, especi
ally now that enterprises all over the world are confronted with exploding volumes of d
ata. Database performance has become a major headache, and most IT departments believe
that developers should provide simple SQL code to solve immediate problems and let DBAs
 tune any "bad SQL" later. In The Art of SQL, author and SQL expert Stephane Faroult ar
gues that this "safe approach" only leads to disaster. His insightful book, named after
 Art of War by Sun Tzu, contends that writing quick inefficient code is sweeping the di
```

```
rt under the rug. SQL code may run for 5 to 10 years, surviving several major releases
of the database management system and on several generations of hardware. The code must
 be fast and sound from the start, and that requires a firm understanding of SQL and re
lational theory. The Art of SQL offers best practices that teach experienced SQL users
to focus on strategy rather than specifics. Faroult's approach takes a page from Sun Tz
u's classic treatise by viewing database design as a military campaign. You need knowle
dge, skills, and talent. Talent can't be taught, but every strategist from Sun Tzu to m
odern-day generals believed that it can be nurtured through the experience of others. T
hey passed on their experience acquired in the field through basic principles that serv
ed as guiding stars amid the sound and fury of battle. This is what Faroult does with S
QL. Like a successful battle plan, good architectural choices are based on contingencie
s. What if the volume of this or that table increases unexpectedly? What if, following
a merger, the number of users doubles? What if you want to keep several years of data o
nline? Faroult's way of looking at SQL performance may be unconventional and unique, bu
t he's deadly serious about writing good SQL and using SQL well. The Art of SQL is not
a cookbook, listing problems and giving recipes. The aim is to get you-and your manager
-to raise good questions. | 44.99 | mysqlspp    |              5 |
| sqltuning   | SQL Tuning                          | Database Theory |A poorly performi
ng database application not only costs users time, but also has an impact on other appl
ications running on the same computer or the same network. SQL Tuning provides an essen
tial next step for SQL developers and database administrators who want to extend their
SQL tuning expertise and get the most from their database applications. There are two b
asic issues to focus on when tuning SQL: how to find and interpret the execution plan o
f an SQL statement and how to change SQL to get a specific alternate execution plan. SQ
L Tuning provides answers to these questions and addresses a third issue that's even mo
re important: how to find the optimal execution plan for the query to use. Author Dan T
ow outlines a timesaving method he's developed for finding the optimum execution plan--
rapidly and systematically--regardless of the complexity of the SQL or the database pla
tform being used. You'll learn how to understand and control SQL execution plans and ho
w to diagram SQL queries to deduce the best execution plan for a query. Key chapters in
 the book include exercises to reinforce the concepts you've learned. SQL Tuning conclu
des by addressing special concerns and unique solutions to "unsolvable problems." Wheth
er you are a programmer who develops SQL-based applications or a database administrator
 or other who troubleshoots poorly tuned applications, SQL Tuning will arm you with a r
eliable and deterministic method for tuning your SQL queries to gain optimal performanc
e. | 39.95 | mysqlspp    |              5 |
| mysqlspp    | MySQL Stored Procedure Programming | MySQL           |The implementatio
n of stored procedures in MySQL 5.0 a huge milestone -- one that is expected to lead to
 widespread enterprise adoption of the already extremely popular MySQL database. If you
 are serious about building the web-based database applications of the future, you need
 to get up to speed quickly on how stored procedures work -- and how to build them the
right way. This book, destined to be the bible of stored procedure development, is a re
source that no real MySQL programmer can afford to do without. In the decade since MySQ
L burst on the scene, it has become the dominant open source database, with capabilitie
s and performance rivaling those of commercial RDBMS offerings like Oracle and SQL Serv
er. Along with Linux and PHP, MySQL is at the heart of millions of applications. And no
w, with support for stored procedures, functions, and triggers in MySQL 5.0, MySQL offe
rs the programming power needed for true enterprise use. MySQL's new procedural languag
e has a straightforward syntax, making it easy to write simple programs. But it's not s
o easy to write secure, easily maintained, high-performance, and bug-free programs. Few
 in the MySQL world have substantial experience yet with stored procedures, but Guy Har
rison and Steven Feuerstein have decades of combined expertise. In MySQL Stored Procedu
re Programming, they put that hard-won experience to good use. Packed with code example
s and covering everything from language basics to application building to advanced tuni
ng and best practices, this highly readable book is the one-stop guide to MySQL develop
ment. It consists of four major sections: * MySQL stored programming fundamentals -- tu
torial, basic statements, SQL in stored programs, and error handling * Building MySQL s
tored programs -- transaction handling, built-in functions, stored functions, and trigg
ers * MySQL stored programs in applications -- using stored programs with PHP, Java, Pe
rl, Python, and .NET (C# and VB.NET) * Optimizing MySQL stored programs -- security, ba
sic and advanced SQL tuning, optimizing stored program code, and programming best pract
ices A companion web site contains many thousands of lines of code, that you can put to
 use immediately. Guy Harrison is Chief Architect of Database Solutions at Quest Softwa
re and a frequent speaker and writer on MySQL topics. Steven Feuerstein is the author o
f Oracle PL/SQL Programming, the classic reference for Oracle stored programming for mo
re than ten years. Both have decades of experience as database developers, and between
them they have authored a dozen books. | 44.99 | mysqlspp    |              5 |
| databaseid  | Database in Depth                   | Database Theory |This book sheds l
```

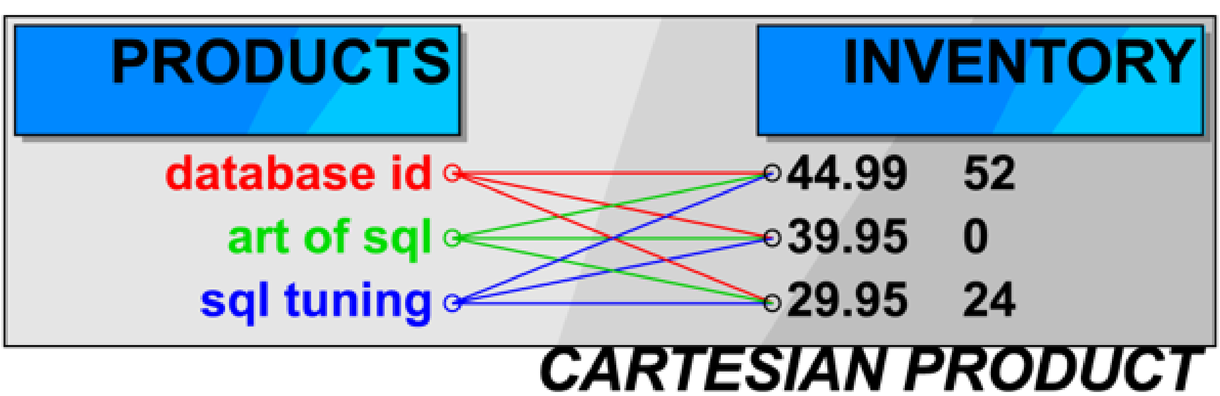
```
ight on the principles behind the relational model, which is fundamental to all databas
e-backed applications--and, consequently, most of the work that goes on in the computin
g world today. Database in Depth: The Relational Model for Practitioners goes beyond th
e hype and gets to the heart of how relational databases actually work. Ideal for exper
ienced database developers and designers, this concise guide gives you a clear view of
the technology--a view that's not influenced by any vendor or product. Featuring an ext
ensive set of exercises, it will help you: * understand why and how the relational mode
l is still directly relevant to modern database technology (and will remain so for the
foreseeable future) * see why and how the SQL standard is seriously deficient * use the
 best current theoretical knowledge in the design of their databases and database appli
cations * make informed decisions in their daily database professional activities Datab
ase in Depth will appeal not only to database developers and designers, but also to a d
iverse field of professionals and academics, including database administrators (DBAs),
information modelers, database consultants, and more. Virtually everyone who deals with
 relational databases should have at least a passing understanding of the fundamentals
of working with relational models. Author C.J. Date has been involved with the relation
al model from its earliest days. An exceptionally clear-thinking writer, Date lays out
principle and theory in a manner that is easily understood. Few others can speak as aut
horitatively the topic of relational databases as Date can. | 29.95 | sqlhks      |
          32 |
| sqlhks      | SQL Hacks                          | SQL             |Whether you're ru
nning Access, MySQL, SQL Server, Oracle, or PostgreSQL, this book will help you push th
e limits of traditional SQL to squeeze data effectively from your database. The book of
fers 100 hacks -- unique tips and tools -- that bring you the knowledge of experts who
apply what they know in the real world to help you take full advantage of the expressiv
e power of SQL. You'll find practical techniques to address complex data manipulation p
roblems. Learn how to: * Wrangle data in the most efficient way possible * Aggregate an
d organize your data for meaningful and accurate reporting * Make the most of subquerie
s, joins, and unions * Stay on top of the performance of your queries and the server th
at runs them * Avoid common SQL security pitfalls, including the dreaded SQL injection
attack Let SQL Hacks serve as your toolbox for digging up and manipulating data. If you
 love to tinker and optimize, SQL is the perfect technology and SQL Hacks is the must-h
ave book for you. | 29.99 | sqlhks      |          32 |
| artofsql    | The Art of SQL                     | Database Theory |For all the buzz
about trendy IT techniques, data processing is still at the core of our systems, especi
ally now that enterprises all over the world are confronted with exploding volumes of d
ata. Database performance has become a major headache, and most IT departments believe
that developers should provide simple SQL code to solve immediate problems and let DBAs
 tune any "bad SQL" later. In The Art of SQL, author and SQL expert Stephane Faroult ar
gues that this "safe approach" only leads to disaster. His insightful book, named after
 Art of War by Sun Tzu, contends that writing quick inefficient code is sweeping the di
rt under the rug. SQL code may run for 5 to 10 years, surviving several major releases
of the database management system and on several generations of hardware. The code must
 be fast and sound from the start, and that requires a firm understanding of SQL and re
lational theory. The Art of SQL offers best practices that teach experienced SQL users
to focus on strategy rather than specifics. Faroult's approach takes a page from Sun Tz
u's classic treatise by viewing database design as a military campaign. You need knowle
dge, skills, and talent. Talent can't be taught, but every strategist from Sun Tzu to m
odern-day generals believed that it can be nurtured through the experience of others. T
hey passed on their experience acquired in the field through basic principles that serv
ed as guiding stars amid the sound and fury of battle. This is what Faroult does with S
QL. Like a successful battle plan, good architectural choices are based on contingencie
s. What if the volume of this or that table increases unexpectedly? What if, following
a merger, the number of users doubles? What if you want to keep several years of data o
nline? Faroult's way of looking at SQL performance may be unconventional and unique, bu
t he's deadly serious about writing good SQL and using SQL well. The Art of SQL is not
a cookbook, listing problems and giving recipes. The aim is to get you-and your manager
-to raise good questions. | 44.99 | sqlhks      |          32 |
| sqltuning   | SQL Tuning                         | Database Theory |A poorly performi
ng database application not only costs users time, but also has an impact on other appl
ications running on the same computer or the same network. SQL Tuning provides an essen
tial next step for SQL developers and database administrators who want to extend their
SQL tuning expertise and get the most from their database applications. There are two b
asic issues to focus on when tuning SQL: how to find and interpret the execution plan o
f an SQL statement and how to change SQL to get a specific alternate execution plan. SQ
L Tuning provides answers to these questions and addresses a third issue that's even mo
re important: how to find the optimal execution plan for the query to use. Author Dan T
ow outlines a timesaving method he's developed for finding the optimum execution plan--
```

rapidly and systematically--regardless of the complexity of the SQL or the database pla
tform being used. You'll learn how to understand and control SQL execution plans and ho
w to diagram SQL queries to deduce the best execution plan for a query. Key chapters in
 the book include exercises to reinforce the concepts you've learned. SQL Tuning conclu
des by addressing special concerns and unique solutions to "unsolvable problems." Wheth
er you are a programmer who develops SQL-based applications or a database administrator
 or other who troubleshoots poorly tuned applications, SQL Tuning will arm you with a r
eliable and deterministic method for tuning your SQL queries to gain optimal performanc
e. | 39.95 | sqlhks | 32 |
| mysqlspp | MySQL Stored Procedure Programming | MySQL |The implementatio
n of stored procedures in MySQL 5.0 a huge milestone -- one that is expected to lead to
 widespread enterprise adoption of the already extremely popular MySQL database. If you
 are serious about building the web-based database applications of the future, you need
 to get up to speed quickly on how stored procedures work -- and how to build them the
right way. This book, destined to be the bible of stored procedure development, is a re
source that no real MySQL programmer can afford to do without. In the decade since MySQ
L burst on the scene, it has become the dominant open source database, with capabilitie
s and performance rivaling those of commercial RDBMS offerings like Oracle and SQL Serv
er. Along with Linux and PHP, MySQL is at the heart of millions of applications. And no
w, with support for stored procedures, functions, and triggers in MySQL 5.0, MySQL offe
rs the programming power needed for true enterprise use. MySQL's new procedural languag
e has a straightforward syntax, making it easy to write simple programs. But it's not s
o easy to write secure, easily maintained, high-performance, and bug-free programs. Few
 in the MySQL world have substantial experience yet with stored procedures, but Guy Har
rison and Steven Feuerstein have decades of combined expertise. In MySQL Stored Procedu
re Programming, they put that hard-won experience to good use. Packed with code example
s and covering everything from language basics to application building to advanced tuni
ng and best practices, this highly readable book is the one-stop guide to MySQL develop
ment. It consists of four major sections: * MySQL stored programming fundamentals -- tu
torial, basic statements, SQL in stored programs, and error handling * Building MySQL s
tored programs -- transaction handling, built-in functions, stored functions, and trigg
ers * MySQL stored programs in applications -- using stored programs with PHP, Java, Pe
rl, Python, and .NET (C# and VB.NET) * Optimizing MySQL stored programs -- security, ba
sic and advanced SQL tuning, optimizing stored program code, and programming best pract
ices A companion web site contains many thousands of lines of code, that you can put to
 use immediately. Guy Harrison is Chief Architect of Database Solutions at Quest Softwa
re and a frequent speaker and writer on MySQL topics. Steven Feuerstein is the author o
f Oracle PL/SQL Programming, the classic reference for Oracle stored programming for mo
re than ten years. Both have decades of experience as database developers, and between
them they have authored a dozen books. | 44.99 | sqlhks | 32 |
| databaseid | Database in Depth | Database Theory |This book sheds l
ight on the principles behind the relational model, which is fundamental to all databas
e-backed applications--and, consequently, most of the work that goes on in the computin
g world today. Database in Depth: The Relational Model for Practitioners goes beyond th
e hype and gets to the heart of how relational databases actually work. Ideal for exper
ienced database developers and designers, this concise guide gives you a clear view of
the technology--a view that's not influenced by any vendor or product. Featuring an ext
ensive set of exercises, it will help you: * understand why and how the relational mode
l is still directly relevant to modern database technology (and will remain so for the
foreseeable future) * see why and how the SQL standard is seriously deficient * use the
 best current theoretical knowledge in the design of their databases and database appli
cations * make informed decisions in their daily database professional activities Datab
ase in Depth will appeal not only to database developers and designers, but also to a d
iverse field of professionals and academics, including database administrators (DBAs),
information modelers, database consultants, and more. Virtually everyone who deals with
 relational databases should have at least a passing understanding of the fundamentals
of working with relational models. Author C.J. Date has been involved with the relation
al model from its earliest days. An exceptionally clear-thinking writer, Date lays out
principle and theory in a manner that is easily understood. Few others can speak as aut
horitatively the topic of relational databases as Date can. | 29.95 | sqltuning |
         105 |
| sqlhks | SQL Hacks | SQL |Whether you're ru
nning Access, MySQL, SQL Server, Oracle, or PostgreSQL, this book will help you push th
e limits of traditional SQL to squeeze data effectively from your database. The book of
fers 100 hacks -- unique tips and tools -- that bring you the knowledge of experts who
apply what they know in the real world to help you take full advantage of the expressiv
e power of SQL. You'll find practical techniques to address complex data manipulation p
roblems. Learn how to: * Wrangle data in the most efficient way possible * Aggregate an

```
d organize your data for meaningful and accurate reporting * Make the most of subquerie
s, joins, and unions * Stay on top of the performance of your queries and the server th
at runs them * Avoid common SQL security pitfalls, including the dreaded SQL injection
attack Let SQL Hacks serve as your toolbox for digging up and manipulating data. If you
 love to tinker and optimize, SQL is the perfect technology and SQL Hacks is the must-h
ave book for you. | 29.99 | sqltuning   |            105 |
| artofsql    | The Art of SQL                      | Database Theory |For all the buzz
about trendy IT techniques, data processing is still at the core of our systems, especi
ally now that enterprises all over the world are confronted with exploding volumes of d
ata. Database performance has become a major headache, and most IT departments believe
that developers should provide simple SQL code to solve immediate problems and let DBAs
 tune any "bad SQL" later. In The Art of SQL, author and SQL expert Stephane Faroult ar
gues that this "safe approach" only leads to disaster. His insightful book, named after
 Art of War by Sun Tzu, contends that writing quick inefficient code is sweeping the di
rt under the rug. SQL code may run for 5 to 10 years, surviving several major releases
of the database management system and on several generations of hardware. The code must
 be fast and sound from the start, and that requires a firm understanding of SQL and re
lational theory. The Art of SQL offers best practices that teach experienced SQL users
to focus on strategy rather than specifics. Faroult's approach takes a page from Sun Tz
u's classic treatise by viewing database design as a military campaign. You need knowle
dge, skills, and talent. Talent can't be taught, but every strategist from Sun Tzu to m
odern-day generals believed that it can be nurtured through the experience of others. T
hey passed on their experience acquired in the field through basic principles that serv
ed as guiding stars amid the sound and fury of battle. This is what Faroult does with S
QL. Like a successful battle plan, good architectural choices are based on contingencie
s. What if the volume of this or that table increases unexpectedly? What if, following
a merger, the number of users doubles? What if you want to keep several years of data o
nline? Faroult's way of looking at SQL performance may be unconventional and unique, bu
t he's deadly serious about writing good SQL and using SQL well. The Art of SQL is not
a cookbook, listing problems and giving recipes. The aim is to get you-and your manager
-to raise good questions. | 44.99 | sqltuning   |            105 |
| sqltuning   | SQL Tuning                          | Database Theory |A poorly performi
ng database application not only costs users time, but also has an impact on other appl
ications running on the same computer or the same network. SQL Tuning provides an essen
tial next step for SQL developers and database administrators who want to extend their
SQL tuning expertise and get the most from their database applications. There are two b
asic issues to focus on when tuning SQL: how to find and interpret the execution plan o
f an SQL statement and how to change SQL to get a specific alternate execution plan. SQ
L Tuning provides answers to these questions and addresses a third issue that's even mo
re important: how to find the optimal execution plan for the query to use. Author Dan T
ow outlines a timesaving method he's developed for finding the optimum execution plan--
rapidly and systematically--regardless of the complexity of the SQL or the database pla
tform being used. You'll learn how to understand and control SQL execution plans and ho
w to diagram SQL queries to deduce the best execution plan for a query. Key chapters in
 the book include exercises to reinforce the concepts you've learned. SQL Tuning conclu
des by addressing special concerns and unique solutions to "unsolvable problems." Wheth
er you are a programmer who develops SQL-based applications or a database administrator
 or other who troubleshoots poorly tuned applications, SQL Tuning will arm you with a r
eliable and deterministic method for tuning your SQL queries to gain optimal performanc
e. | 39.95 | sqltuning   |            105 |
| mysqlspp    | MySQL Stored Procedure Programming | MySQL           |The implementatio
n of stored procedures in MySQL 5.0 a huge milestone -- one that is expected to lead to
 widespread enterprise adoption of the already extremely popular MySQL database. If you
 are serious about building the web-based database applications of the future, you need
 to get up to speed quickly on how stored procedures work -- and how to build them the
right way. This book, destined to be the bible of stored procedure development, is a re
source that no real MySQL programmer can afford to do without. In the decade since MySQ
L burst on the scene, it has become the dominant open source database, with capabilitie
s and performance rivaling those of commercial RDBMS offerings like Oracle and SQL Serv
er. Along with Linux and PHP, MySQL is at the heart of millions of applications. And no
w, with support for stored procedures, functions, and triggers in MySQL 5.0, MySQL offe
rs the programming power needed for true enterprise use. MySQL's new procedural languag
e has a straightforward syntax, making it easy to write simple programs. But it's not s
o easy to write secure, easily maintained, high-performance, and bug-free programs. Few
 in the MySQL world have substantial experience yet with stored procedures, but Guy Har
rison and Steven Feuerstein have decades of combined expertise. In MySQL Stored Procedu
re Programming, they put that hard-won experience to good use. Packed with code example
s and covering everything from language basics to application building to advanced tuni
```

```
ng and best practices, this highly readable book is the one-stop guide to MySQL develop
ment. It consists of four major sections: * MySQL stored programming fundamentals -- tu
torial, basic statements, SQL in stored programs, and error handling * Building MySQL s
tored programs -- transaction handling, built-in functions, stored functions, and trigg
ers * MySQL stored programs in applications -- using stored programs with PHP, Java, Pe
rl, Python, and .NET (C# and VB.NET) * Optimizing MySQL stored programs -- security, ba
sic and advanced SQL tuning, optimizing stored program code, and programming best pract
ices A companion web site contains many thousands of lines of code, that you can put to
 use immediately. Guy Harrison is Chief Architect of Database Solutions at Quest Softwa
re and a frequent speaker and writer on MySQL topics. Steven Feuerstein is the author o
f Oracle PL/SQL Programming, the classic reference for Oracle stored programming for mo
re than ten years. Both have decades of experience as database developers, and between
them they have authored a dozen books. | 44.99 | sqltuning   |            105 |
+-------------+------------------------------------+-----------------+-----------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
----------------------------------------+-------+-------------+-----------------+
25 rows in set (0.00 sec)
mysql>
```

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 looks like this:



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, for performance reasons, we only want to join on our primary key (or another key).

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 a
s 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>
```

You see the five rows from the Inventory and Products tables that have the same ProductCode. Your results might be in a different order, because we didn't ask the database to return the results in any particular order. Let's look a little closer.

OBSERVE:

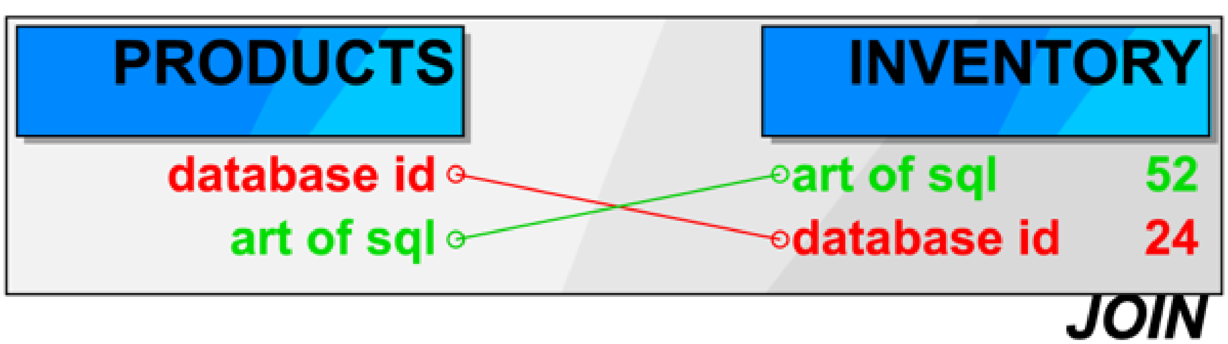
```
mysql> SELECT Products.ProductCode, Products.Title, Products.Price, I.QuantityInStock a
s Qty
    -> FROM Products
    -> JOIN Inventory as I on (Products.ProductCode = I.ProductCode);
```

Notice that we listed the ProductCode column as **Products.**ProductCode? Different tables might have columns with the same names, so 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 specify a shorter name for a table as a *table alias*. In our example we created an alias for Inventory called **I** by typing **Inventory 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 on by using the **on** keyword: **on (Products.ProductCode = I.ProductCode)**. Graphically, a join could look something like this:



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;
Bye
cold1:~$ curl -L http://courses.oreillyschool.com/dba1/downloads/products-1.sql
> products-1.sql
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
101  1219  101  1219    0     0   9718      0 --:--:-- --:--:-- --:--:-- 1190k
cold1:~$
```

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

Type the following at the Unix prompt:

```
cold1:~$ mysql -h sql -p -u username username < products-1.sql
Enter password:
cold1:~$
```

Again, if the import is successful, you see no results. This added a record to the Products table for relationaldb, "The Relational Database Dictionary."

Once you have that data imported, reconnect to MySQL. Be sure to set **sql_mode='traditional'** again.

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

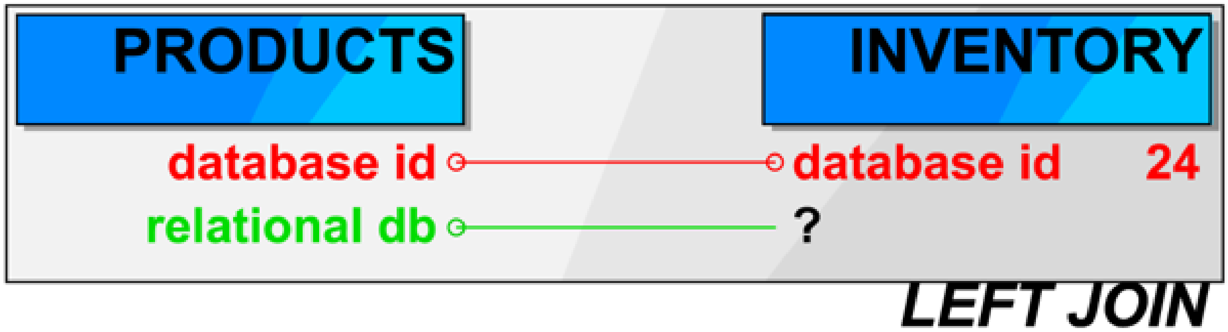
Type the following at the MySQL prompt:

```
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);
+-------------+------------------------------------+-------+-----+
| 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>
```

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

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 like this:



Let's try a left join.

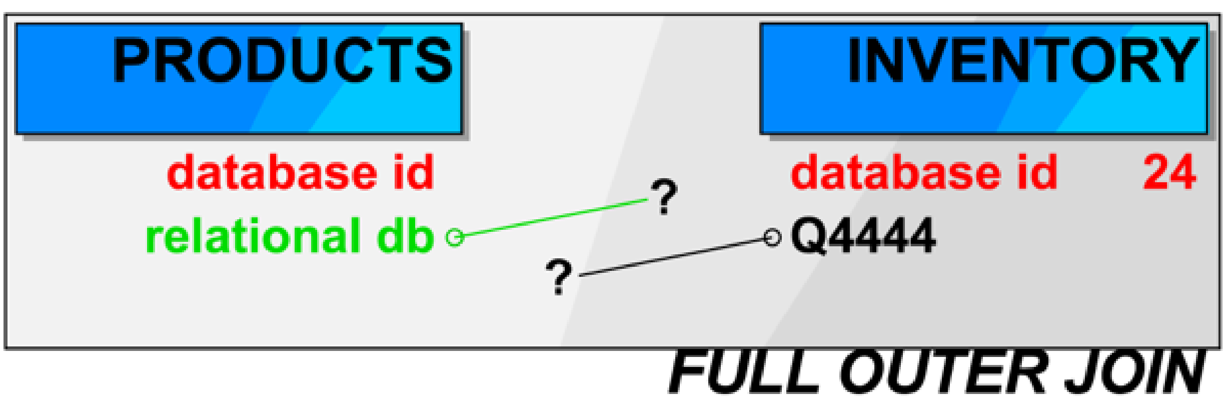
Type the following at the MySQL prompt:

```
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);
+--------------+-------------------------------------+-------+------+
| 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>
```

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.

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 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);
Query OK, 1 row affected (0.01 sec)

mysql>
```

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*. We'll discuss this in a future course.

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);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'FULL
OUTER JOIN Inventory as I on (P.ProductCode = I.ProductCode)' at line 3
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;
+-------------+-----------------+-------+-------+
| ProductCode | QuantityInStock | Title | Price |
+-------------+-----------------+-------+-------+
| Q4440       |              14 | NULL  |  NULL |
+-------------+-----------------+-------+-------+
1 row in set (0.00 sec)

mysql>
```

For this query we added 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.

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;
+--------------+-----------------+-------------------------------------+-------+
| 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)

mysql>
```

Take a closer look at the command and the results:

OBSERVE:

```
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;
+--------------+-----------------+-------------------------------------+-------+
| 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)
```

First you see our **previous query**, then the **UNION ALL** keyword, and then **our new query**.

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** are from the first query, and the **last row** 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

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 rows (products) rows in the Products table.

Type the following at the MySQL prompt:

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

mysql>
```

Note If you've changed the Products table from the last lesson, you'll see a different number.

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;
+-----------------+------------------+
| 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 want 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;
+-----------------+--------------+--------------+
| 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>
```

OBSERVE:

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

In this query, we 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.

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

Type the following at the MySQL prompt:

```
mysql> SELECT Category, AVG(Price) as AveragePrice, MAX(Price) as MaximumPrice
    -> FROM Products;
+-----------------+--------------+--------------+
| Category        | AveragePrice | MaximumPrice |
+-----------------+--------------+--------------+
| Database Theory |    34.143333 |        44.99 |
+-----------------+--------------+--------------+
1 row in set (0.00 sec)

mysql>
```

You only see one result row.

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';
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Now when you run a query without a GROUP BY you will see an error (assuming your sql_mode is still set to 'ONLY_FULL_GROUP_BY').

Type the following at the MySQL prompt:

```
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 col
umns is illegal if there is no GROUP BY clause
```

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);
+--------------+-------+------+
| 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)

mysql>
```

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.QuantityInS
tock 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)

mysql>
```

OBSERVE:

```
P.Price * I.QuantityInStock as DollarValue
```

The result includes has a new **DollarValue** column which is the **product of Price and Quantity**.

It might seem 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.Quantit
yInStock) as DollarValue
    -> 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 col
umns is illegal if there is no GROUP BY clause
mysql>
```

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 a correct answer, so you get the nonsensical result.

To correct this problem, 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);
+------------------+
| TotalDollarValue |
+------------------+
|          7718.86 |
+------------------+
1 row in set (0.00 sec)

mysql>
```

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

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);
+--------------+-------------------------------------+------+--------------------+
| 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)
```

OBSERVE:

```
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** and **results**. **Comparisons** are checked in order. As soon as a TRUE comparison is found, the corresponding **result** is returned.

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);
+--------------+-----------------------------------+-----+---------------------+
| 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)
```

OBSERVE:

```
mysql> SELECT P.ProductCode, P.Title, IFNULL(I.QuantityInStock, 0) as Qty,
    -> CASE WHEN I.QuantityInStock IS NULL THEN 'Place Initial Order'
```

We used the **IFNULL** conditional to check the QuantityInStock column. NULLs in that column are replaced by zeros (0). We also added a comparison to the CASE statement—**I.QuantityInStock IS NULL**—to display the action "Place Initial Order."

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!

Subqueries and Views

In the past few lessons, we've learned more complex ways to query our databases. In this lesson we'll examine one more way to 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);
+--------------+------+-------------------------------------+-------+
| 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)

mysql>
```

You could scan all of the returned items to find the one with the largest quantity. But it would be even better to return a single row. 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;
+-------------+------+-------------------------------------+-------+
| 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)

mysql>
```

OBSERVE:

```
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)."

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 single product with the maximum quantity in Inventory.

In order to return this information, let's try to restrict our query with a WHERE clause:

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);
ERROR 1111 (HY000): Invalid use of group function

mysql>
```

This query isn't quite right either. Fear not, there is hope! We can correct this error if we use a *subquery*.

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 subquery.

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 subquery is correct. First, find the maximum QuantityInStock in Inventory.

Type the following at the MySQL prompt:

```
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 might return the desired result now, on a live system it won't work reliably. After all, the QuantityInStock might change before we get a chance to run the query. Instead let's use a subquery. 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);
+-------------+------+------------+-------+
| 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 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);
+--------------+-------------------------------------+-------+------+
| 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>
```

(Of course, if the data in your tables has changed, your results will be different.)

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);
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;
+--------------+------------------------------------+-------+------+
| 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;
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);
Query OK, 0 rows affected (0.00 sec)

mysql>
```

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

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*.

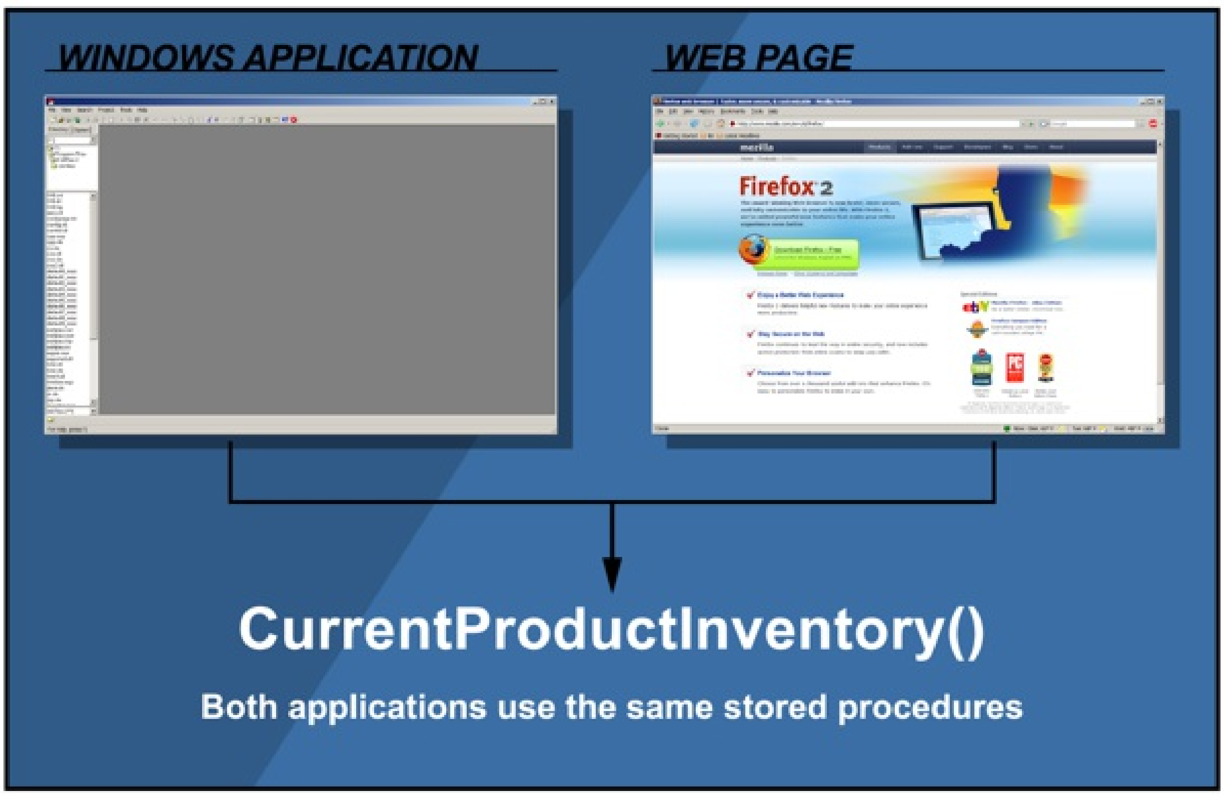
Using Stored Procedures

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 1998, 1988, or 1978. But the technology has changed pretty dramatically—inventory tracking was done on paper in 1978, then maybe on a green-screen computer in 1988, and then in a Windows application in 1998, but today, inventory tracking is likely to be done using a web browser *and* a Windows application . Stored procedures created back in 1988 would still function today, twenty-plus 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 a 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 **delimiter** keyword.

Type the following at the MySQL prompt:

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

This command doesn't return anything, but you'll notice a difference after you run a query.

Type the following at the MySQL prompt:

```
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 results.

Type the following at the MySQL prompt:

```
    -> //
+--------------+------------------------------------+
| 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.

INTERACTIVE SESSION:

```
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>
```

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

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;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql>
```

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;
    -> //
```

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.

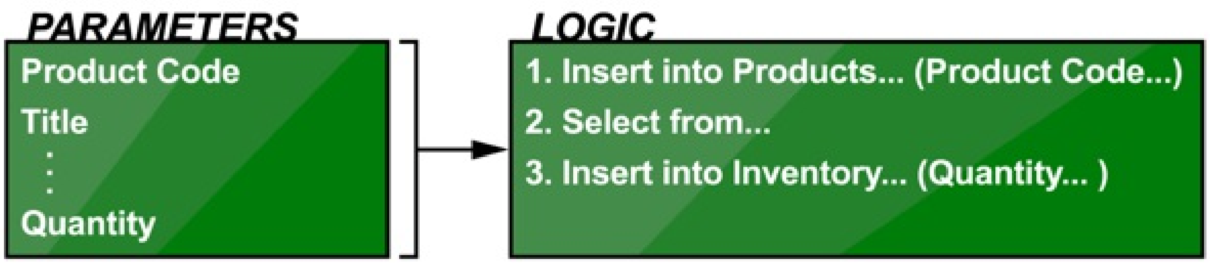
Type the following at the MySQL prompt:

```
mysql> CALL CurrentProductInventory ();
    -> //
+--------------+------------------------------------+-------+------+
| 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>
```

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.

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.



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 functions, as well as MySQL
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. Here's a skeleton 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;
//
```

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;
    -> //
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 ();
    -> //
ERROR 1318 (42000): Incorrect number of arguments for PROCEDURE certjosh.CreateP
roduct; expected 6, got 0

mysql>
```

This procedure has six parameters, and requires all six.

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*.

Let's call the procedure with parameters this time. The example below has one parameter per line to facilitate our discussion, but you can enter all parameters on one line if you like.

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 admin
istration.",
    ->   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 ();
    -> //
+--------------+-----------------------------------+-------+------+
| 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 no 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 one 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;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql>
```

OBSERVE:

```
...
    ->    DECLARE product_code varchar(20);
    ->
    ->    SELECT ProductCode into product_code
    ->    FROM Products
    ->    WHERE Title = product_title;
    ->
    ->    SELECT product_code as ProductCode;
...
```

The **DECLARE** statement in our procedure tells MySQL that we are using a variable named **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 look up the title **SQL Hacks**.

Type the following at the MySQL prompt:

```
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.

Remove our old procedure before we continue.

Type the following at the MySQL prompt:

```
mysql> DROP PROCEDURE IF EXISTS GetProductCode;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Now, create the 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;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql>
```

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';
    -> //
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);
    -> //
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';
    -> //
+-------------+-----------------+
| ProductCode | QuantityInStock |
+-------------+-----------------+
| sqlhks      |              99 |
+-------------+-----------------+
1 row in set (0.00 sec)

mysql>
```

Sure enough, it worked perfectly!

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, and then think about how you could handle this problem.

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

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. 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:

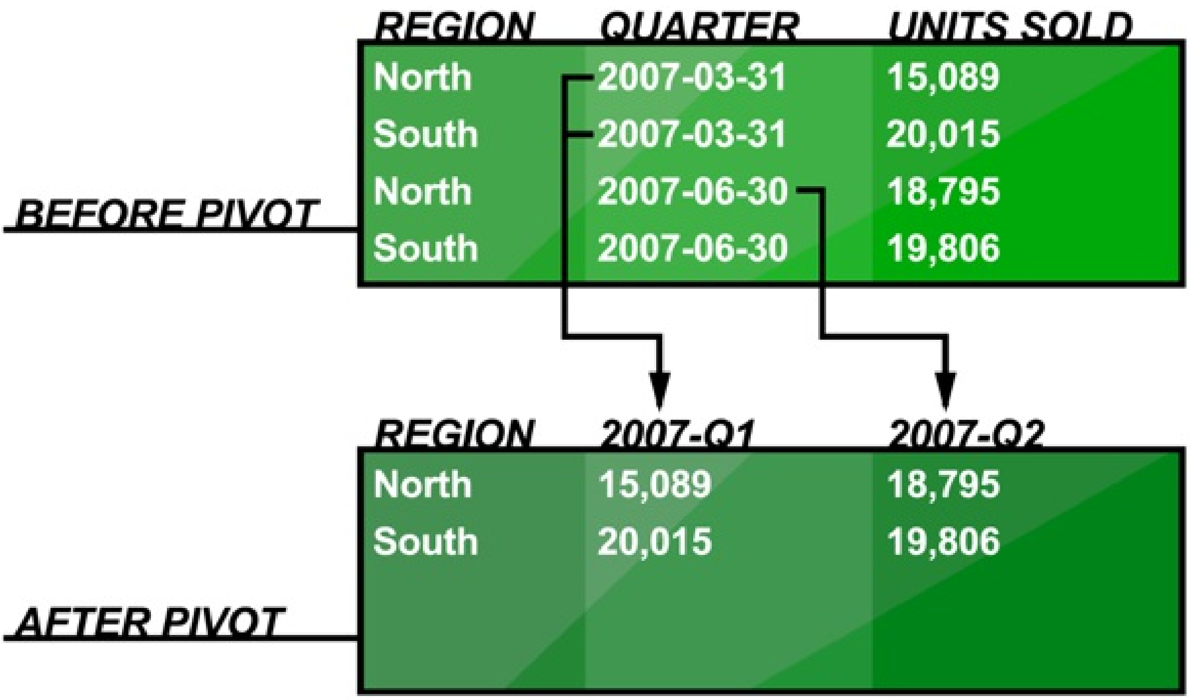
Region	Quarter	Units Sold
North	2007-03-31	15,089
North	2007-06-30	18,795
North	2007-09-30	19,065
North	2007-12-31	19,987
South	2007-03-31	20,015
South	2007-06-30	19,806
South	2007-09-30	21,053
South	2007-12-31	23,068

Showing the data in this way is useful, especially to programmers, but 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 dialects. While MySQL doesn't currently have that keyword, 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.

Note If you haven't closed the terminal session since the last lesson, set your delimiter back from "//" to ";".

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;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

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

Tip Remember you can use the Up arrow key to repeat a command in mysql and then edit the command. For example, after you enter the first command below, press the up arrow key and then edit the Quarter date and the UnitsSold integer to create the next command.

Type the following at the MySQL prompt:

```
mysql> INSERT INTO SalesAnalysis values ('NORTH', '2007-03-31',15089);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-06-30',18795);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-09-30',19065);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('NORTH', '2007-12-31',19987);

Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-03-31',20015);
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-06-30',19806);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-09-30',21053);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO SalesAnalysis VALUES ('SOUTH', '2007-12-31',23068);
Query OK, 0 rows affected (0.01 sec)

mysql>
```

Check that you entered everything correctly:

Type the following at the MySQL prompt:

```
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 named **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;
+--------+---------+---------+---------+---------+
| 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>
```

Let's look a little closer.

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;
```

Here, 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-Q2'** 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 the useful values.

How can we collapse this into meaningful data? 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:

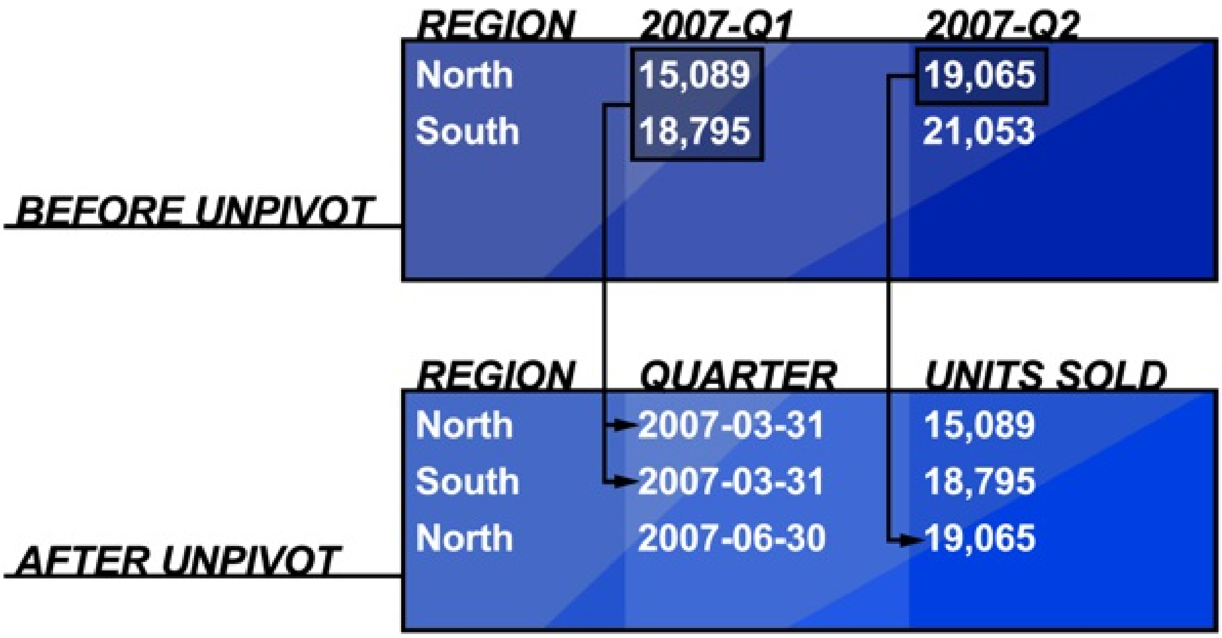
```
mysql> SELECT Region,
    -> SUM(CASE WHEN Quarter='2007-03-31' THEN UnitsSold ELSE 0 END) AS '2007-Q1',
    -> SUM(CASE WHEN Quarter='2007-06-30' THEN UnitsSold ELSE 0 END) AS '2007-Q2',
    -> SUM(CASE WHEN Quarter='2007-09-30' THEN UnitsSold ELSE 0 END) AS '2007-Q3',
    -> SUM(CASE WHEN Quarter='2007-12-31'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

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



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

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'.

WARNING Be sure to type the correct character for the column **2007-Q1**. MySQL requires column names to be quoted with a backwards apostrophe (`) 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 key is usually located near the **Esc** key on your keyboard.

Type the following at the MySQL prompt:

```
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>
```

OBSERVE:

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

Here, we added a new column named **Quarter** and renamed the column **`2007-Q1`** "UnitsSold".

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;
+--------+------------+-----------+
| 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>
```

This is exactly the same data that is stored in the SalesAnalysis table!

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

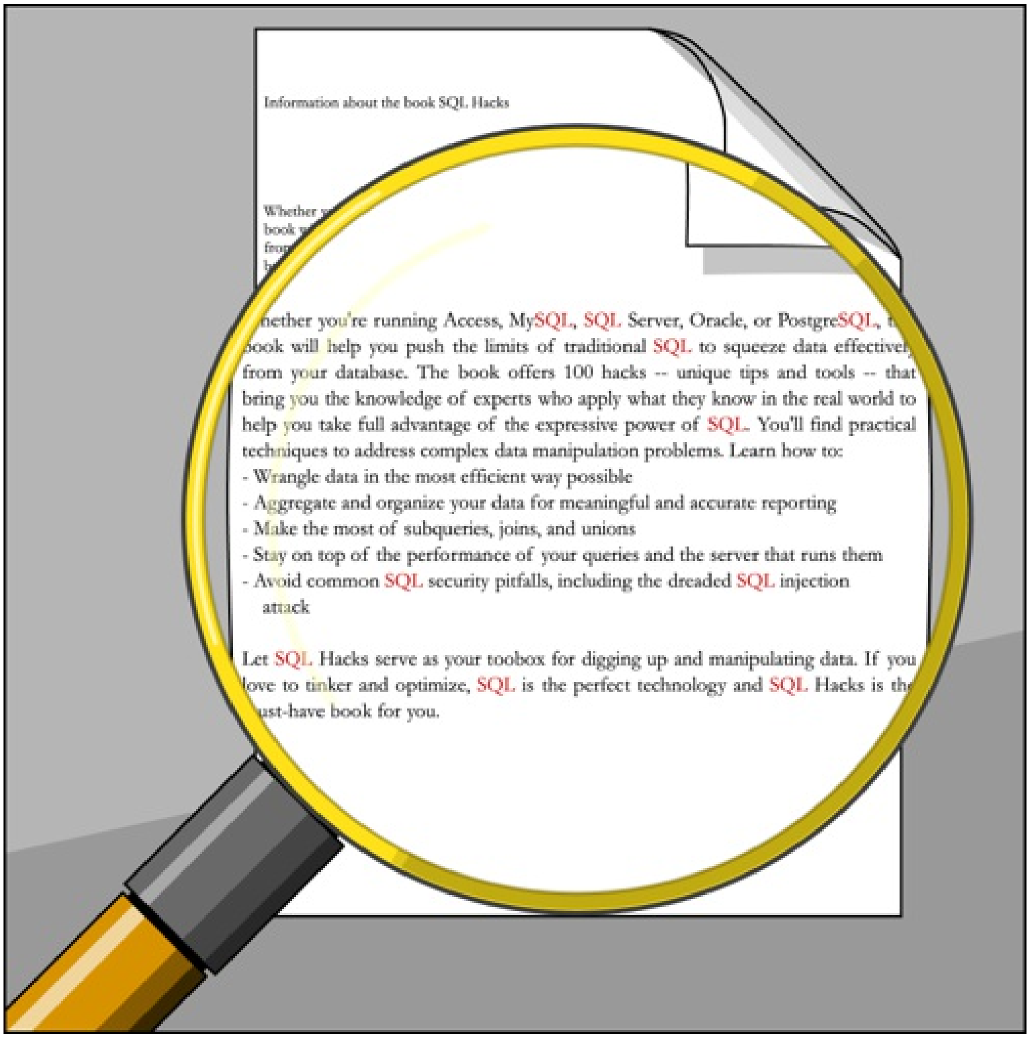
You've learned yet another way to query the data in your database. MySQL offers additional ways to facilitate your database applications—user defined types, functions, and extensions are all ways to expand the ways you store and interact with your data. See you in the next lesson!

Full Text

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 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.

The version of MySQL we currently use has some restrictions on full-text indexes. The most important restriction is that full-text indexes can only be used with the *MyISAM* engine. Before we can use this index on our Products table, we'll have to make sure the table type is MyISAM. MySQL version 5.6.4 and later can use full-text indexes on InnoDB tables too.

Type the following at the MySQL prompt:

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

mysql>
```

(It's OK if you get a warning here.)

Before we start experimenting with full-text indexes, we'll add some products with long descriptions. First, delete existing rows from the Products and Inventory tables.

Type the following at the MySQL prompt:

```
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.

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

```
cold1:~$ curl http://courses.oreillyschool.com/dba1/downloads/products-10.sql > product
s-10.sql
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 11829  100 11829    0     0  94214      0 --:--:-- --:--:-- --:--:-- 5775k
cold1:~$
```

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:

```
cold1:~$ mysql -h sql -p -u username username < products-10.sql
Enter password:
cold1:~$
```

If the import is successful, you see no results.

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, but our users will primarily search on title and description.

Switch back to the first terminal session.

Type the following at the MySQL prompt:

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

mysql>
```

OBSERVE:

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

We added **FULLTEXT** to the **Title** and **Description** columns.

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');
+--------------+-------------------------------------+-------+
| 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>
```

Take a peek at the descriptions for these books—all have some mention of the phrase "relational database."

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%datab
ases%';
+--------------+-------------------------------------+-------+
| 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, you are correct. Full-text indexes take the language being indexed into account 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 these 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.)

For a list of default stop words, check out MySQL's web site.

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');
+--------------+-------------------------------------+-------+
| 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>
```

This time five books matched the criteria.

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');
+--------------+-------------------------------------+-------+-------------------+
| ProductCode  | Title                               | Price | Relevance         |
+--------------+-------------------------------------+-------+-------------------+
| databaseid   | Database in Depth                   | 29.95 |  1.31360828876495 |
| relationaldb | The Relational Database Dictionary  | 14.99 | 0.812412083148956 |
| artofsql     | The Art of SQL                      | 44.99 | 0.108494639396667 |
+--------------+-------------------------------------+-------+-------------------+
3 rows in set (0.00 sec)
mysql>
```

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.

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.

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 there!

Information About the Database

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*—data about data—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 useful because we can use our standard query tools to access 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, but you are able to connect to it. You can query any table in the INFORMATION_SCHEMA, but you can't 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** enables 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';
+-------------------+------------+--------+
| 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>
```

Some of the rows have been omitted for clarity, and you may see a different number of rows.

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='username' 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)
```

We have four tables that do not support full-text indexing, and our boss has his answer in about one second!

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';
+-------------+-----------+--------------------------+
| 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'll use the *length* function—it returns an integer count of a string's length.

Note Be sure to include a backslash (\) before the apostrophe (') in the title. The backslash "escapes" the apostrophe so MySQL won't think it's the end of the title.

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 About the Internet');
+-------------+-----------+--------------------------+
| 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. Your boss 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 WHERE table_schema='username';
+------------------+--------------+---------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------+
| table_name       | is_updatable | view_definition








                                                           |
+------------------+--------------+---------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------+
| ProductInventory | NO           | /* ALGORITHM=UNDEFINED */ select `P`.`Produc
tCode` AS `ProductCode`,`P`.`Title` AS `Title`,`P`.`Price` AS `Price`,`I`.`Quant
ityInStock` AS `Qty` from (`certjosh`.`Products` `P` left join `certjosh`.`Inven
tory` `I` on((`P`.`ProductCode` = `I`.`ProductCode`)))





                                                           |
| SalesDataPivot   | NO           | /* ALGORITHM=UNDEFINED */ select `certjosh`.
`SalesAnalysis`.`Region` AS `Region`,sum((case when (`certjosh`.`SalesAnalysis`.
`Quarter` = _latin1'2007-03-31') then `certjosh`.`SalesAnalysis`.`UnitsSold` els
e 0 end)) AS `2007-Q1`,sum((case when (`certjosh`.`SalesAnalysis`.`Quarter` = _l
atin1'2007-06-30') then `certjosh`.`SalesAnalysis`.`UnitsSold` else 0 end)) AS `
2007-Q2`,sum((case when (`certjosh`.`SalesAnalysis`.`Quarter` = _latin1'2007-09-
30') then `certjosh`.`SalesAnalysis`.`UnitsSold` else 0 end)) AS `2007-Q3`,sum((
case when (`certjosh`.`SalesAnalysis`.`Quarter` = _latin1'2007-12-31') then `cer
tjosh`.`SalesAnalysis`.`UnitsSold` else 0 end)) AS `2007-Q4` from `certjosh`.`Sa
lesAnalysis` group by `certjosh`.`SalesAnalysis`.`Region` |
+------------------+--------------+---------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------+
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
    -> WHERE table_schema='username';
+------------------+--------------+--------------------------------------------
-------+
| table_name       | is_updatable | definition
       |
+------------------+--------------+--------------------------------------------
-------+
| ProductInventory | YES          | S `ProductCode`,`P`.`Title` AS `Title`,`P`.`
Price` |
| SalesDataPivot   | NO           | lysis`.`Region` AS `Region`,sum((case when (
`smill |
+------------------+--------------+--------------------------------------------
-------+
2 rows in set (0.99 sec)

mysql>
```

That's much easier on the eyes!

You give the results to your boss, who is still very impressed with your work, and promises to have a talk with you soon about your future with the company.

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
*************************** 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>
```

Your results might be slightly different, but they should look similar.

These are exactly the results we want. Our boss 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 provide that information.

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;
+-------------------------+--------------+----------------------+
| routine_name            | routine_type | Uses_QuantityInStock |
+-------------------------+--------------+----------------------+
| CreateProduct           | PROCEDURE    |                    1 |
| CurrentProductInventory | PROCEDURE    |                    1 |
+-------------------------+--------------+----------------------+
2 rows in set (0.00 sec)

mysql>
```

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

That's 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

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 Specification

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 a 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 this:

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 finish hand in this SQL file.

GOOD LUCK!