Database Processing

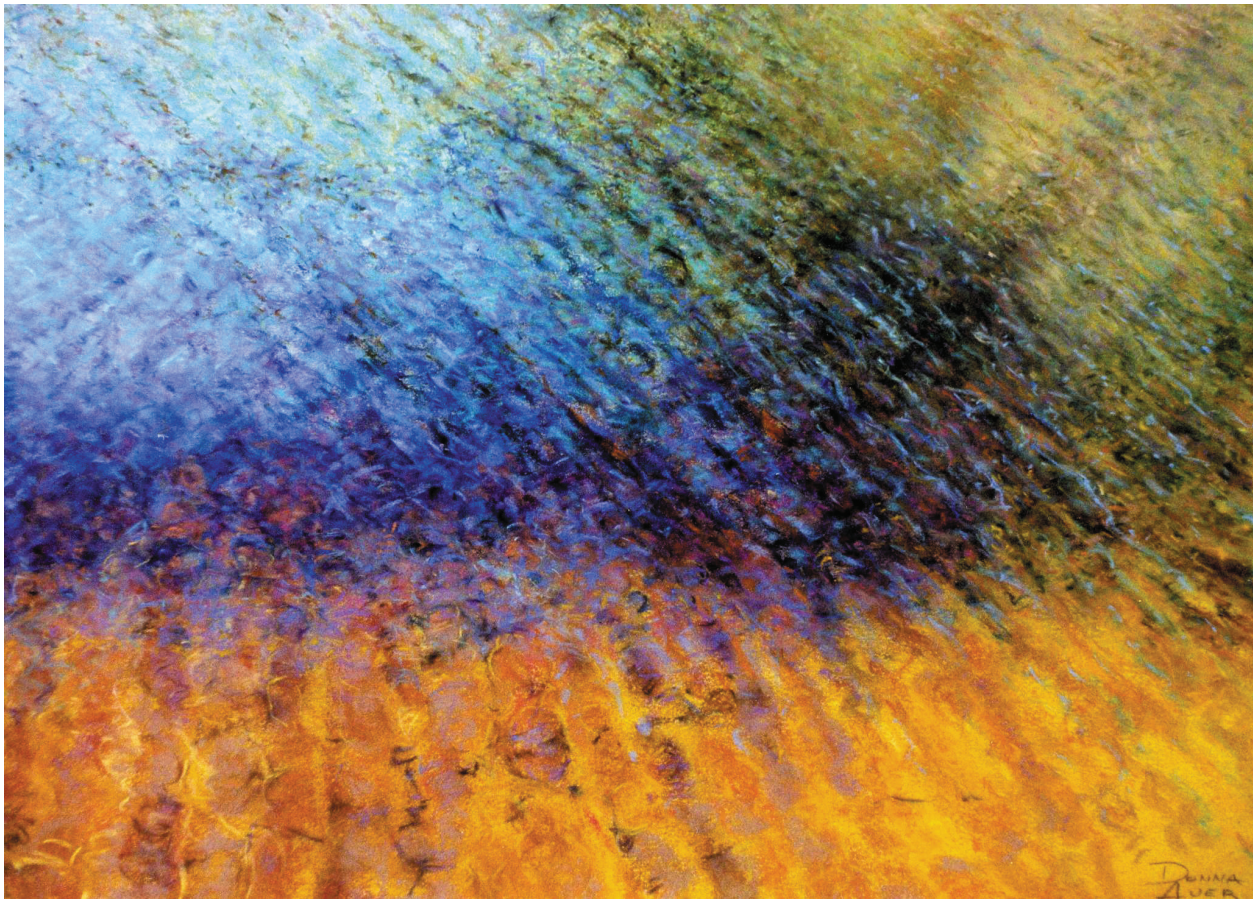
Fundamentals, Design, and Implementation

14th Edition

David M. Kroenke • David J. Auer

Online Chapter 10C

Managing Databases with MySQL 5.6



Vice President, Business Publishing: Donna Battista
Editor in Chief: Stephanie Wall
Acquisitions Editor: Nicole Sam
Program Manager Team Lead: Ashley Santora
Program Manager: Denise Weiss
Editorial Assistant: Olivia Vignone
Vice President, Product Marketing: Maggie Moylan
Director of Marketing, Digital Services and Products: Jeanette Koskinas
Executive Product Marketing Manager: Anne Fahlgren
Field Marketing Manager: Lenny Ann Raper
Senior Strategic Marketing Manager: Erin Gardner
Product Marketing Assistant: Jessica Quazza
Project Manager Team Lead: Jeff Holcomb
Project Manager: Ilene Kahn
Operations Specialist: Diane Peirano
Senior Art Director: Janet Slowik
Text Designer: Integra Software Services Pvt. Ltd.
Cover Designer: Integra Software Services Pvt. Ltd.
Cover Art: Donna Auer
Vice President, Director of Digital Strategy & Assessment: Paul Gentile
Manager of Learning Applications: Paul Deluca
Digital Editor: Brian Surette
Digital Studio Manager: Diane Lombardo
Digital Studio Project Manager: Robin Lazrus
Digital Studio Project Manager: Alana Coles
Digital Studio Project Manager: Monique Lawrence
Digital Studio Project Manager: Regina DaSilva
Full-Service Project Management and Composition: Integra Software Services Pvt. Ltd.
Printer/Binder: RRD Willard
Cover Printer: Phoenix Color/Hagerstown
Text Font: 10/12 Mentor Std Light

Library of Congress Cataloging-in-Publication Data

Kroenke, David M.
Database processing: fundamentals, design, and implementation/David M. Kroenke, David J. Auer.–Fourteenth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-387670-3 (student edition)–ISBN 978-0-13-387676-5
(instructor's review copy)
1. Database management. I. Auer, David J. II. Title.
QA76.9.D3K76 2016
005.74–dc23

2015005632

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-387670-5
ISBN 13: 978-0-13-387670-3

10C

Managing Databases with MySQL 5.6

Chapter Objectives

- To install MySQL 5.6 and create a database
- To use the MySQL Workbench graphical utility
- To submit both SQL DDL and DML via the MySQL Workbench
- To import Microsoft Excel worksheet data into a database table
- To understand the use of SQL/Persistent Stored Modules (SQL/PSM) in MySQL SQL
- To understand the purpose and role of user-defined functions and learn how to create simple user-defined functions
- To understand the purpose and role of stored procedures and learn how to create simple stored procedures
- To understand the purpose and role of triggers and learn how to create simple triggers
- To understand how MySQL 5.6 implements indexes, concurrency control and cursors
- To understand how MySQL 5.6 implements server and database security
- To understand the fundamental features of MySQL 5.6 backup and recovery facilities

This chapter describes the basic features and functions of MySQL 5.6. The discussion uses the View Ridge Gallery database from Chapter 7, and it parallels the discussion of the database administration tasks in Chapter 9. The presentation is similar in scope and orientation to that of Microsoft SQL Server 2014 in Chapter 10A and to Oracle's Oracle Database in Chapter 10B.

MySQL 5.6 is a large and complicated product. In this one chapter, we will only be able to scratch the surface. Your goal should be to learn sufficient basics so you can continue learning on your own or in other classes. Most of the topics and techniques discussed here also apply to the earlier MySQL 5.1 and MySQL 5.5.

The MySQL 5.6 DBMS

MySQL is an enterprise-class DBMS that has been around for many years. In November 2005, MySQL 5.0 was released, followed by MySQL 5.1. As of this writing, MySQL 5.6 is the generally available (GA) release, and a release candidate pre-GA version MySQL 5.7 is available for testing.

In February 2008, Sun Microsystems acquired MySQL AB, the company that created and owned MySQL. In turn, Oracle Corporation acquired Sun Microsystems—the deal was finalized on January 27, 2010, after European Commission approval on January 21, 2010—see *www.sun.com/third-party/global/oracle/*. Thus, Oracle Corporation now owns MySQL in addition to its flagship Oracle Database product. Although MySQL does not have as many features as Microsoft SQL Server 2014 or Oracle Database 12*c*, it has become widely used and very popular as a DBMS supporting Web sites running the Apache Web server. The MySQL Community Edition and MySQL Workbench CE graphical user interface (GUI) utility are free.

Under MySQL AB and Sun Microsystems, MySQL was available in three versions: MySQL Enterprise, MySQL Community Server, and MySQL Cluster:

- **MySQL Enterprise.** MySQL Enterprise (*www.mysql.com/products/enterprise*) is a subscription service that includes the MySQL Workbench Standard Edition, MySQL Enterprise Server, MySQL Production Support, and some utilities that are not available as open source downloads (including the MySQL Enterprise Monitor, MySQL Enterprise Backup, and MySQL Query Analyzer). It includes both the ISAM and InnoDB storage engines.
- **MySQL Community Edition.** Previously called (and still downloaded as) MySQL Community Server, MySQL Community Edition (*www.mysql.com/products/community/*) is the free version of MySQL available for download from *www.mysql.com/downloads/mysql/*, where it is still (as of this writing) referred to as MySQL Community Server. It includes both the ISAM and InnoDB storage engines (see the discussion of MySQL storage engines later in this chapter).
- **MySQL Cluster.** MySQL Cluster (*www.mysql.com/products/cluster/*) is a specialized version of MySQL designed for distributed databases (see Chapter 9 for a discussion of distributed databases, which run a set of multiple servers referred to as a *cluster of servers*. MySQL Cluster has its own, unique version numbers, and MySQL Cluster 7.1 is the current generally available edition. MySQL Cluster is open source and freely available for download at *www.mysql.com/downloads/cluster/*.

Oracle has introduced some new editions of MySQL:

- **MySQL Standard Edition.** MySQL Standard Edition (*www.mysql.com/products/standard*), like MySQL Enterprise, is a subscription service, but it does not include the MySQL Enterprise Server utilities, MySQL Production Support, and some utilities that are not available as open source downloads (MySQL Workbench Standard Edition, MySQL Enterprise Monitor, MySQL Enterprise Backup, and MySQL Query Analyzer). It includes both the ISAM and InnoDB storage engines.
- **MySQL Classic Edition.** A version of MySQL intended for independent software vendors (ISVs), original equipment manufacturers (OEMs), and value-added resellers (VARs) to use as an embedded database for read-intensive applications. It uses only the MyISAM storage engine (the lack of the InnoDB storage engine is the main distinction between MySQL Standard and MySQL Classic), and it is available only for this purpose. See *www.mysql.com/products/classic*.
- **MySQL Cluster Carrier Grade Edition (CGE).** MySQL Cluster CGE (*www.mysql.com/products/cluster/*) is the commercial version of MySQL.
- **MySQL Embedded (OEM/ISV).** This is actually an option for the use of MySQL Enterprise, MySQL Standard, MySQL Classic, or MySQL Cluster CGE as an embedded database. See *www.mysql.com/oem/*.

More than one release of MySQL is often available, but as this text is being written only MySQL 5.6 is a GA (generally available) release, and MySQL 5.7 is available as a release candidate. (Alpha releases are early test releases, followed by milestone [commonly known as beta] releases, and then a release candidate [RC] version.) Community server versions and development releases are available for download at *http://dev.mysql.com/downloads/*.

BY THE WAY

Oracle Corporation does not publish a time line for expected release dates of new versions of MySQL. We have been anticipating the release of MySQL 5.7 for some time, but it has remained in milestone development releases since April of 2013. At the time we are working on this material (April 2015), MySQL 5.7 has just been released as a release candidate version (MySQL 5.7.7 on April 8, 2015), and it is likely that MySQL 5.7 will soon be available as GA software.

We do not include material about software versions that have not been officially released in this book, and so we do not specifically cover MySQL 5.7. However, we have tested the material in this book with development versions of MySQL 5.7, and all the material in this book about MySQL 5.6 is also directly applicable to MySQL 5.7.

Installing MySQL 5.6

Regardless of which release of MySQL you are going to use, you should download and install it now. Versions are available for various operating systems, so download the correct version for your computer's operating system. While there are many MySQL components available, we will install a minimum configuration consisting of:

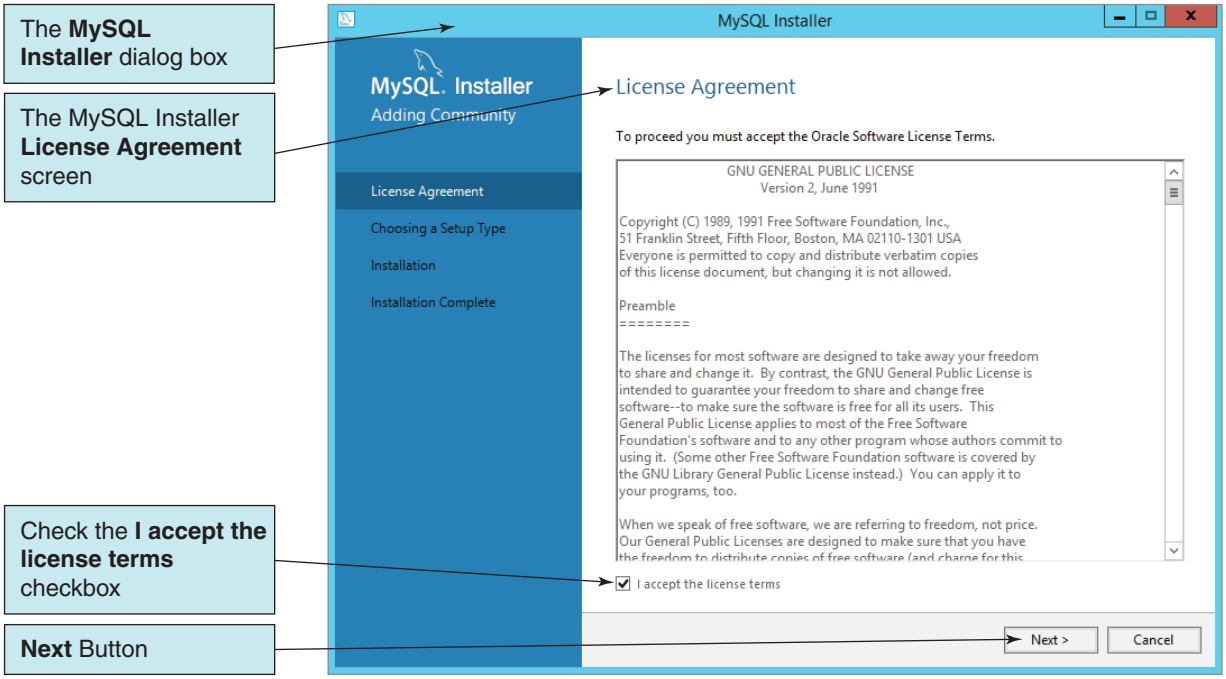
1. MySQL Community Server 5.6
2. MySQL Workbench 6.2.4
3. MySQL Connector/ODBC 5.3.4
4. MySQL Documentation 5.6 (available for Windows, otherwise use online version at *http://dev.mysql.com/doc/refman/5.6/en/*

If you are using a Microsoft Windows operating system, we recommend that you download the **MySQL Installer for Windows**, which packages current versions of MySQL 5.6 Community Server, the MySQL Workbench, several MySQL connectors (including the ODBC connector needed for the Web application work in Chapter 11), other utilities, samples, examples, and documentation together with an installation utility that controls which products are actually installed. The MySQL Installer for Windows can be downloaded from *http://dev.mysql.com/downloads/*. There is a separate version of the MySQL Installer for Windows for each major version of MySQL, and the version we are using here is for MySQL 5.6. Before running the MySQL Installer for Window, you need to:

1. Install the **.Net Framework 3.5**. This can be done in **Control Panel**. Select **Programs | Programs and Features | Turn Windows features on or off** to launch the **Add Roles and Features Wizard** to install .NET Framework 3.5 in Features (interestingly, .NET Framework 4.5 is installed by default, but .NET Framework 3.5 is not).
2. Download and install the **Microsoft Visual C++ Redistributable Package for Visual Studio 2013** (32-bit or 64-bit version depending upon your operating system) from *http://www.microsoft.com/en-us/download/details.aspx?id=40784*.

Non-Windows OS versions will require that you download and install each of the needed components separately. Regardless of whether you are installing a Windows OS or non-Windows OS version of MySQL, be sure to download and configure the set of MySQL components listed above.

We will step through the installation process of MySQL 5.6 on a Windows operating system using the MySQL Installer for Windows.



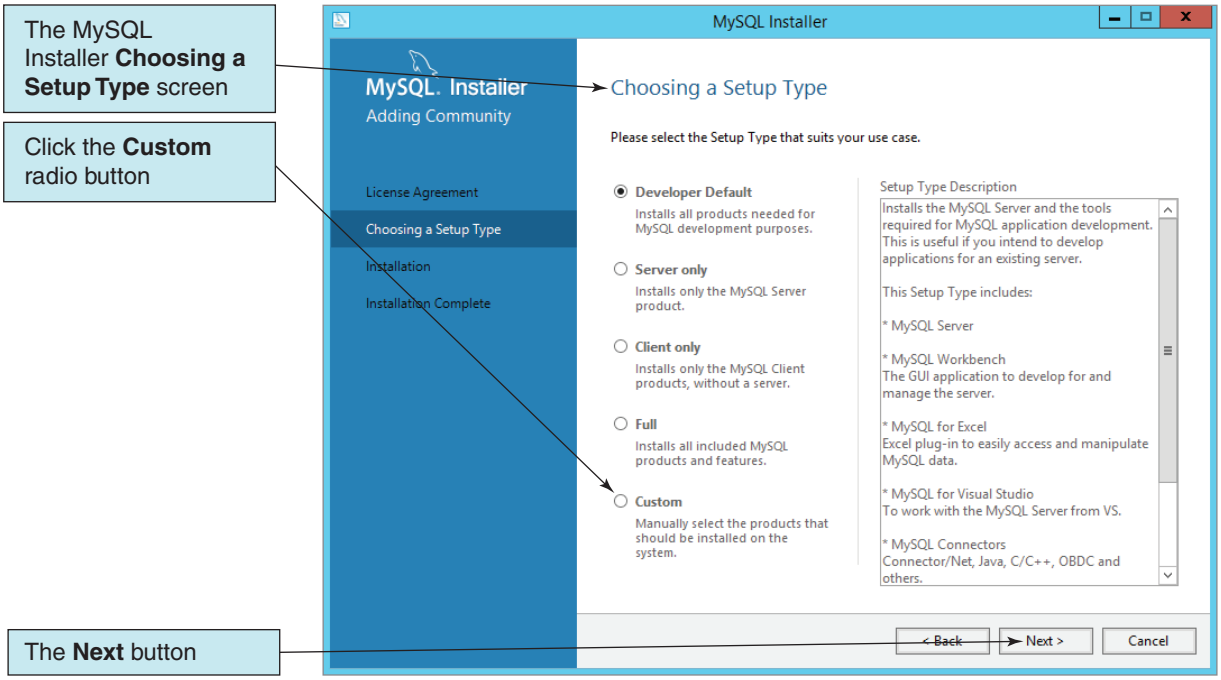
(a) The MySQL Installer License Agreement Screen

FIGURE 10C-1(a)
MySQL 5.6 Installation and Configuration

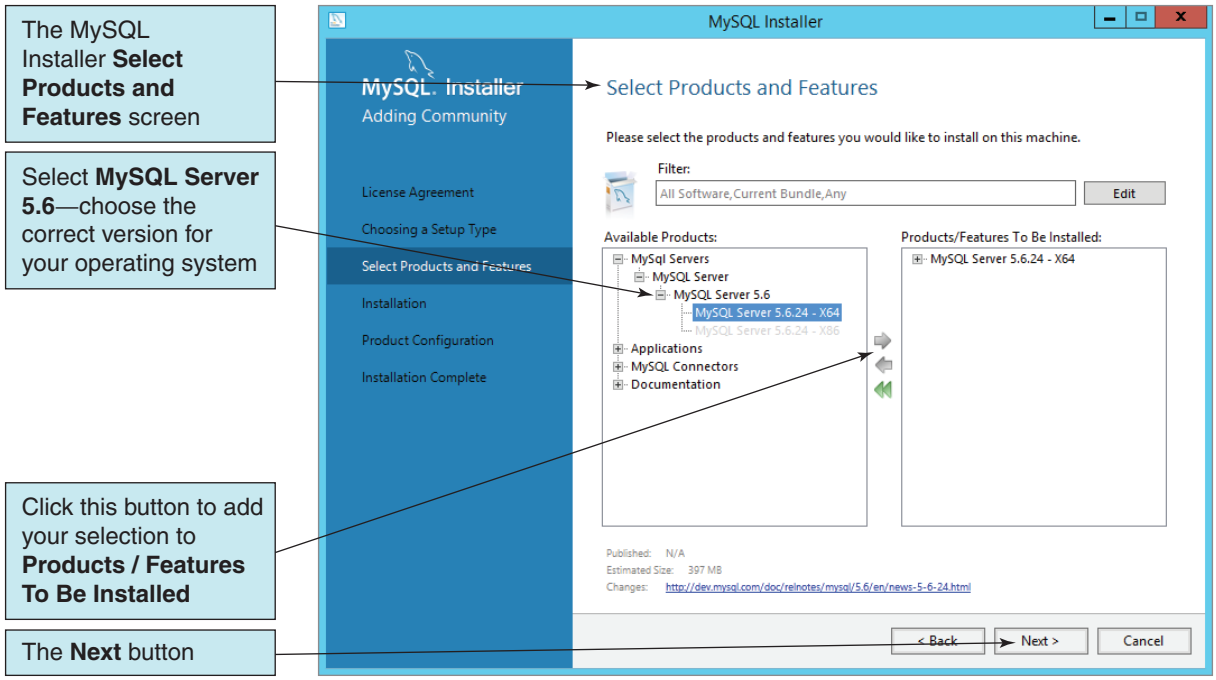
Installing MySQL Community Server 5.6

1. To start the actual installation process, open the **MySQL Installer utility** from the Windows Apps window (in Windows 8.1 and Windows Server 2012 R2) or the Windows menu (in Windows 7).
2. The **MySQL Installer** dialog box opens, and the *License Agreement* screen is displayed, as shown in Figure 10C-1(a). Check the **I accept the license terms** check box, and then click the **Next** button.
3. The *Choosing A Setup Type* screen is displayed, as shown in Figure 10C-1(b). Because we want to install only a minimum set of MySQL components, we will *not* use the

FIGURE 10C-1(b)
MySQL 5.6 Installation and Configuration



(b) The Choosing a Setup Type Screen



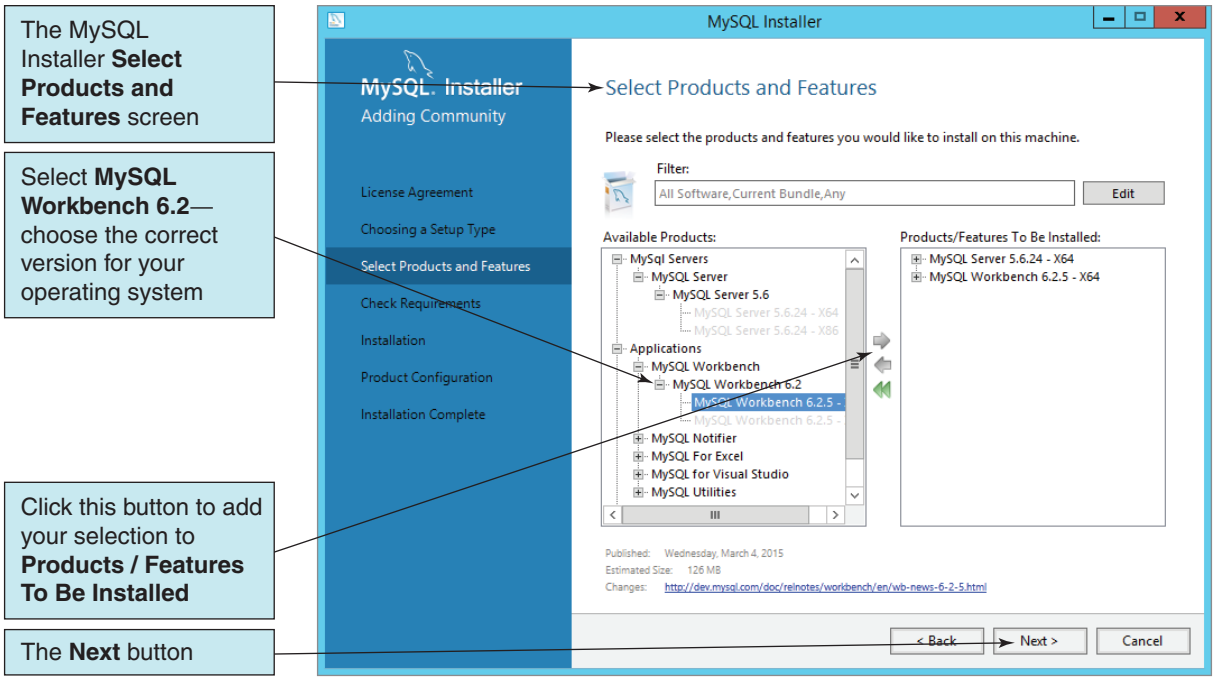
(c) The Select Products and Features Screen - MySQL Servers

FIGURE 10C-1(c)

MySQL 5.6 Installation and Configuration

Developer Default setup type. Click the **Custom** radio button to select the *Custom* setup, and click the **Next** button.

4. The *Select Products and Features* screen is displayed, as shown in Figure 10C-1(c). The first selection will be MySQL Community Server 5.6 itself. Expand the **MySql Servers** option as shown in the figure. Select the version of **MySQL Server 5.6** that matches your operating system (we are using a 64-bit version of Windows Server 2012 R2, so we have chosen the X64 version), and then click the **right-facing arrow** button to add MySQL Server to the *Products/Features To Be Installed* list.
5. Staying on the *Select Products and Features* screen, expand the **Applications** options as shown in Figure 10C-1(d). Select the version of **MySQL Workench 6.2** that matches your operating system (we are using a 64-bit version of Windows Server 2012 R2, so we have chosen the X64 version), and then click the **right-facing arrow** button to add MySQL Workbench to the *Products/Features To Be Installed* list.
   - **NOTE:** Although we will not install it, the **MySQL Notifier** utility is useful, and you may want to install it on your computer. It is available as an Applications option.
   - **NOTE:** If you are installing MySQL 5.6 on a computer that has Microsoft Excel 2013 installed on it, then you should install **MySQL for Excel**. We will use it later in this chapter to import Microsoft Excel data into a MySQL database.
6. Staying on the *Select Products and Features* screen, expand the MySQL Connectors options as shown in Figure 10C-1(e). Select the version of **Connector/ODBC 5.3** that matches your operating system (we are using a 64-bit version of Windows Server 2012 R2, so we have chosen the X64 version), and then click the **right-facing arrow** button to add Connector/ODBC to the *Products/Features To Be Installed* list.
7. Staying on the *Select Products and Features* screen, expand the **Documentation** options as shown in Figure 10C-1(f). Select the **MySQL Documentation 5.6** option (at this time there is only a 32-bit version available–it will also run on a 64-bit operating system), and then click the **right-facing arrow** button to add MySQL Documentation to the *Products/Features To Be Installed* list.

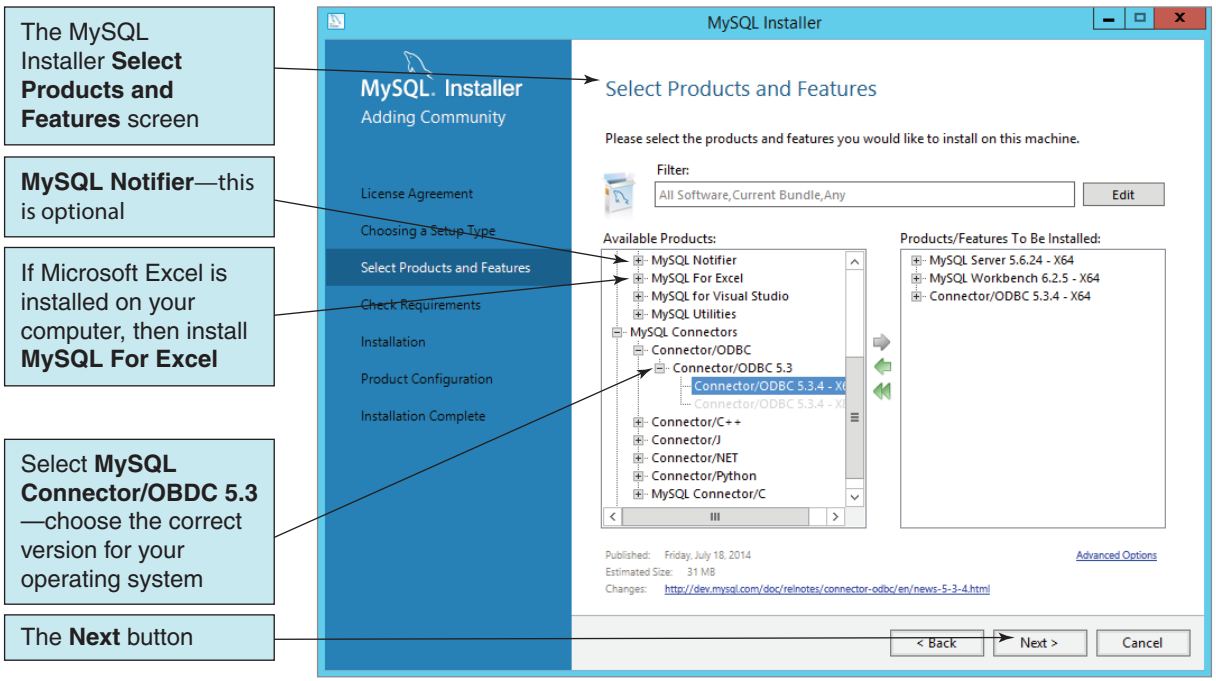


(d) The Select Products and Features Screen - Applications

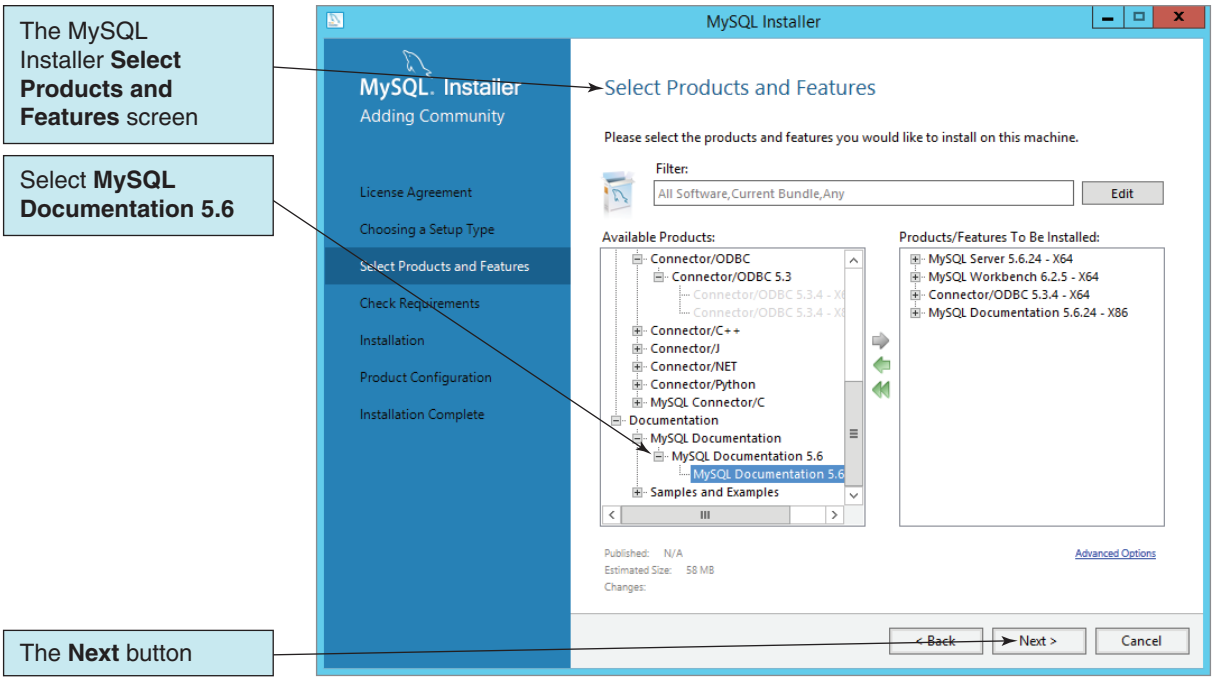
FIGURE 10C-1(d)
MySQL 5.6 Installation and Configuration

8. Click the **Next** button. The MySQL Installer checks your computer to make sure all installation requirements are met. If any requirement test fails, a message is displayed. If you have not installed the **Microsoft Visual C++ Redistributable Package for Visual Studio 2013** as discussed above, you will get an error message at this point. If all requirements are met, the MySQL Installer *Installation* screen is displayed, as shown in Figure 10C-1(g). This screen summarizes all the products that will be installed.
9. Click the **Next** button (note that although the screen text states *Press Execute…*, the button itself is labeled *Next*). The MySQL Installer installs all the selected products, as shown in the Figure 10C-1(g). The installation status for each product is marked as *Complete* after it is installed.

FIGURE 10C-1(e)
MySQL 5.6 Installation and Configuration



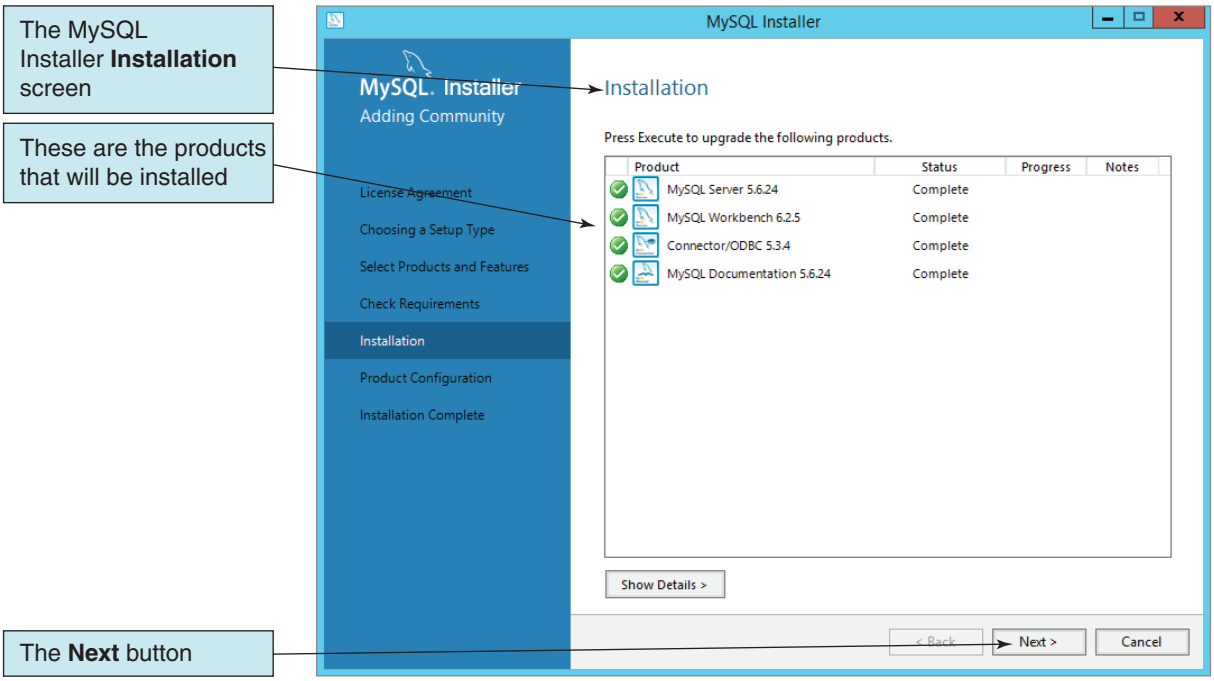
(e) The Select Products and Features Screen – MySQL Connectors



(f) The Select Products and Features Screen – Documentation

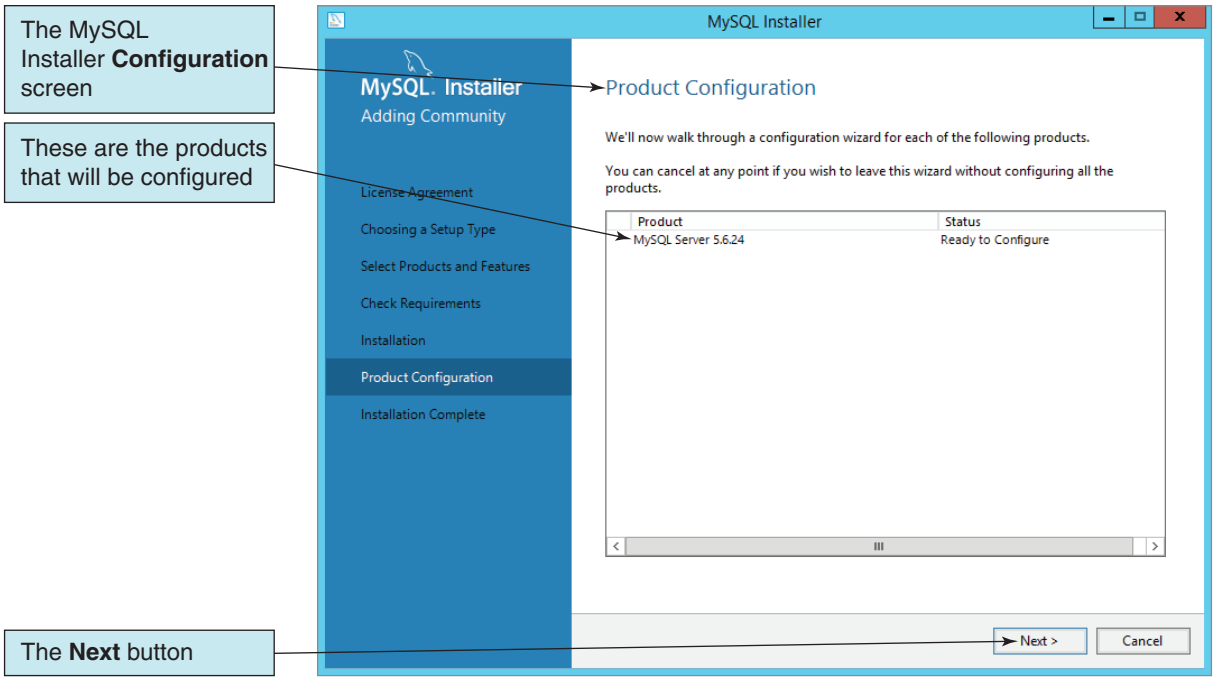
FIGURE 10C-1(f)
MySQL 5.6 Installation and Configuration

10. After the product installation for all products is complete, click the **Next** button. The *Product Configuration* screen is displayed, as shown in Figure 10C-1(h). We will now configure our installation of MySQL Server 5.6
11. Click the **Next** button. The *Type and Networking* screen is displayed, as shown in Figure 10C-1(i). We will configure our installation of MySQL as the default development machine, with the default TCP/IP options. We will not need to use the advanced configuration options.
12. Click the **Next** button. The *Accounts and Roles* screen is displayed, as shown in Figure 10C-1(j). We will need to enter a password for the *root user* account. The **root user** is



(g) The Installation Screen

FIGURE 10C-1(g)
MySQL 5.6 Installation and Configuration



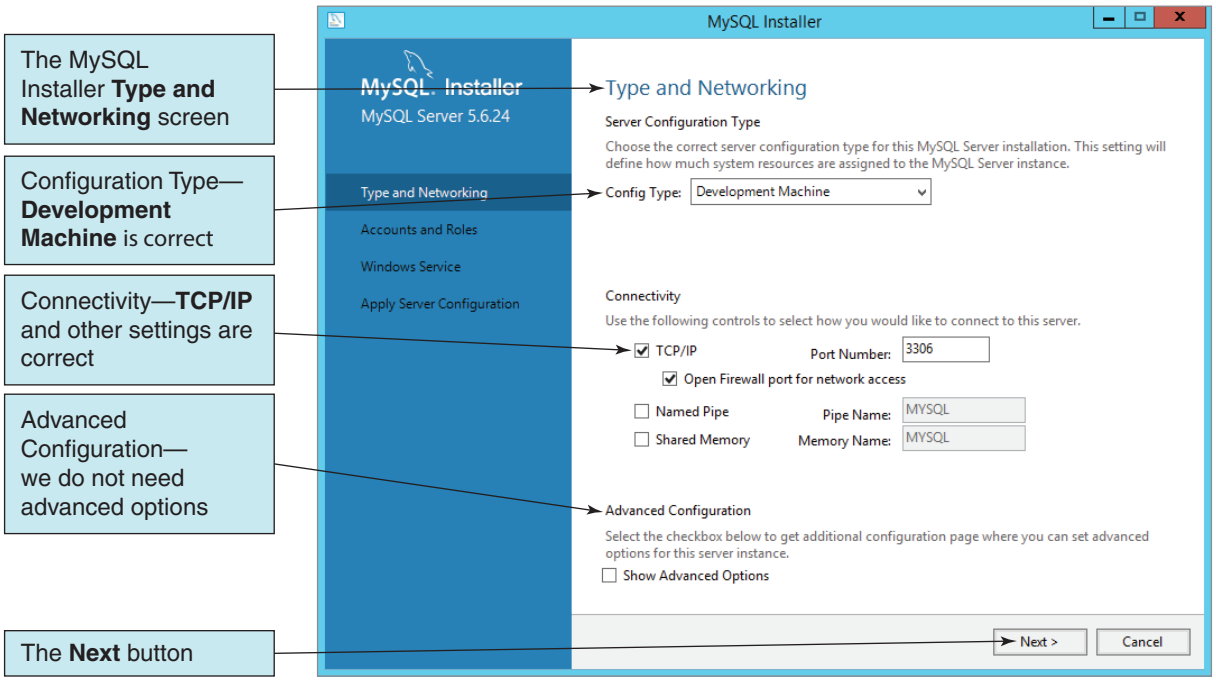
(h) The Product Configuration Screen

FIGURE 10C-1(h)
MySQL 5.6 Installation and Configuration

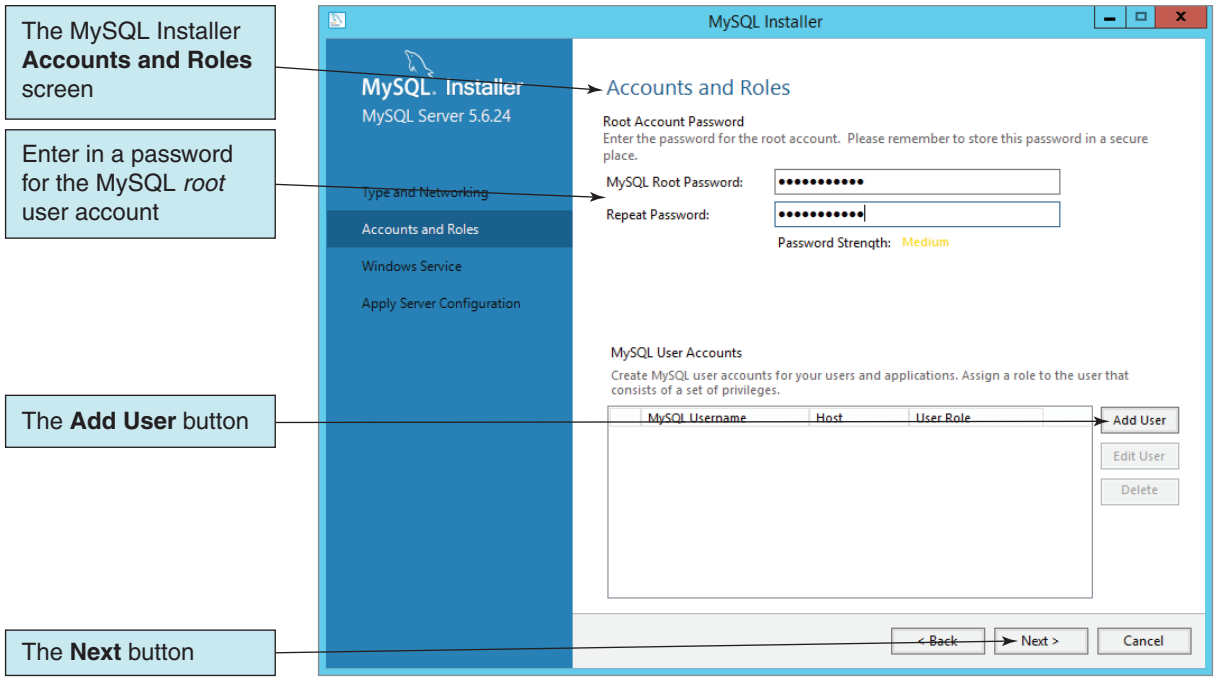
the name of the administrator user in MySQL, and this user has complete administration privileges on the MySQL server. Choose a password, then enter it in both the **MySQL Root Password** and **Repeat Password** text boxes.

13. In order to add yourself as a MySQL server administrator, click the **Add User** button. The **MySQL User Detail** dialog box is displayed, as shown in Figure 10C-1(k). Enter your name as the Username, and enter a password. Leave the other settings as shown. When you are done, click the **OK** button to close the MySQL User Detail dialog box.

FIGURE 10C-1(i)
MySQL 5.6 Installation and Configuration



(i) The Type and Networking Screen

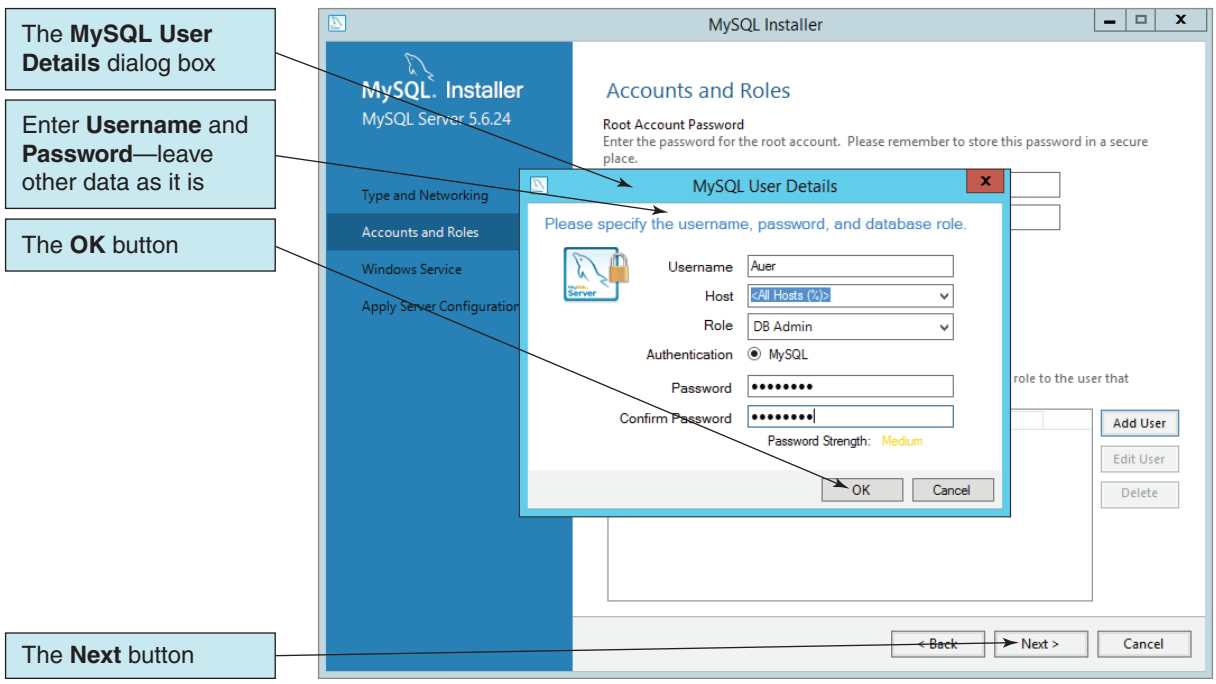


(j) The Account and Roles Screen

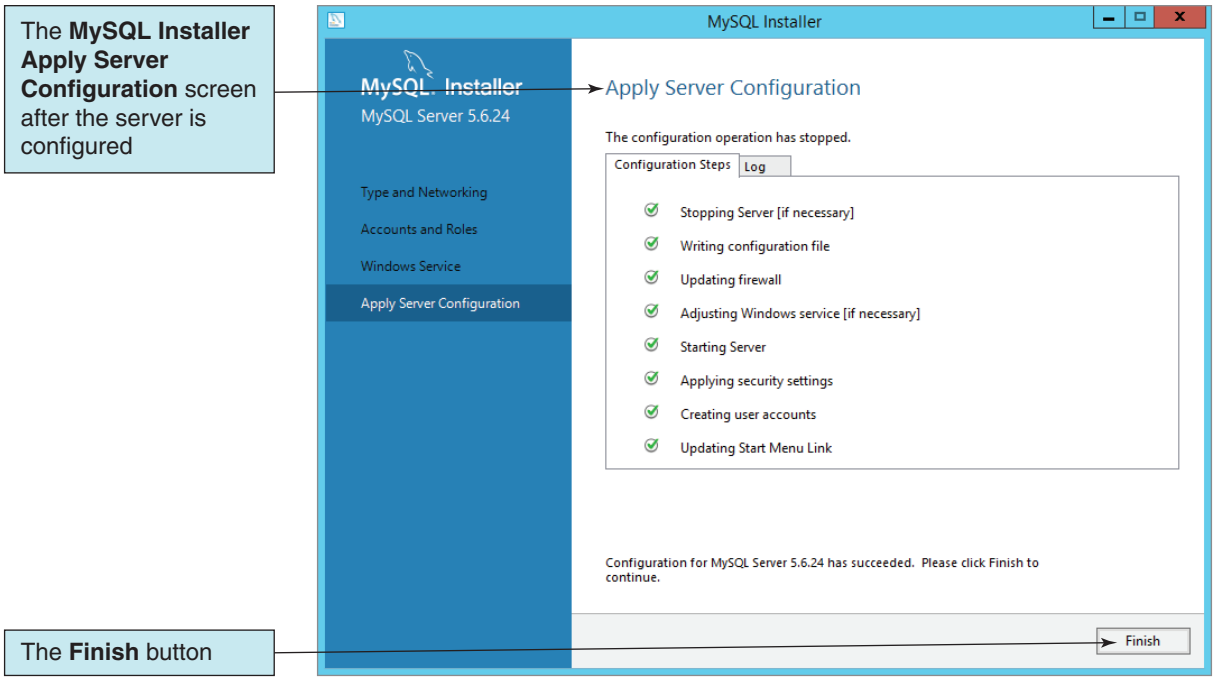
FIGURE 10C-1(j)
MySQL 5.6 Installation and Configuration

14. Back in the *Accounts and Roles* screen, you will see yourself added as a user. Click the **Next** button to display the *Windows Service* screen. We will use the default settings on this screen, so click the **Next** button.
15. The *Apply Server Configuration* screen is displayed, which summarizes the steps that will be run to configure the server. Click the **Finish** button to configure the MySQL server. When the configuration is complete, the *Apply Server Configuration* screen is updated and displayed as shown in Figure 10C-1(l).
16. Click the **Next** button. The *Product Configuration* screen is displayed, showing that the configuration is complete. Click the **Next** button.

FIGURE 10C-1(k)
MySQL 5.6 Installation and Configuration



(k) The MySQL User Details Dialog Box

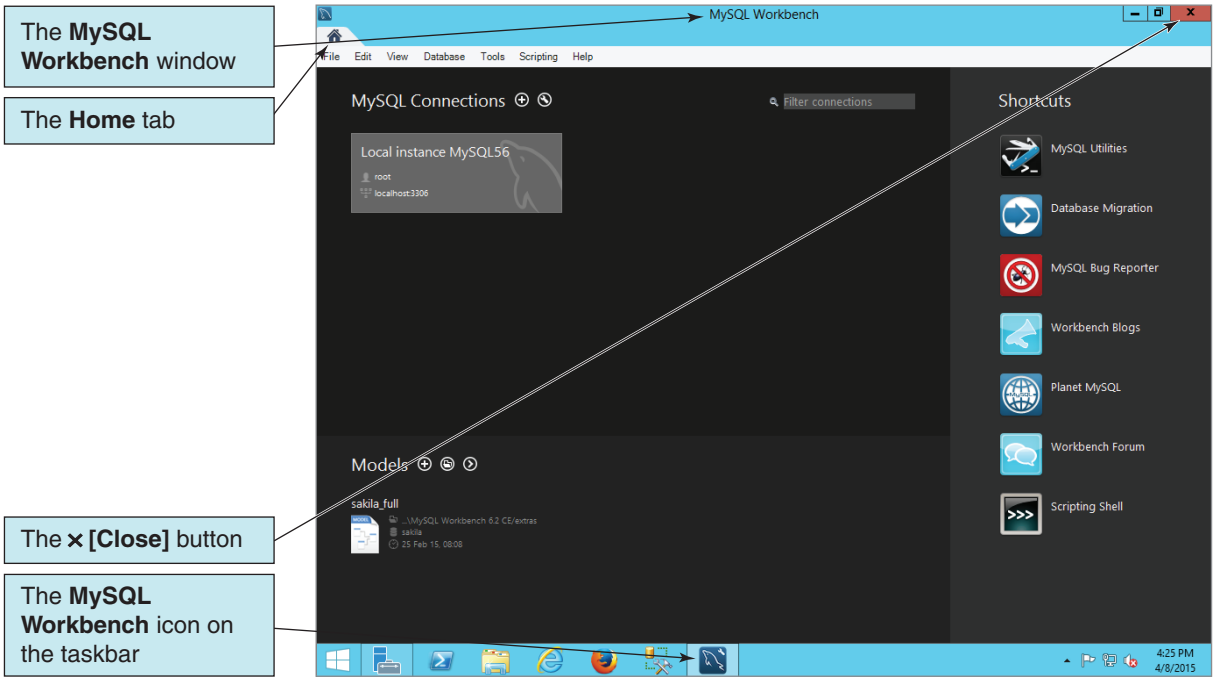


(I) The Completed Apply Server Configuration Page

FIGURE 10C-1(I)
MySQL 5.6 Installation and Configuration

17. The *Installation Complete* screen is displayed, with the *Start MySQL Workbench after Setup* checkbox checked. Click the **Finish** button.
18. The MySQL Installer dialog box closes, and the MySQL Workbench is opened and displayed as shown in Figure 10C-2. The MySQL Workbench will be the main MySQL GUI utility we use for our work in MySQL 5.6.
19. Click the MySQL Workbench **X [Close]** button to close MySQL Workbench.
    - **NOTE:** Now is a good time to add the MySQL Workbench icon to the Windows Task bar—this will make it much more convenient to open MySQL Workbench.

FIGURE 10C-2
MySQL Workbench



BY THE WAY Your installation of MySQL will install the current versions of these products, which will probably have different version numbers then shown in the discussion here. That is appropriate, because you always want to have the most current versions of the software installed.

After you have installed MySQL, you should check regularly for updated versions of the MySQL release you are using. For systems running a Windows OS, this functionality is built into the MySQL Installer for Windows, and you should use this utility to check for updates and then update your MySQL installation. Note that the MySQL Installer for Windows checks for updates for all MySQL products, not just MySQL Server.

For non-Windows OS installations, check the version number on the appropriate Downloads page. For example, we are currently using MySQL Community Server version 5.6.24, so when version 5.6.25 is released, we will need to download the new version and install it over the existing installation.

We need to check for updates because these updates are used in lieu of service packs and patches to make sure your installation is as secure as possible. There is no problem installing a newer version of MySQL over an older version of the same release. For Windows OS installations, the MySQL Installer for Windows can be installed into an existing MySQL setup and can then be used to maintain that installation.

Configuring Non-Windows Versions of MySQL Community Server

During installation of MySQL, you will be asked to configure your MySQL Community Server DBMS. Non-Windows OS systems (and Windows OS versions of previous versions of MySQL) will run the **MySQL Server Instance Configuration Wizard**. The term **instance** refers to the fact that you can have more than one MySQL DBMS installed on a computer. Each installed DBMS is referred to as an *instance*, and each instance must have a unique name.

We recommend that, for general use with this text, you choose the following settings and options with a Windows operating system installation.

Settings for the MySQL Installer for Windows

- For use with this text, choose **Development Machine.** Obviously, if we were setting up, for example, a DBMS dedicated to supporting Web database applications, we would choose one of the other settings.
- By default, **TCP/IP** connectivity is selected and enabled. Keep the default **3306** port number and the open firewall port. Unless you have a specific need for another connectivity option, do *not* enable the other options.
- We do *not* need advanced configuration options.
- Set the **MySQL Root Password** (this is the password for the user named *root*, which is the default Administrator account).
- Add a database administrator account for yourself. This will allow you administrator access without using the root user account.

MySQL Storage Engines

As we noted above and unlike DBMS products like SQL Server 2014 and Oracle Database, MySQL can actually use various **database storage engines**, each of which stores database table structures and data in a different manner. The MySQL architecture that supports more than one storage engine and the various storage engines available (there are nine) are discussed in the MySQL documentation at *http://dev.mysql.com/doc/refman/5.6/en/storage-engines.html*.

As far as we are concerned, the two main storage engines are the MyISAM engine and the InnoDB storage engine. The **MyISAM storage engine** was developed by MySQL AB and is the default storage engine for MySQL. The **InnoDB storage engine** was developed by the Finnish company Innobase Oy, which was purchased by Oracle in October 2005. In April 2006 (and after speculation in the database community due to the purchase by Oracle of a primary supplier of a major component of a competing product), Innobase and MySQL

implemented a multiyear extension of MySQL's license of the InnoDB storage engine. All this became a moot point when Oracle acquired the MySQL DBMS itself.

As far as we are concerned, the InnoDB storage engine has several advantages over the MyISAM engine:

- InnoDB stores rows in primary key order.
- InnoDB supports foreign keys and referential integrity.
- InnoDB supports ACID transactions (see Chapter 9).

Therefore, we will *always* use the InnoDB storage engine in our databases, and our configuration of MySQL in the previous section was done in such a way as to include InnoDB.

The MySQL Utilities

MySQL provides both **command-line utilities** and **graphical user interface (GUI) utilities** for use with the MySQL community server DBMS. While some DBAs prefer command-line utilities, the current GUI utilities are very comprehensive and user-friendly, and after introducing both types, we will use only the GUI utilities for the remainder of this chapter.

The MySQL Command-Line Client

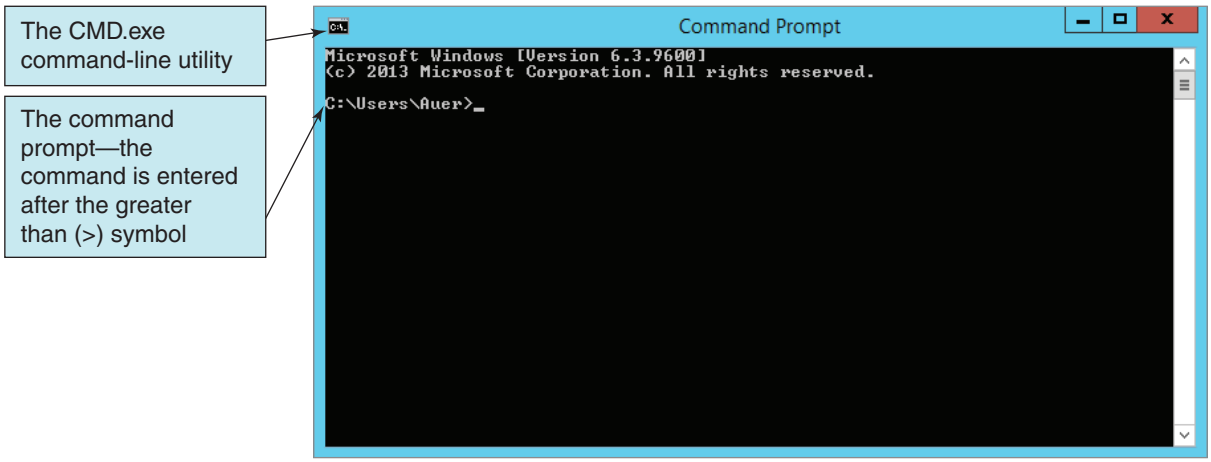
In the beginning, there were the command-line utilities. A command-line utility is strictly text based. You are presented with a symbolic prompt to show you where to enter your commands. You type in a command (only one at a time) and press the **Enter** key to execute it. The results are displayed as plaintext (with some rudimentary character-based line- and box-drawing capabilities) in response. All major computer operating systems have their version of a command-line utility. For personal computer users using a Microsoft operating system, the classic example is the MS-DOS command line, which still exists in Windows as the CMD program. The CMD program uses the Command prompt shown in Figure 10C-3.

For all versions of MySQL, the classic command-line tool has been the **MySQL Command Line Client**, which was installed as part of the MySQL installation. In Windows 8.1, the MySQL Command Line Client is started by clicking on an icon in the Start screen.[1] Type in the **root** user password, and the MySQL Command Line Client is displayed, as shown in Figure 10C-4.

The MySQL GUI Utilities

Although command-line utilities can be powerful, they can also be tedious and ugly. That's why GUI applications such as Windows were created in the first place. And popular personal databases such as Microsoft Access have certainly put GUI features to good use. Therefore, we

FIGURE 10C-3
The Microsoft Windows CMD Command-Line Utility



[1]The command to start the MySQL Command Line Client in Windows 7.1 is **Start | All Programs | MySQL | MySQL Server 5.6 | MySQL Command Line Client**.

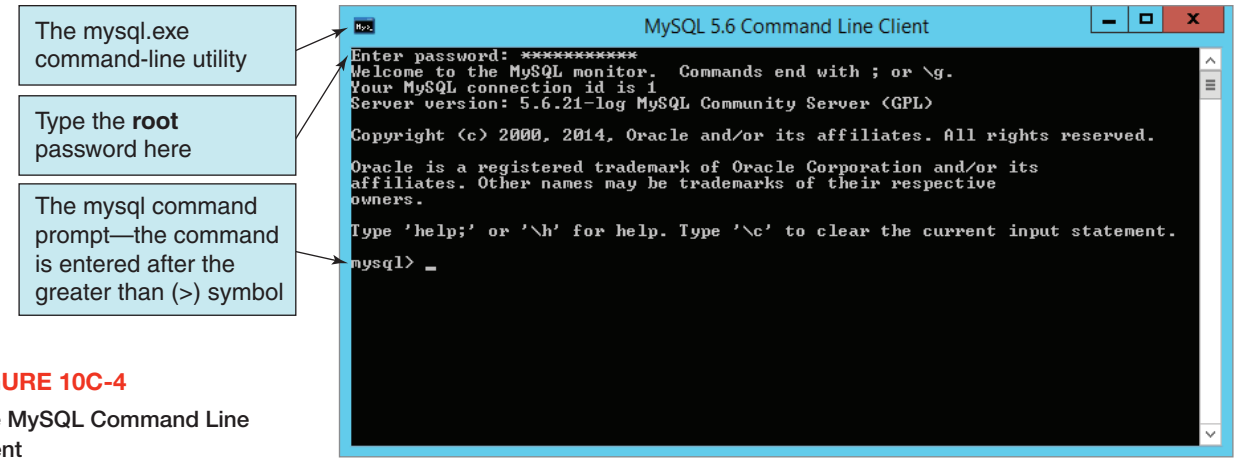


FIGURE 10C-4
The MySQL Command Line Client

have to ask, "Is there a GUI display for MySQL?" The answer, of course, is "yes, there is," and it is the MySQL Workbench that we saw at the end of the MySQL installation process.

The **MySQL Workbench** provides data modeling, database development, and MySQL server administration tools. The MySQL Installer for Windows installs the MySQL Workbench on Windows OS systems. If you are running a non-Windows OS, you should download and install the MySQL Workbench for your operating system from *http://dev.mysql.com/downloads/workbench/* after you install and configure MySQL 5.6.

As with MySQL itself, check for updated versions of the MySQL Workbench and install them as they become available. We are running MySQL Workbench CE 6.2.5. This is the freely downloadable Community Edition, and it has some functional limitations. A full-featured, commercial version of MySQL Workbench is included in MySQL Enterprise Edition.

In this chapter, we will use the MySQL Workbench for database development and MySQL server administration. However, it is a very useful tool for creating database designs, as discussed in Chapter 6. For a full discussion of how to use MySQL Workbench for data modeling, see Appendix E.

BY THE WAY

During the time period for the writing and publishing of this edition of *Database Processing*, we have seen some major changes in the GUI format and features of the MySQL Workbench. Apparently, this has been a period of rapid development for MySQL Workbench. It also means that, although we are showing the current version of the program in the screenshots and listing the correct steps for using the various features of MySQL, you may encounter a slightly different version of MySQL Workbench while you are using this book.

We expect all the functionality and features we describe to still be available, but labels, locations, and access methods may change. If something isn't quite the way we describe it, it should still be there in a slightly different incarnation.

We can start working with MySQL by opening MySQL Workbench. In Windows 8.1 and Windows Server 2012 R2, MySQL Workbench is opened by clicking an icon in the Start screen. However, to make it easier, we recommend pinning the program to the Windows taskbar, as shown in Figure 10C-5,[2] which illustrates the basic **MySQL Workbench Home tab** screen.

The MySQL Workbench Home tab is a dashboard allowing us access to MySQL database design, SQL database development, and MySQL DBMS administration. These features can be accessed in several ways, but we will cover only some basic ones here.

[2]In the Windows 7 operating system, select **Start | All Programs | MySQL | MySQL Workbench CE 6.2**.

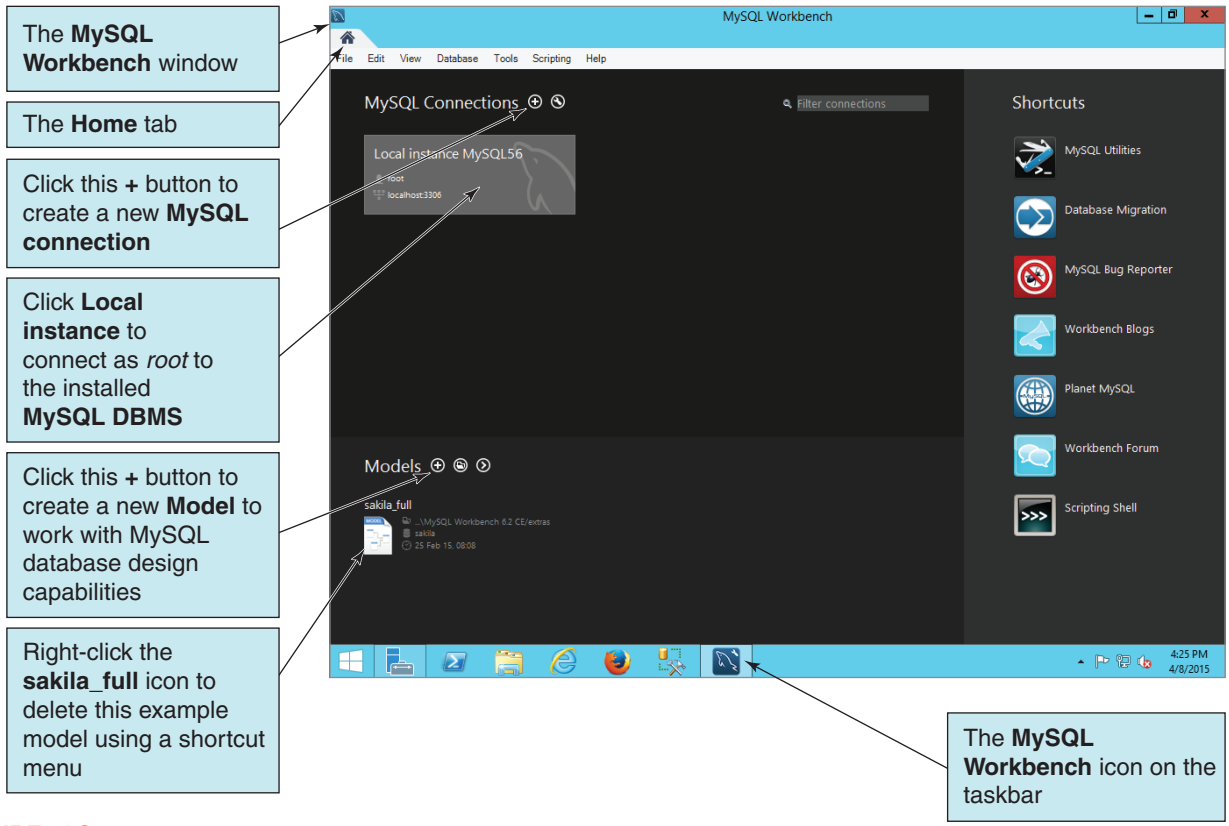


FIGURE 10C-5

The MySQL Workbench Window

BY THE WAY The first time you open the MySQL Workbench, you will see an existing model named *sakila_full* in the Models list. We do not need this model. To delete the *sakila_full* model:

- Right-click the EER model name ***sakila_full*** in the Models list to display a shortcut menu.
- In the shortcut menu, click either the **Remove Model File from List** command (to remove just the sakila_full model from the list) or the **Clear List** command (to remove all the models from the list).

Creating a Workspace for the MySQL Workbench Files

Before using the MySQL Workbench, we recommend creating a folder named *MySQL Workbench* under the My Documents folder (or whatever your main data storage area is named). In Windows, this can be done using Windows Explorer, as shown in Figure 10C-6. In this workspace, create two folders, *EER Models* (for database designs) and *Schemas* (for database scripts). In the Schemas folder, create a subfolder for each database project. At this point, we need to create a subfolder for the Cape Codd database as used in Chapter 2.

Creating and Using a MySQL Database

Now that the MySQL DBMS is installed and we have the MySQL Workbench open, we can create a new database. We will create a database named *Cape_Codd* for the Cape Codd Outdoor Sports database we used in Chapter 2 to discuss SQL query statements.

Creating a Database in MySQL

To create a database in MySQL Workbench, start by opening an SQL Development window. To do this, we need to use a **MySQL connection**, which links MySQL Workbench to the

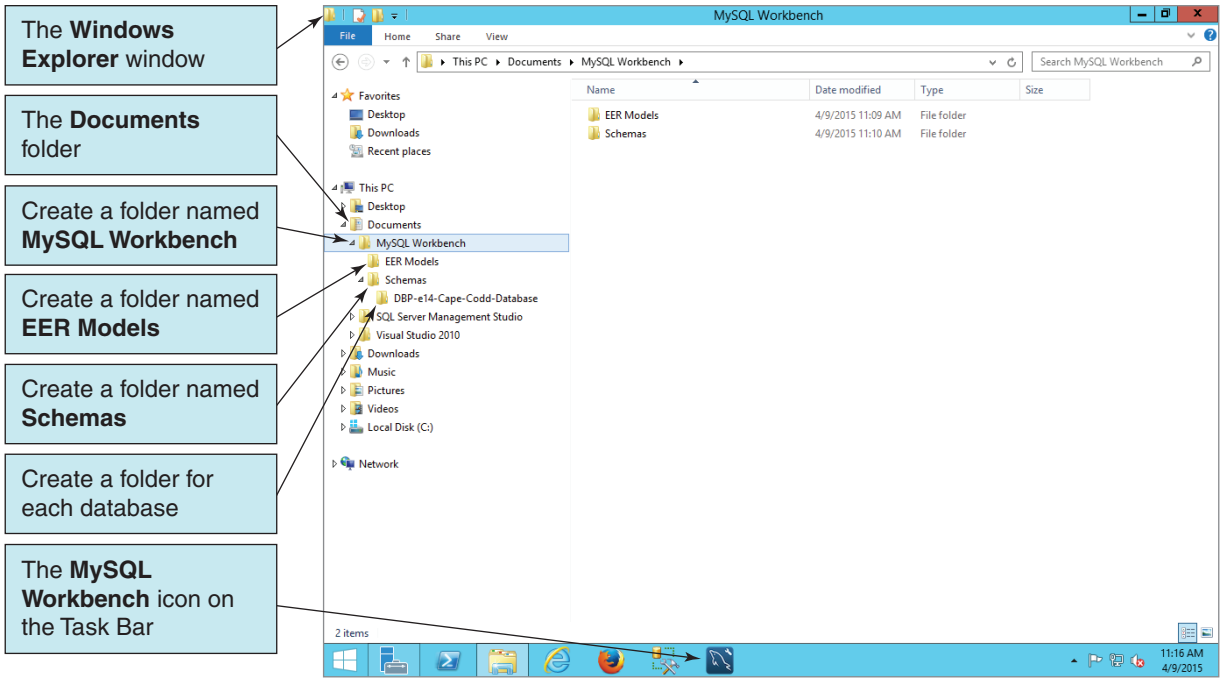


FIGURE 10C-6

The MySQL Workbench Folder in Windows Explorer

installed MySQL community server DBMS. When we install MySQL using the MySQL Installer for Windows, the installation process actually creates a **local instance connection**, which is a connection to the MySQL community server DBMS we installed on our computer. This can be seen in Figure 10C-5, where it appears as *Local Instance MySQL56*. Note that this user is a connection for the *root*[3] user and requires the password for the root user that we created during installation.

We will create another database for the user account we created during the installation process (in our case, for the *Auer* account).

Creating a MySQL Connection in MySQL Workbench

1. Click the New Connection button shown in Figure 10C-5. The **Setup New Connection** dialog box is displayed, as shown in Figure 10C-7.
2. Type in a name for the connection in the *Connection Name* text box. In this case, we are naming the connection **WS12R2-001 MySQL 5.6**. This name is based on the name of the computer we are using (*WS12R2-001*, which is running Windows Server 2012 R2) and the version of the MySQL DBMS we are connecting to (*MySQL 5.6*).
3. Type in a username for the connection in the *Username* text box. In this case, we are setting up a connection for **Auer**.
4. Leave all the other settings as their default values.
5. Click the **OK** button. The next connection is created and added to the MySQL Connections section of the MySQL Workbench Home tab, as shown in Figure 10C-8.

Now we can log onto the MySQL DBMS using our new connection.

Connection to a MySQL DBMS in MySQL Workbench

1. Click the new MySQL connection that you created, which is *WS12R2-001 MySQL 5.6* in our example. The **Connect to MySQL Server** dialog box is displayed, as shown in Figure 10C-9.

[3]As discussed earlier in this chapter, *root* is the name of the default MySQL administrator account. You were prompted for a password for this account during the installation of MySQL, and you should have already created that password. The username *root* comes from the Unix and Linux operating systems, where it is the name of the default system administrator account.

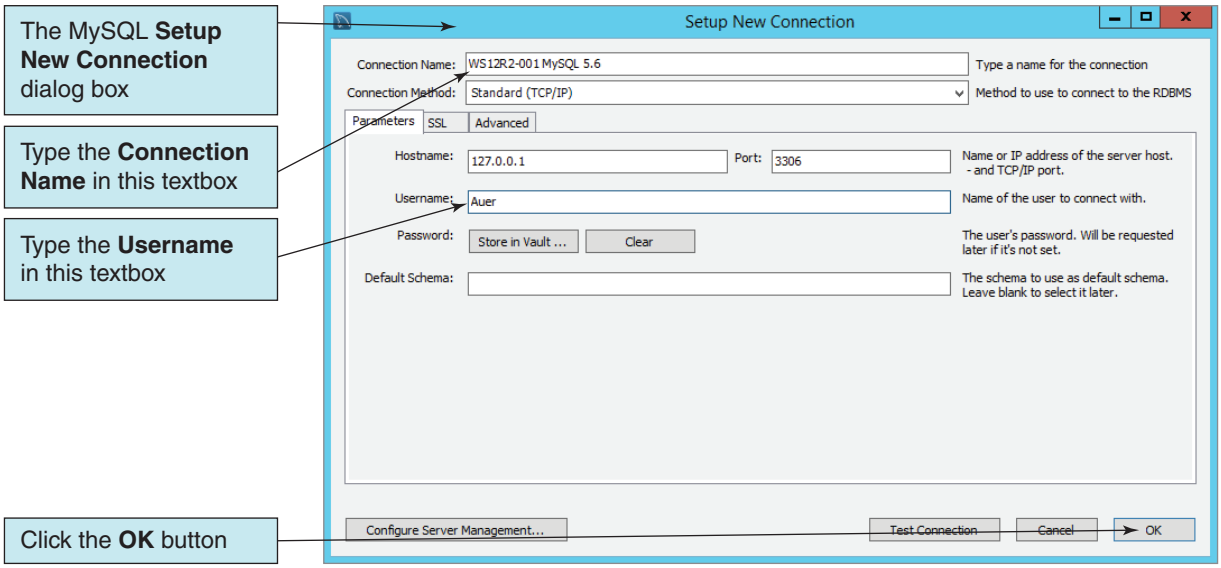
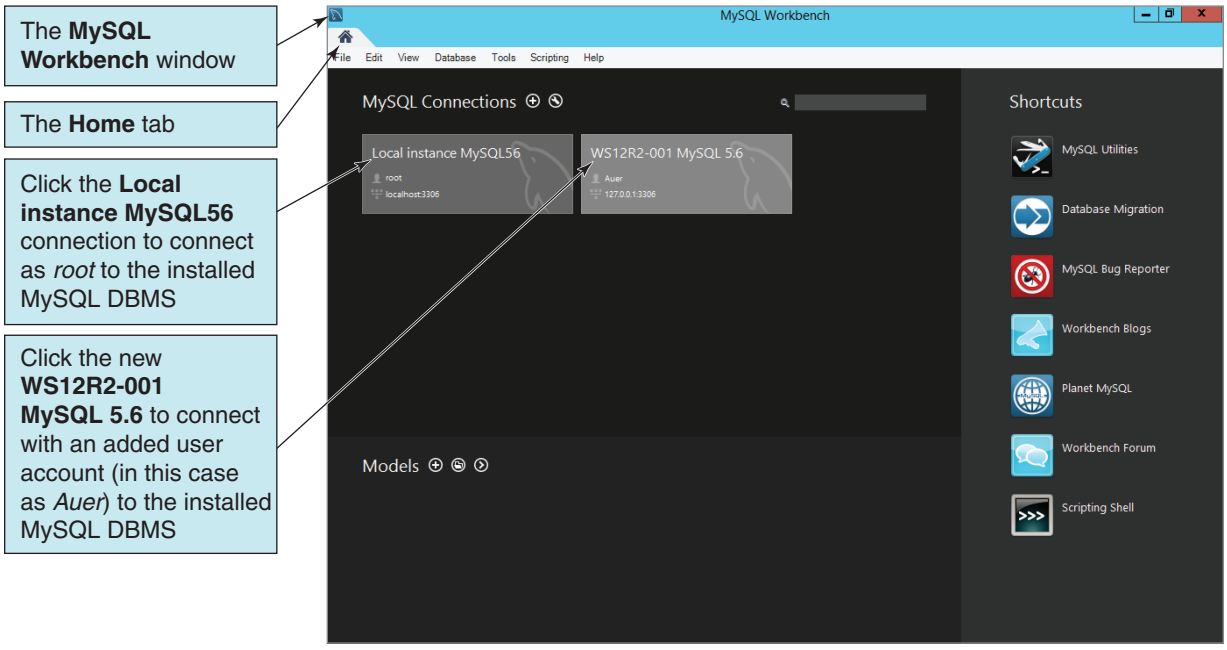


FIGURE 10C-7
The Setup New Connection Dialog Box

2. Type in the password for the username for the connection in the *Password* text box. In this case, we are using the password we created for the user *Auer*.
3. We recommend that you do *not* save the password in the vault.
4. Click the **OK** button. The user is logged in, and the MySQL Workbench MySQL Editor window is displayed, as shown in Figure 10C-10.

The MySQL Workbench reappears, but now with an **SQL Editor window**, designated by an **SQL Editor window tab**, as shown in Figure 10C-10(a), where we connected by clicking the **WS12R2-001 MySQL 5.6** connection object [note that we are connected to IP address 127.0.0.1:3306—127.0.0.1 indicates our own computer, and 3306 is the MySQL TCP/IP port number we designated during installation as shown in Figure 10C-1(i)]. Note the buttons that control the **Navigator window**, the **SQL Additions window**, and the **Output window**. We prefer to generally work with the Navigator open, but with the other two windows closed. This configuration can be seen in Figure 10C-10(b), which is much cleaner. Note the **Add Schema button** shown in this figure. **Schema** is MySQL's term for a database.

FIGURE 10C-8
The New MySQL Connection in MySQL Workbench



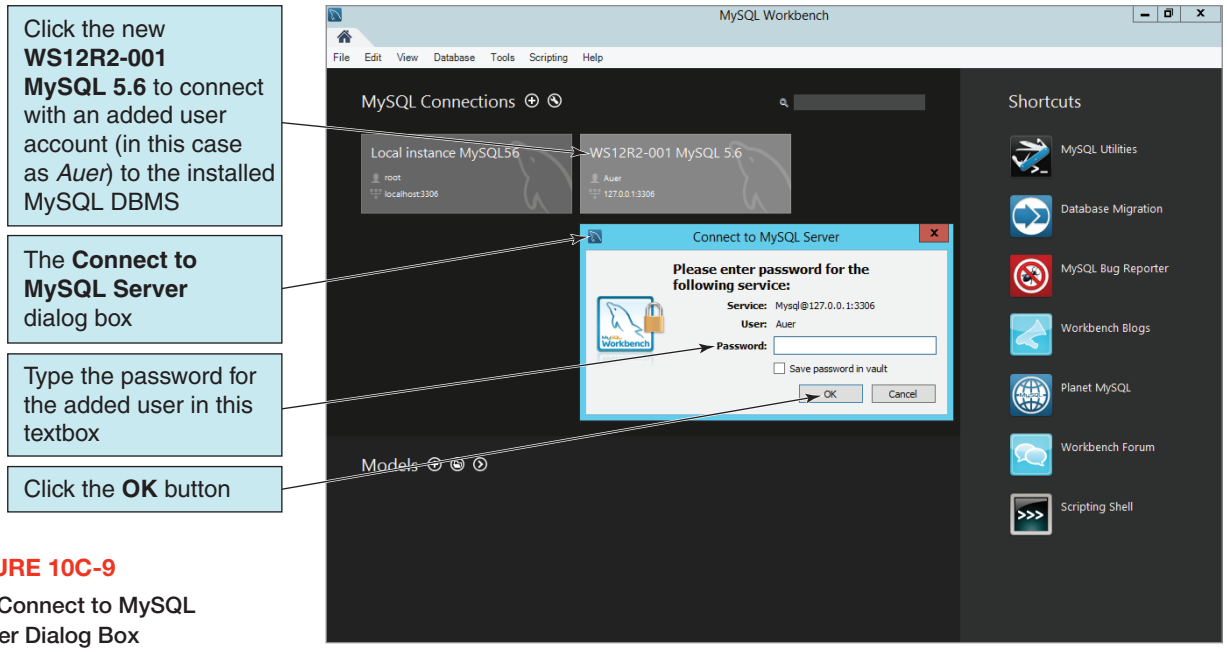


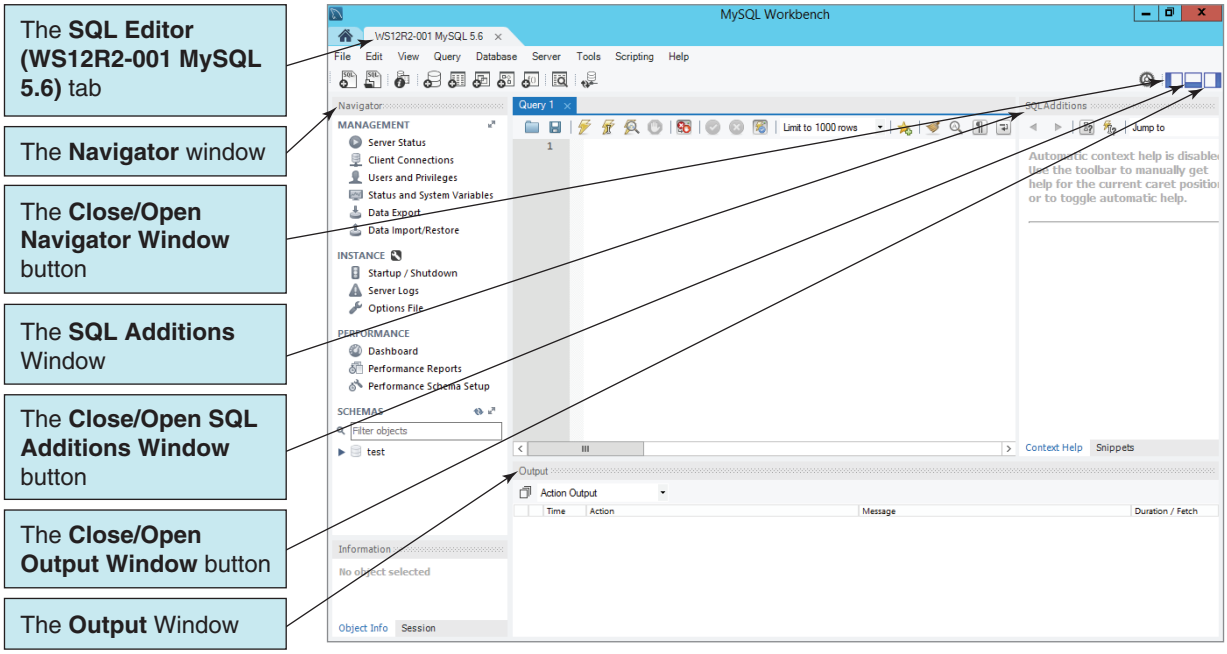
FIGURE 10C-9
The Connect to MySQL Server Dialog Box

To illustrate how to create a new database in MySQL Workbench, we will create the database for Cape Codd Outdoor Sports that is used as our example database in Chapter 2. We will use the database name *Cape_Codd*, and the Cape_Codd database can be used to create and run all the SQL statements in Chapter 2 and the Chapter 2 Review Questions.

Creating a MySQL Database

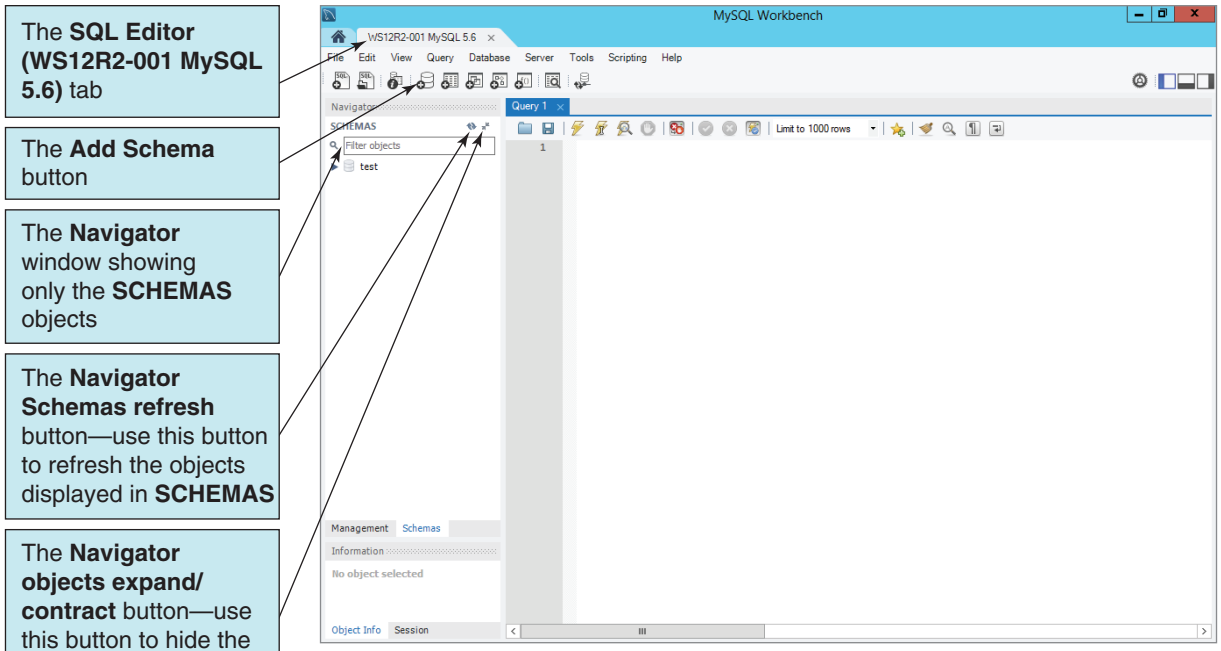
1. Click the **Add Schema** button, shown in Figure 10C-10(b).
2. The new_schema dialog tab is displayed, as shown in Figure 10C-11.
3. Type the new database (schema) name *Cape_Codd* in the **Name** textbox, and then click the **Apply** button. Because we used *capital (uppercase) letters* in our schema

FIGURE 10C-10
The SQL Editor Window



(a) The WS12R2-001 MySQL 5.6 Connection

(*continued*)



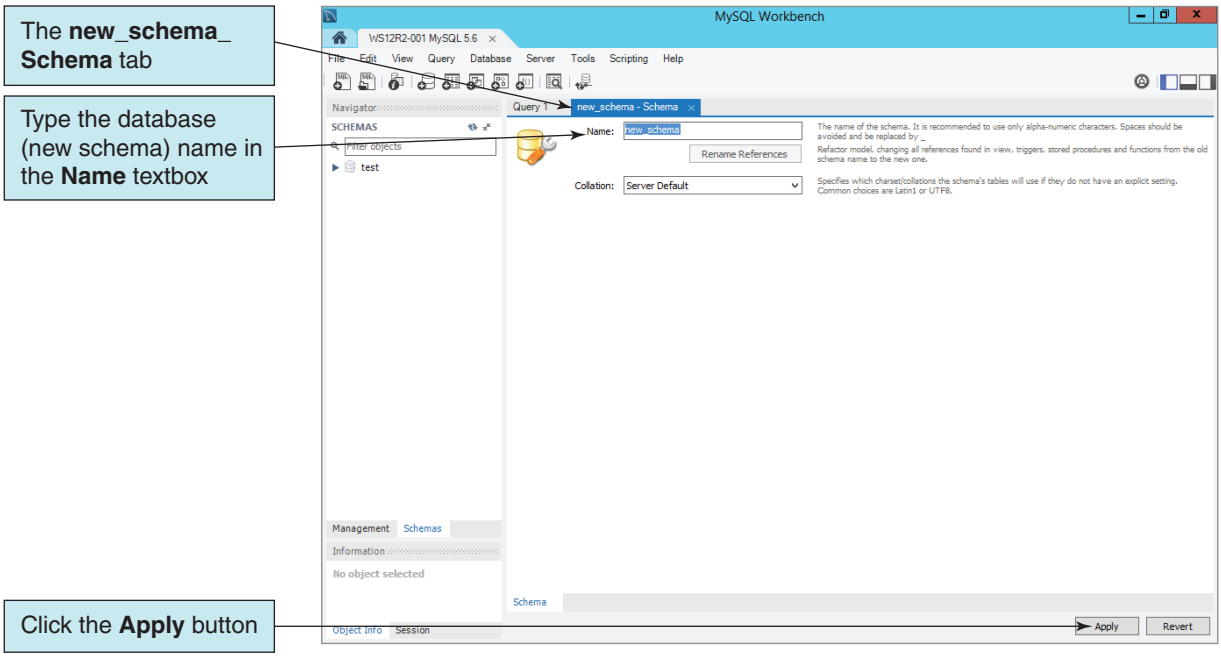
(b) The Add Schema Button

FIGURE 10C-10
Continued

name, the Apply Changes to Object warning dialog box is displayed, as shown in Figure 10C-12. By default, MySQL is configured to allow only *lowercase* letters in database names. While we could change this, we prefer to use as many MySQL defaults as possible, and so we will acknowledge the warning and allow MySQL to implement the database as *cape_codd*.

4. Click the **OK** button to close the Apply Changes to Object warning dialog box, and then click the Apply button at the bottom right of the **Cape_Codd - Schema** window.
5. As shown in Figure 10C-13, an **Apply SQL Script to Database** dialog box is displayed so the user can review the SQL command before it is executed. The SQL

FIGURE 10C-11
The new_schema Tab



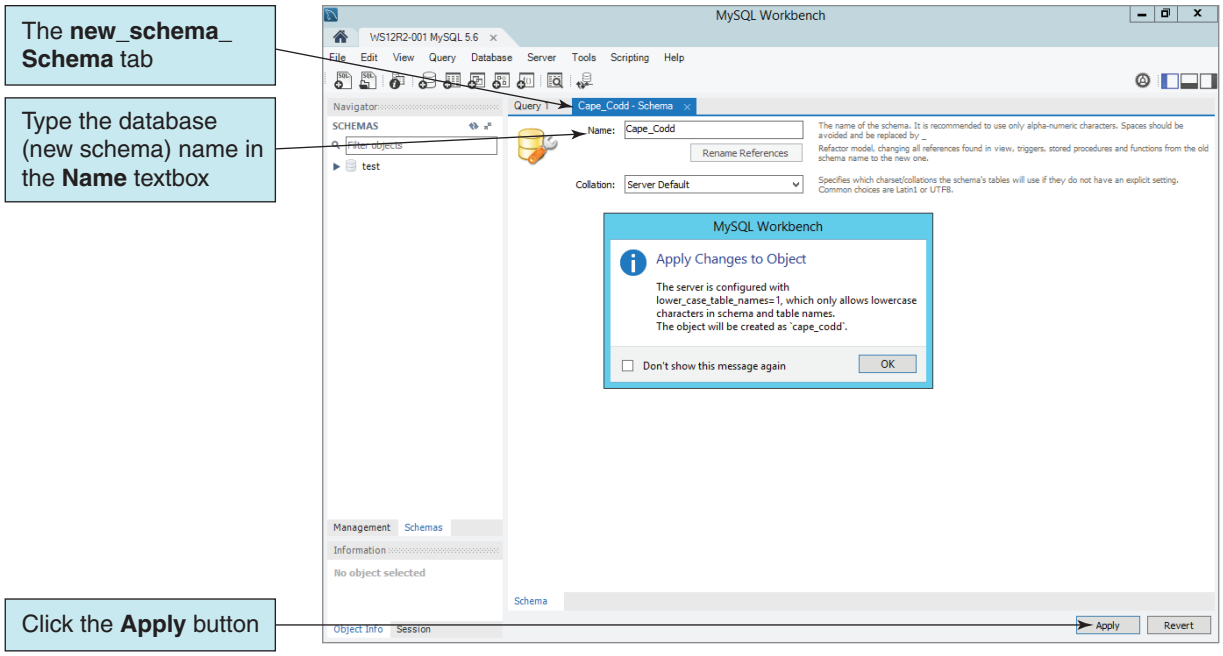


FIGURE 10C-12

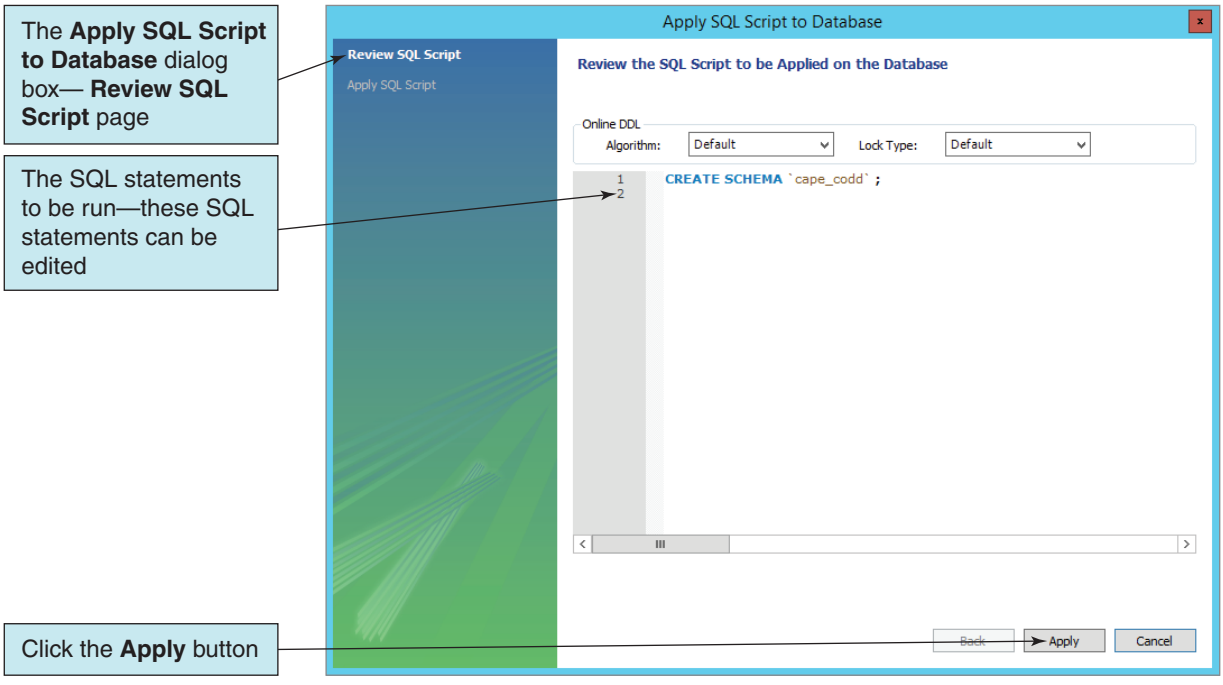
The Apply Change to Object Warning Dialog Box

statement shown in this dialog box is the SQL statement that will be run. If any modifications are needed, the edits can be made here. Because we do not need to make any edits, click the **Apply SQL** button.

6. As shown in Figure 10C-14, the Apply SQL Script to Database dialog box is displayed with the results of executing the SQL command. Click the **Finish** button.
7. As shown in Figure 10C-15, the **cape_codd - Schema** dialog box is displayed again, but note the name change from Cape_Codd (uppercase letters) to cape_codd (lowercase letters) and the *cape_codd* schema object in the Navigator. Click the **Close** button on the *cape_codd - Schema* tab.
8. Figure 10C-16 shows the new cape_codd database (schema) object in the Navigator window.

FIGURE 10C-13

The Apply SQL Script to Database—Review SQL Script Dialog Box



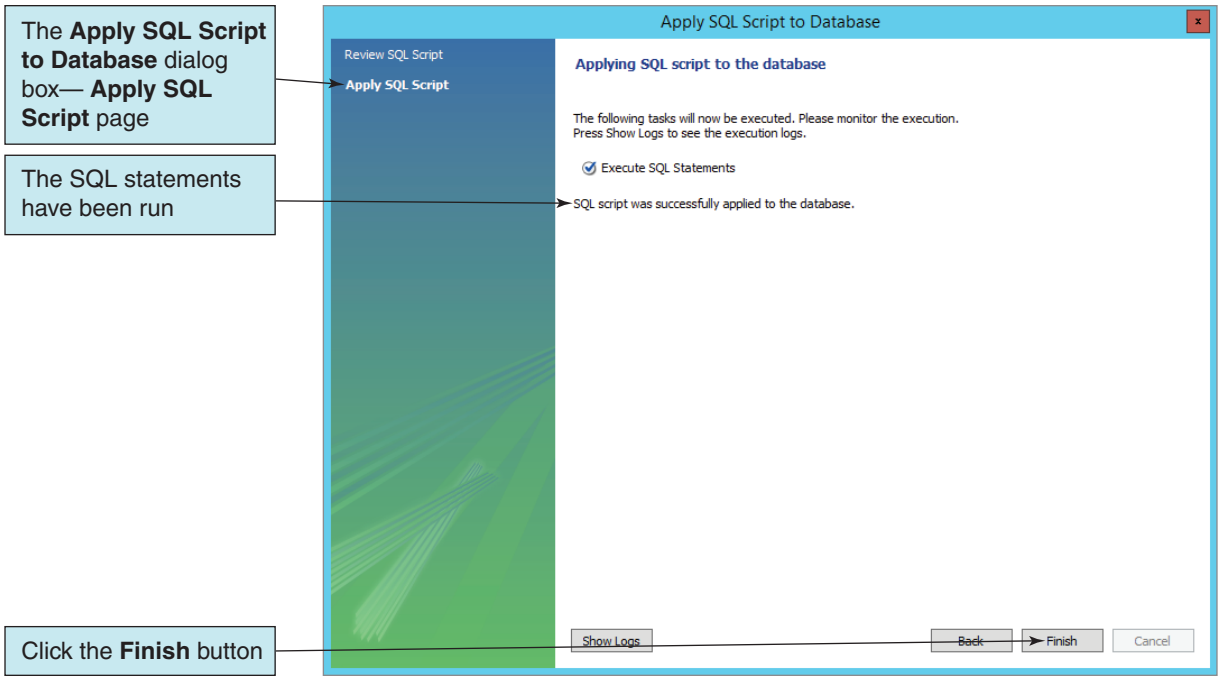


FIGURE 10C-14

The Apply SQL Script to Database—Apply SQL Script Dialog Box

Setting the Active Database in MySQL

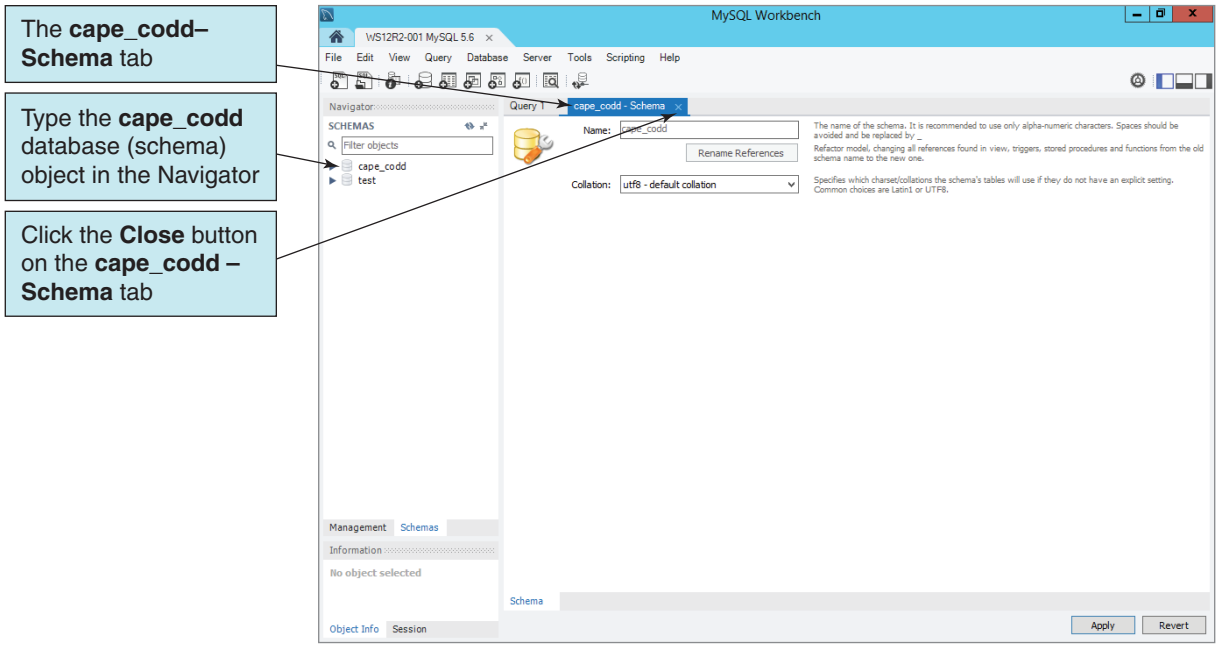
To work with a MySQL database, you must select it as the active database. In MySQL terms, this is called the **default schema**. As mentioned earlier, *schema* is MySQL's synonym for *database* (and if we use the **SQL CREATE DATABASE statement** to create a database in MySQL, the database is displayed as a *schema* in MySQL). We will make *cape_codd* the default schema so we can use the cape_codd database.

Setting a Default Schema in MySQL

1. Right-click on the cape_codd schema object to display a shortcut menu, as shown in Figure 10C-17(a).

FIGURE 10C-15

The New cape_codd Database (Schema) Object in the Navigator



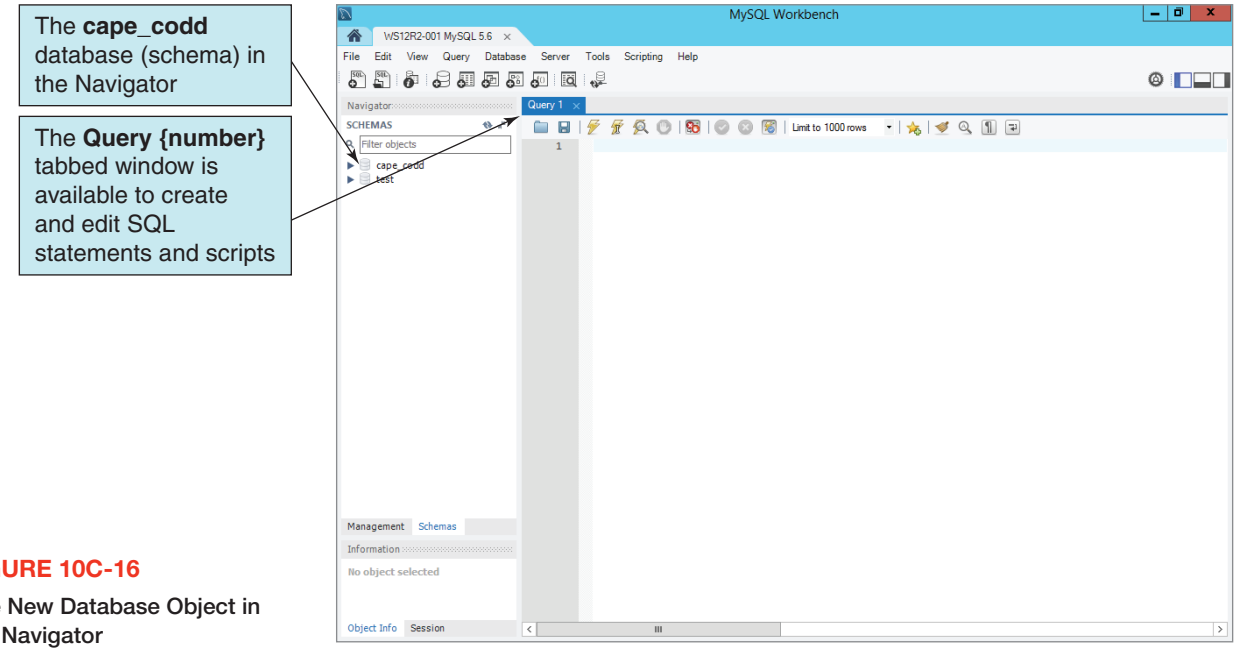


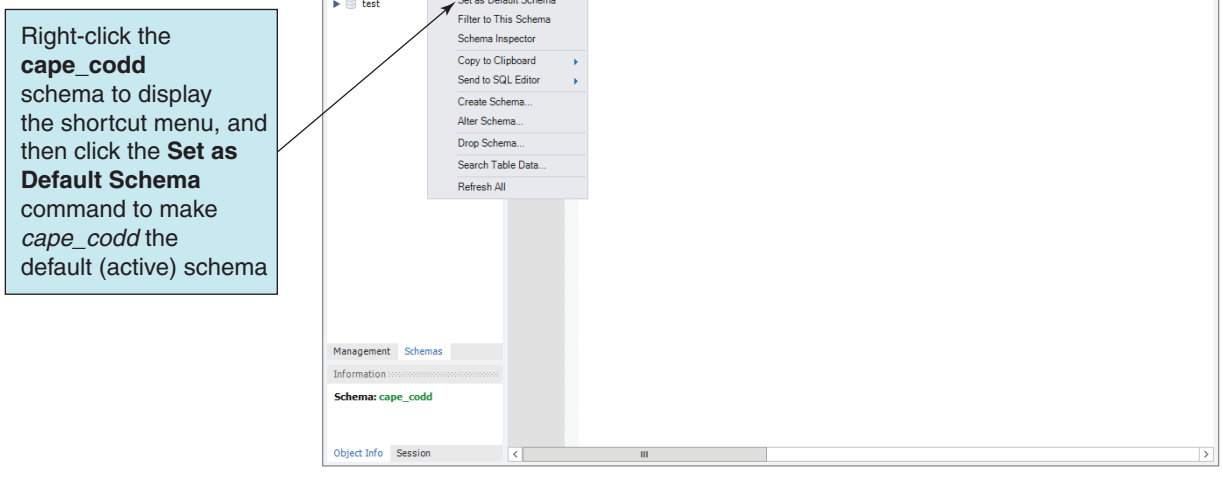
FIGURE 10C-16

The New Database Object in the Navigator

2. Click the **Set as Default Schema** name command to set cape_codd as the active database.
3. The results can be seen in Figure 10C-17(b), where the cape_codd schema object is now shown in a bold font.
4. Figure 10C-17(b) also illustrates an alternate technique to set the default schema–the use of the SQL USE {Schema Name} statement. When this statement is run (one way to do this is to click the **Execute SQL Statement [under the keyboard cursor]** button, which is used to run individual SQL statements), it will set the default schema. The advantage of this method is that the SQL statement can precede and be combined with other SQL statements to ensure that the SQL statements are applied to the correct database.

FIGURE 10C-17

Setting the Default Schema



(a) The Set as Default Schema Command

(*continued*)

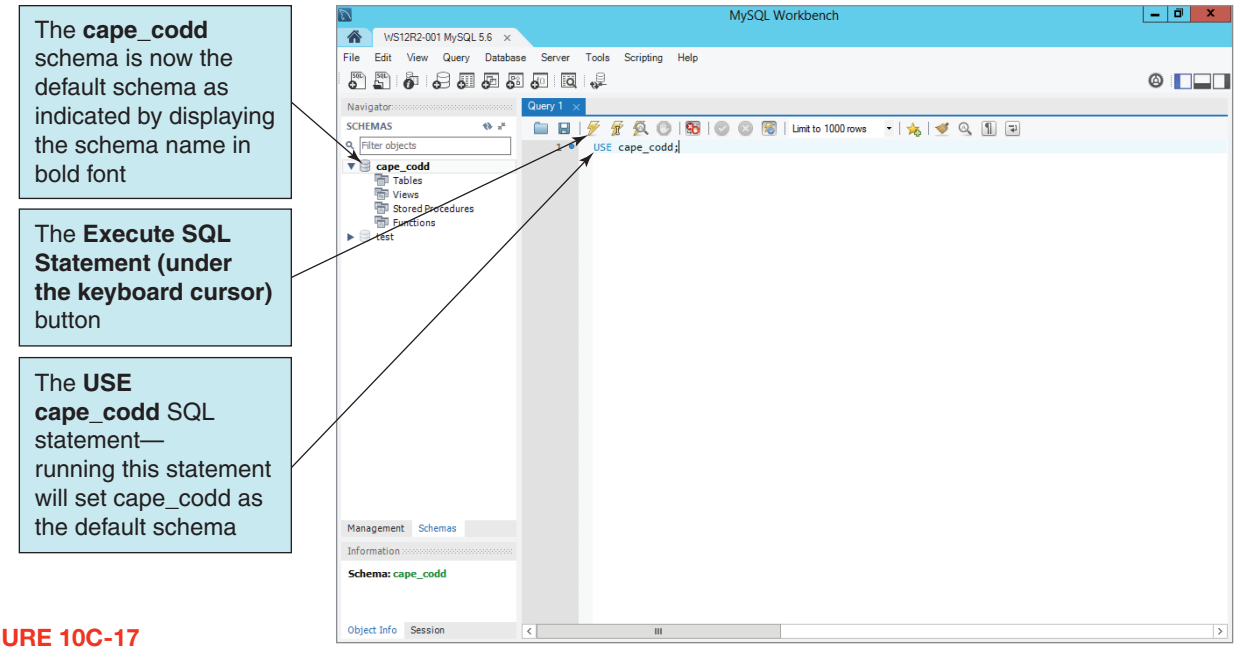


FIGURE 10C-17
Continued

(b) The SQL USE {SchemaName} Statement

BY THE WAY

Books on systems analysis and design often identify three design stages:

- Conceptual design (conceptual schema)
- Logical design (logical schema)
- Physical design (physical schema)

The creation and use of file structure and file organization (including physical storage placement and file characteristics) to store database components and physical records of data are a part of the *physical design*, which is defined in these books as the aspects of the database that are actually implemented in the DBMS. Besides physical record and file structure and organization, this includes indexes and query optimization.

MySQL SQL Statements and SQL Scripts

Because we have already argued that you need to know how to write and use SQL statements instead of relying on GUI tools, we come back to simply using SQL as the basis for our work. But we do not want to use a command-line utility, and we are not going to use the GUI tool in GUI mode, so what is left?

The answer is that the MySQL Workbench provides us with an excellent SQL editing environment. This lets us take advantage of GUI capabilities while still working with text-based SQL statements. We do this by opening and using an SQL Editor window in which to create and edit our SQL statements. Note that, when we connect to the MySQL server, the SQL Editor window is opened with a tabbed SQL Query window open by default. Thus, we already have one available. We can open others, as needed, by using the **File | New Query Tab** command or by clicking the **Create a new SQL tab** button.

The SQL editing environment in a MySQL Script window will be our tool of choice for editing SQL DDL statements. One advantage of using this SQL Editor is the ability to save and reuse SQL Scripts. For MySQL, **SQL scripts** are plaintext files labeled with the **.sql* file extension. We can save, open, and run (and rerun) SQL scripts.

By default, MySQL will save scripts in the user's Documents folder. Because this does not separate MySQL files from other data files, we recommend using the folder structure we created earlier in this chapter and shown in Figure 10C-6 with a *MySQL Workbench* folder; with

separate folders named *EER Models* and *Schemas*; and with a separate folder for each database project, such as *DBP-e14-Cape-Codd-Database*, under it.

An SQL script is composed of one or more SQL statements, which can include SQL script comments. **SQL script comments** are lines of text that do not run when the script is executed but are used to document the purpose and contents of the script. Each comment line begins with the characters **/* (slash asterisk)** and ends with the characters ***/ (asterisk slash)**.

Using Existing SQL Scripts

Another advantage of SQL scripts is that we can use scripts written by others. To do this we simply store the script in an appropriate location on our computer, open it in an SQL script tabbed window, and execute it.

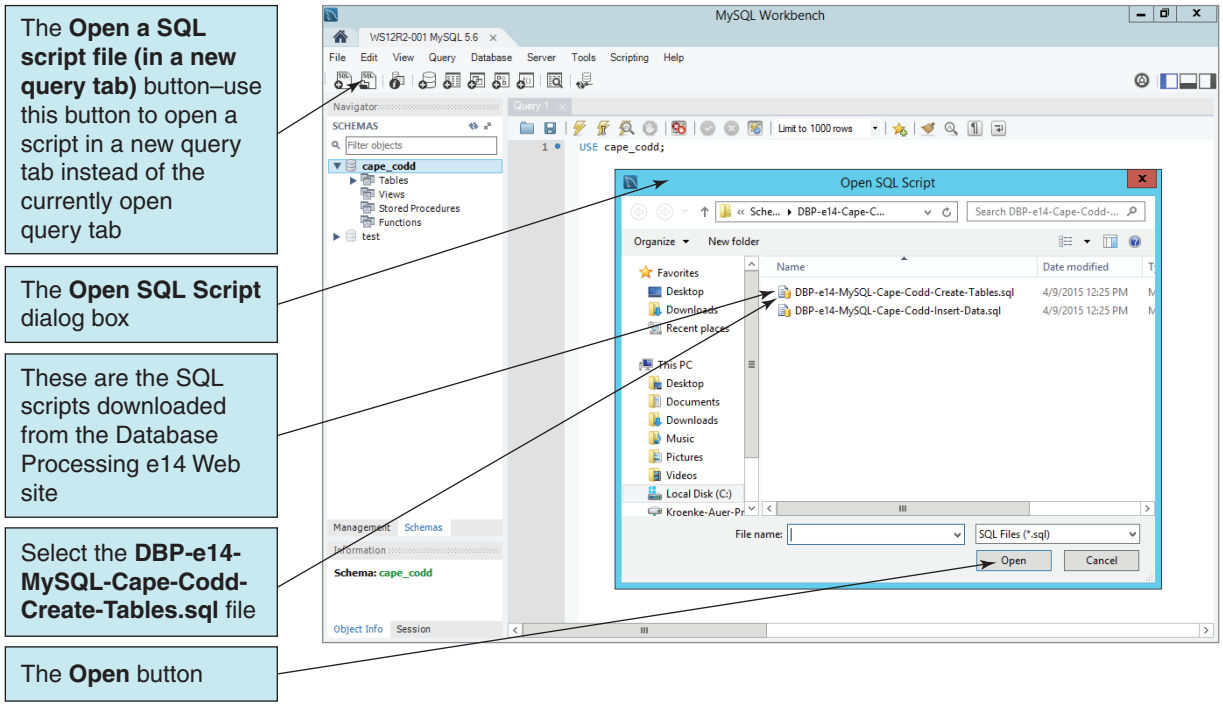
To illustrate this, we will build the Cape Codd database used in Chapter 2 to run the example SQL query statements. This will then allow you to run the Chapter 2 statements in an actual database and to work the Project Questions at the end of the chapter.

Opening and Running an Existing SQL Script

1. The SQL scripts needed to build the Cape Codd database are available at *www.pearsonhighered.com/kroenke*. Go to the *Database Processing* 14/e Companion Web site, and download the Student Data Files to your *Downloads* folder. There is a ZIP archive file named **DBP-e14-IM-SRC.zip**, so you will need to extract the files. After you have done this, copy the two files in the *Downloads/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder to your *Documents/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder (if you have not already created the *DBP-e14-Cape-Codd-Database* folder in *Schemas*, then copy the entire *Downloads/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder to your *Documents/MySQL Workbench/Schemas* folder).
2. Click the **Open a SQL script file (in a new query tab)** button (note that although we prefer to the syntax "an SQL", MySQL itself uses "a SQL") shown in Figure 10C-18 to display the Open SQL Script dialog box (alternately, we can use the *File | Open SQL Script* menu command to open the dialog box). The advantage of this button is that the script will be opened in a new query tabbed window, not the *Query 1* tabbed window that we currently have open.
3. Browse to the **DBP-e14-MySQL-Cape-Codd-Create-Tables.sql** SQL script as shown in Figure 10C-18.

FIGURE 10C-18

The Open SQL Script Dialog Box



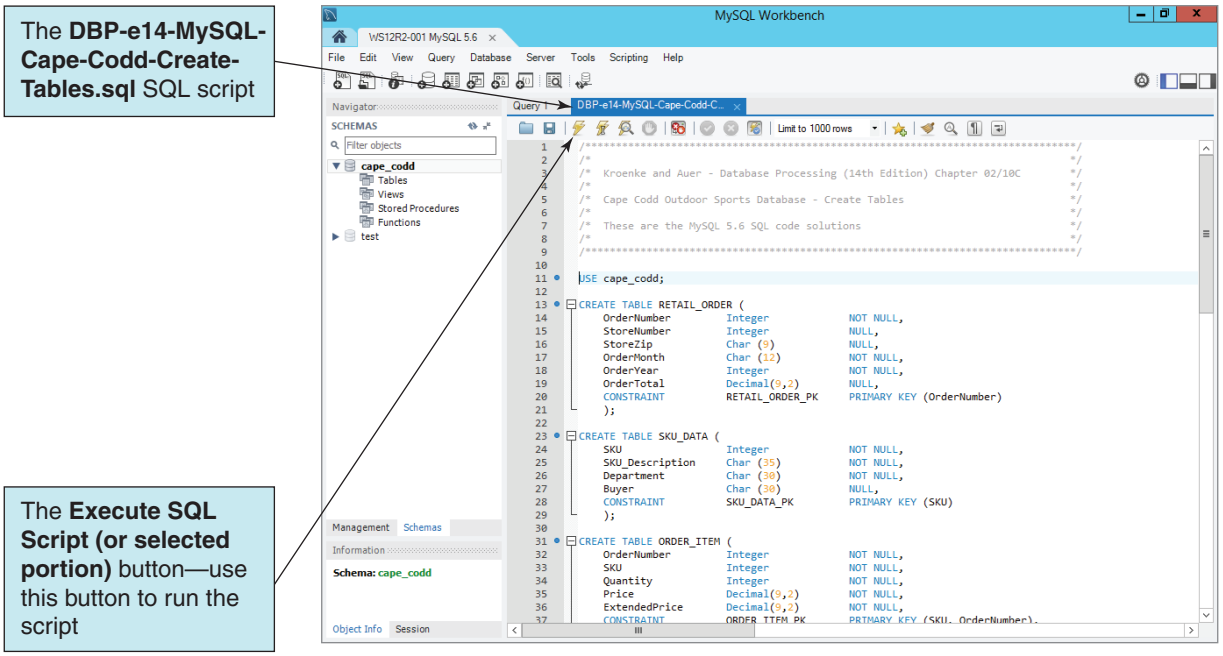


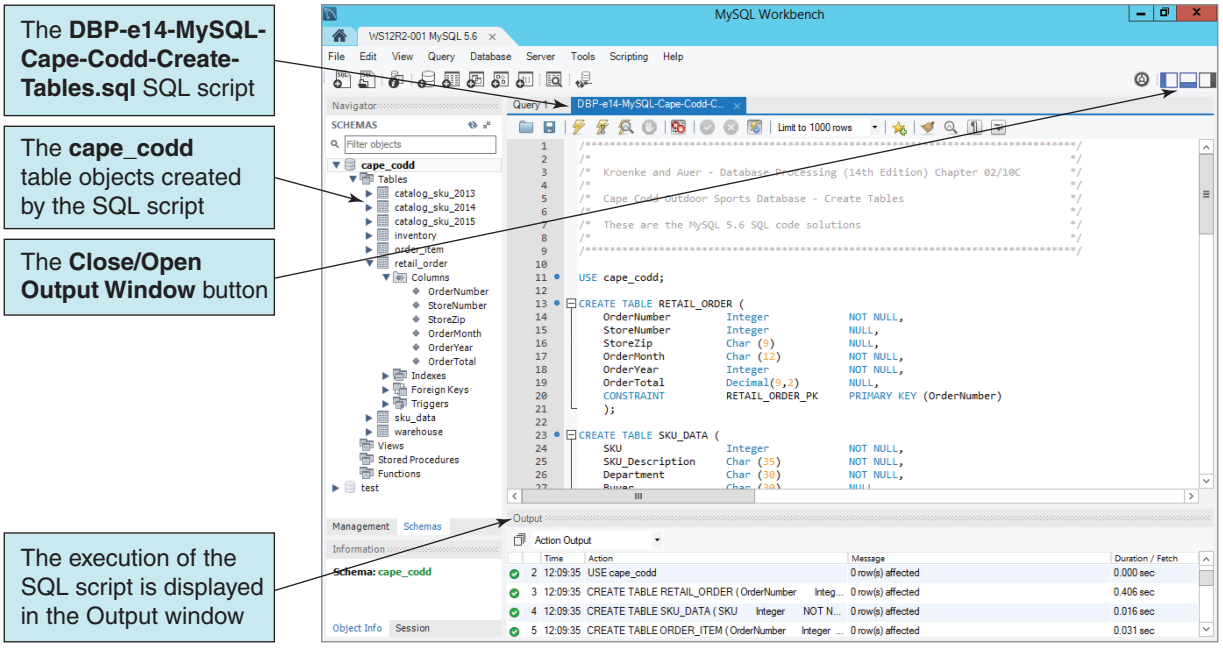
FIGURE 10C-19

The Cape Codd Database Create Tables Script

4. Click the **Open** button. The DBP-e14-MySQL-Cape-Codd-Create-Tables SQL script is displayed in a new SQL query tabbed window, as shown in Figure 10C-19.
5. Click the Execute SQL Script (or selected portion) button. The SQL script is run, and the Cape Codd database tables are created, as shown in Figure 10C-20. Note that we have opened the Output window, which displays the script actions.
6. Click the **Open an SQL script file (in a new query tab)** button to display the Open SQL Script dialog box.
7. Browse to the **DBP-e14-MySQL-Cape-Codd-Insert-Data.sql** SQL script.
8. Click the **Open** button. The DBP-e14-MySQL-Cape-Codd-Insert-Data.sql SQL script is displayed in a new SQL query tabbed window.
9. Click the **Execute SQL Script (or selected portion)** button (we are using an abbreviated button name—on a mouse-over, the button is labeled as *Execute the selected portion of the script or everything, if there is no selection*). The SQL script is run, and the Cape Codd database tables are populated with data, as shown in Figure 10C-21. Note that we have opened the Output window, which displays the script actions.

FIGURE 10C-20

The Cape Codd Database Table Objects in the Navigator



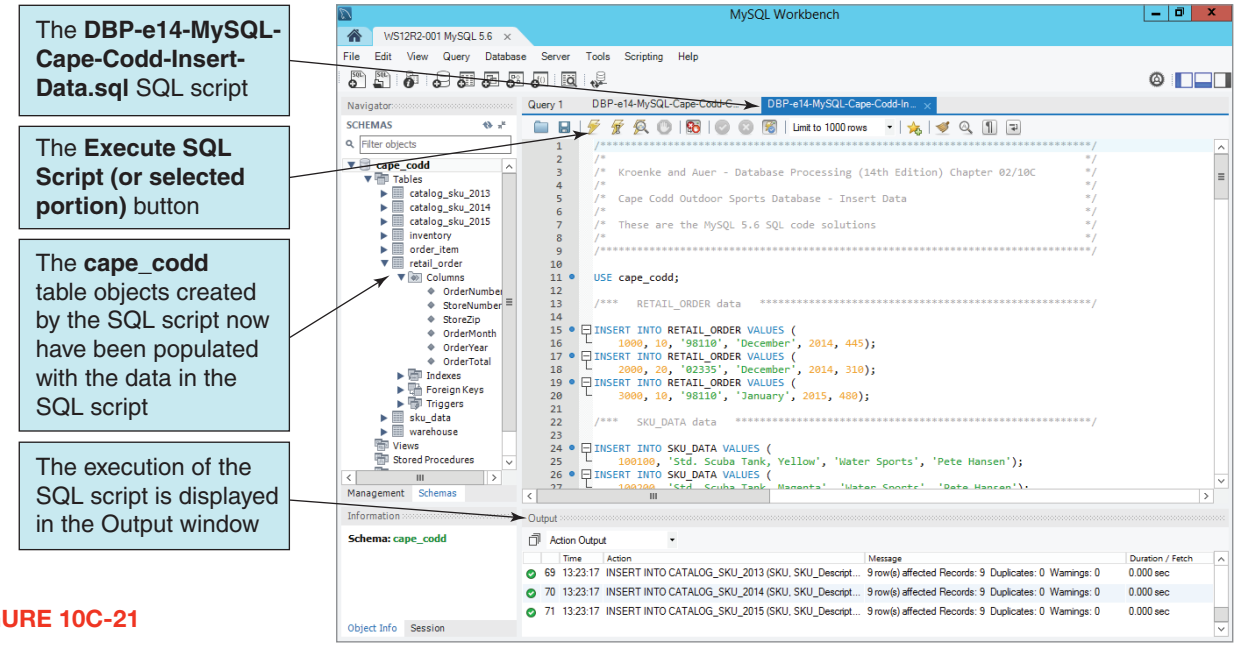


FIGURE 10C-21

Populating the Cape Codd Database Tables

10. In the DBP-e14-MySQL-Cape-Codd-Insert-Data.sql SQL script tabbed window, click the **Close** button to close this tabbed window.
11. In the DBP-e14-MySQL-Cape-Codd-Create-Tables.sql SQL script tabbed window, click the **Close** button to close this tabbed window.
12. We will now test the Cape Codd database by running a query against the database. As discussed in Chapter 2, we do this by using an SQL SELECT statement. We will run the first SQL query demonstrated in Chapter 2, which is:

```
SELECT    SKU, SKU_Description, Department, Buyer
FROM      SKU_DATA;
```

13. As shown in Figure 10C-22, enter the SQL statement for the SQL SELECT statement, and then click the **Execute SQL Statement (under the keyboard cursor)** button (we are using an edited button name—on a mouse-over, the button is labeled as *Execute the statement under the keyboard cursor*). The SQL SELECT statement is run, and the query

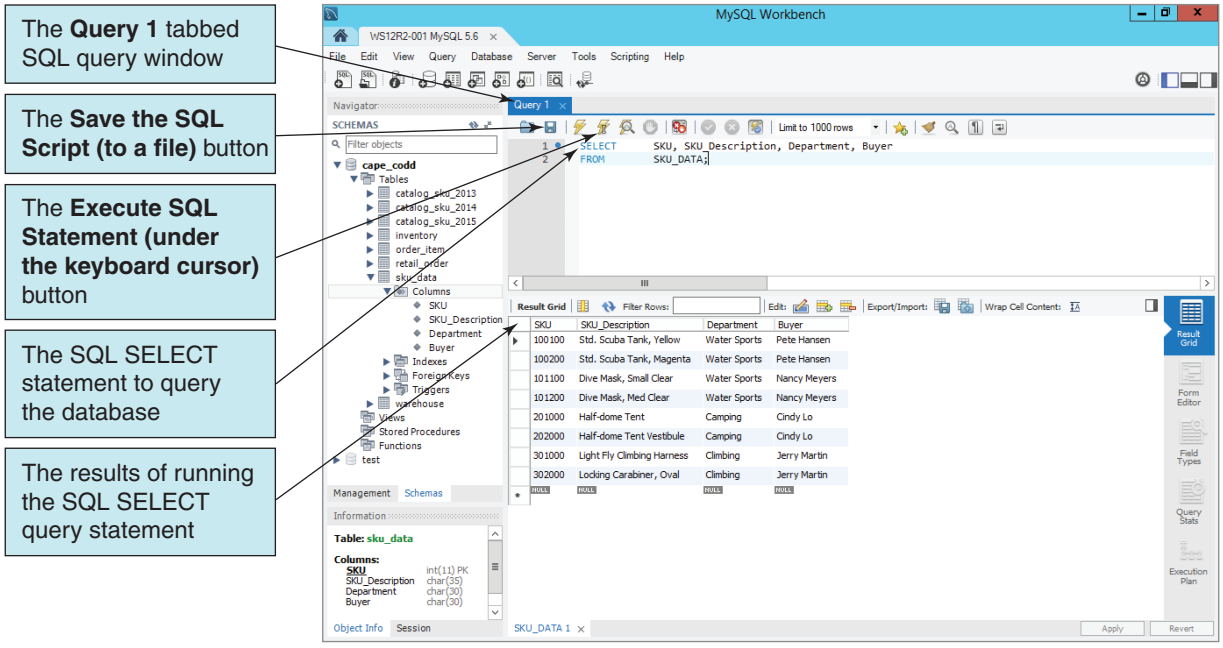


FIGURE 10C-22

Querying the Cape Codd Database

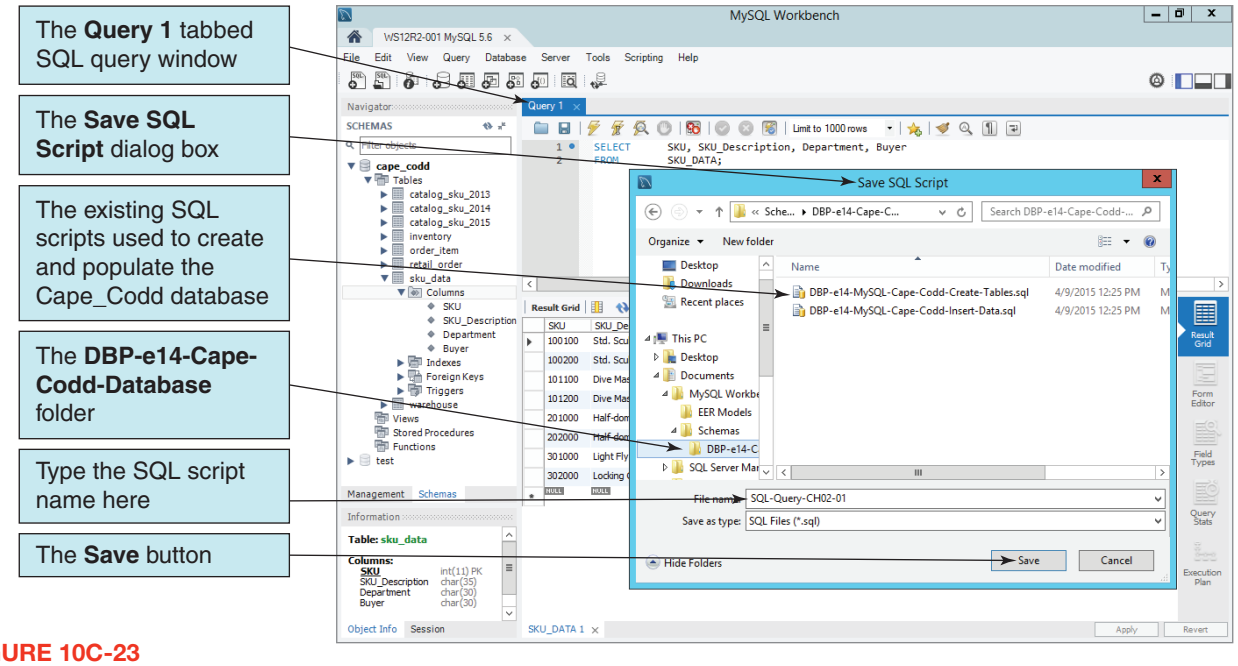


FIGURE 10C-23

Saving the SQL Query as an SQL Script

results are displayed in the SKU_DATA 1 tabbed result grid, as shown at the bottom of Figure 10C-22. (Note that we have closed the Output window).These results confirm that the Cape Codd data was successfully entered into the Cape Codd database.

14. As discussed in Chapter 2, we can save this query as an SQL script for later use. Click the **Save the SQL Script (to a file)** button to open the Save SQL Script dialog box as shown in Figure 10C-23. Browse to the *Documents/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder, and save this SQL query as SQL-Query-CH02-01.sql.
15. In the Query 1 SQL query tabbed window, click the **X [Close]** button to close this tabbed window.

Using a Single SQL Script to Store Multiple SQL Commands

We have now created and queried the Cape Codd Outdoor Sports database used in Chapter 2 for our discussion of SQL query statements. We could save each of the Chapter 2 SQL queries as a separate SQL script, but a more efficient way to store these SQL statements is to combine them in a single SQL script.

We can annotate the SQL script with comments, and, as we will demonstrate, we can run a single SQL statement in the script by selecting that statement and then executing it.

Creating and Using an SQL Script to Store SQL Queries

1. Click the **Open (a script file in this editor)** button to display the Open SQL Script dialog box.
2. Browse to the **SQL-Query-CH02-01.sql** SQL script in the *Downloads/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder.
3. Click the **Open** button. **SQL-Query-CH02-01.sql** SQL script is displayed in the SQL query tabbed window (although this is a new tabbed window, it will appear nearly identical to the tabbed window shown above in Figure 10C-22—the only differences will be that the tab is labeled *SQL-Query-CH02-01* and the query has *not* been executed, so there is no results grid.
4. Edit the SQL script as shown in Figure 10C-24. Note that we are adding one new query (SQL-Query-CH02-02 from Chapter 2), an SQL comment header to identify the script, and individual comments to identify each query.
5. While we could save our work under the same file name, the current name really doesn't describe the SQL script with the changes we have made. We will save it under a new, more descriptive name: *Cape-Codd-Chapter-02-SQL-Queries.sql*.

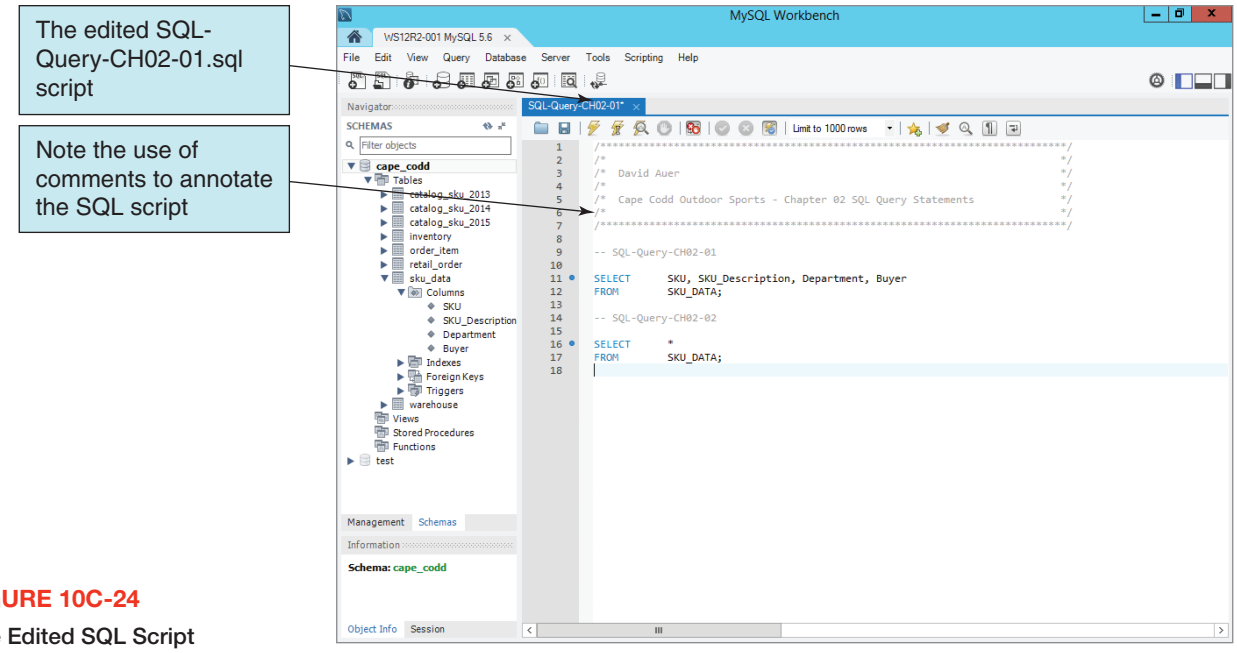
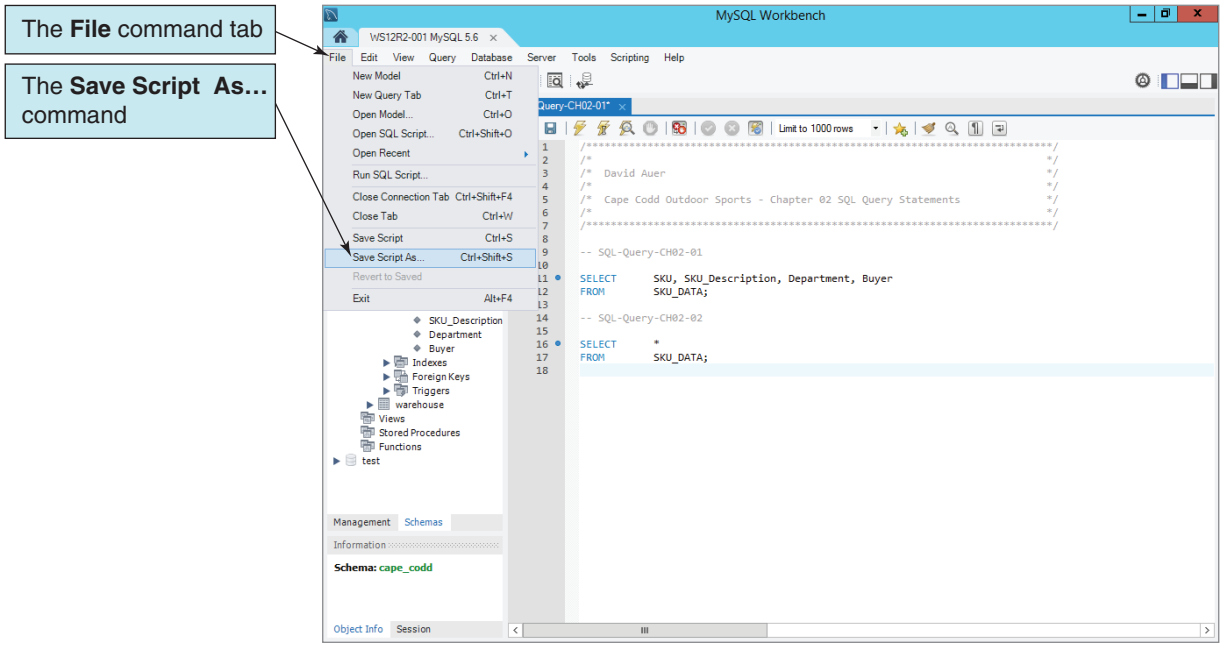


FIGURE 10C-24
The Edited SQL Script

6. Use the **File | Save Script As…** command in the File menu as shown in Figure 10C-25 to display the **Save SQL Script** dialog box.
7. Type in the new SQL script name **Cape-Codd-Chapter-02-SQL-Queries.sql**.
8. Click the **Save** button to save the SQL script under the new file name.
9. As shown in Figure 10C-26, the tabbed window now displays the new file name.
10. As shown in Figure 10C-26, use the mouse cursor to select (highlight) **SQL-Query-CH02-02** (see how convenient comment labels are?).
11. Click the **Execute** button. Only the selected SQL command is executed, and the results are displayed as shown in the tabbed Results grid in Figure 10C-26. This illustrates how to select and run an individual SQL statement in an SQL script that contains many SQL statements.
12. Click the **X [Close]** button to close this tabbed window.

FIGURE 10C-25
The File | Save Script As… Command



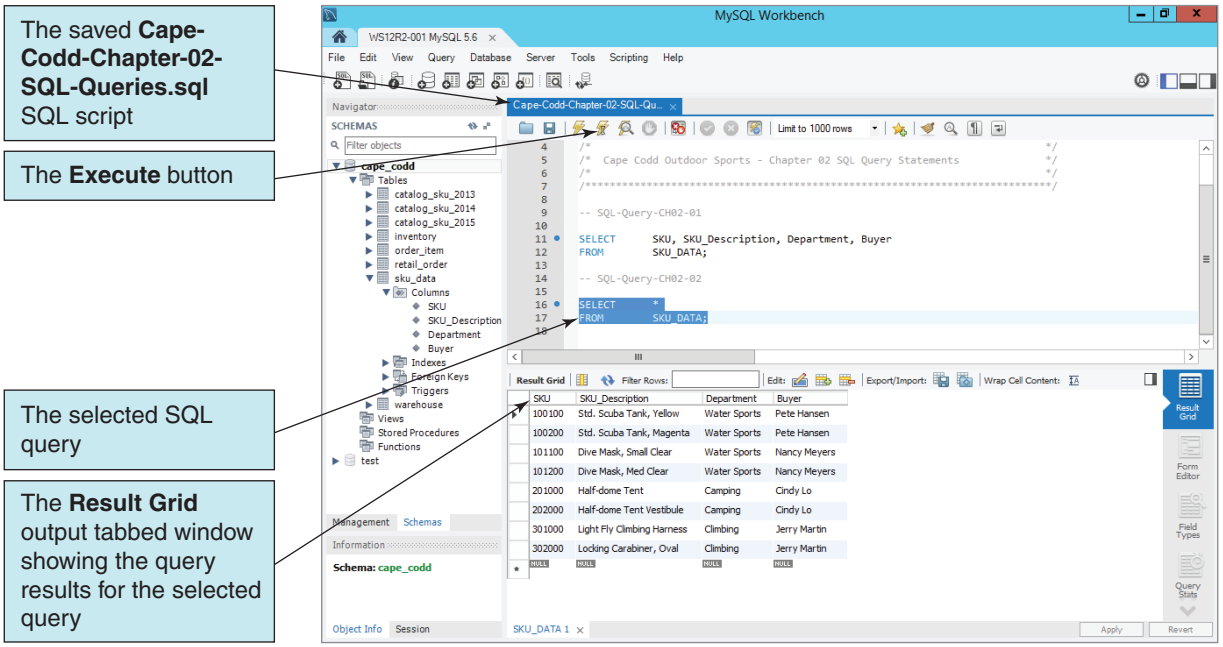


FIGURE 10C-26
The Renamed SQL Script

BY THE WAY

At this point you have covered all the material about MySQL 5.6 that you need to work with the SQL query statements in Chapter 2. If you worked through this material because of the directions in the **Using SQL in Oracle MySQL 5.6** section on pages 58–61, you should return to that section at this time and continue your work on SQL query statements.

Use the new **Cape-Codd-Chapter-02-SQL-Queries.sql** script to store all your work in Chapter 02—that will be much easier and more efficient than storing a separate SQL script for each query!

Implementing the View Ridge Gallery *VRG* Database in MySQL 5.6

Now that we know how to use existing SQL scripts, and how to create and save SQL scripts for SQL query statements, we will discuss how to use SQL scripts to create and populate database tables of our own. To illustrate this, we will use the View Ridge Gallery *VRG* database introduced in Chapter 6 in our discussion of *database designs*, and used as our example of *database implementation* in Chapter 7. In this chapter, we will discuss the specific implementation of the *VRG* database in MySQL 5.6, and use that implementation to introduce some topics not covered in Chapter 7.

BY THE WAY

Because the VRG database example we use in Chapter 7 and this chapter is fairly complex, complete SQL scripts to create the VRG tables and populate them with data are available at the book's Web site at *www.pearsonhighered.com/kroenke*. These scripts will allow you to build the basic VRG database, and then actually try out the VRG database SQL code examples in these chapters. You will still need to read and understand the discussions of the SQL code for these two scripts to be sure you understand all the underlying concepts.

As we have seen, tables and other MySQL structures can be created and modified in two ways. The first is to write SQL code using either the CREATE or ALTER SQL statements we discussed in Chapter 7. The second is to use the MySQL GUI display tools discussed earlier in this chapter. Although either method will work, CREATE statements are preferred for the reasons described in Chapter 7. Some professionals choose to create structures via SQL but then modify them with the GUI tools.

Each DBMS product has its own variant of SQL, and each variant usually includes procedural extensions. Unlike some other vendors, however, MySQL has no product-specific name for its variant of SQL—it is just called SQL! Nonetheless, MySQL does implement a standard set of SQL extensions called **SQL/Persistent Stored Modules (SQL/PSM)**, and MySQL's version of SQL is thus sometimes referred to by the ANSI/ISO SQL Standard name of **SQL/PSM**. We will point out specific MySQL SQL syntax as we encounter it in our discussion. For more on SQL for MySQL, see the MySQL 5.6 Documentation "Chapter 13. SQL Statement Syntax" at *http://dev.mysql.com/doc/refman/5.6/en/sql-syntax.html*.

First, we need to create the *VRG* database itself.

Creating the VRG Database

1. Click the **Add Schema** button.
2. Type the new database (schema) name *VRG* in the **Name** textbox, and then click the **Apply** button. Because we used *capital (uppercase) letters* in our schema name, the Apply Changes to Object warning dialog box is displayed. By default, MySQL is configured to allow only *lower case* letters in database names. While we could change this, we prefer to use as many MySQL defaults as possible, and so we will acknowledge the warning and allow MySQL to implement the database as *vrg*.
3. The Apply SQL Script to Database dialog box is displayed with the results of executing the SQL command. Click the **Finish** button.
4. The new_schema tabbed dialog box is displayed again renamed as vrg_schema. Click the **Close** button.
5. Click the **Refresh** button—the databases are sorted into alphabetical order and the *vrg* database object is displayed in the Navigator, as shown in Figure 10C-27.
6. Set the *vrg* database as the default database, and then expand the *vrg* database object to display the *vrg* database folders, also shown in Figure 10C-27.

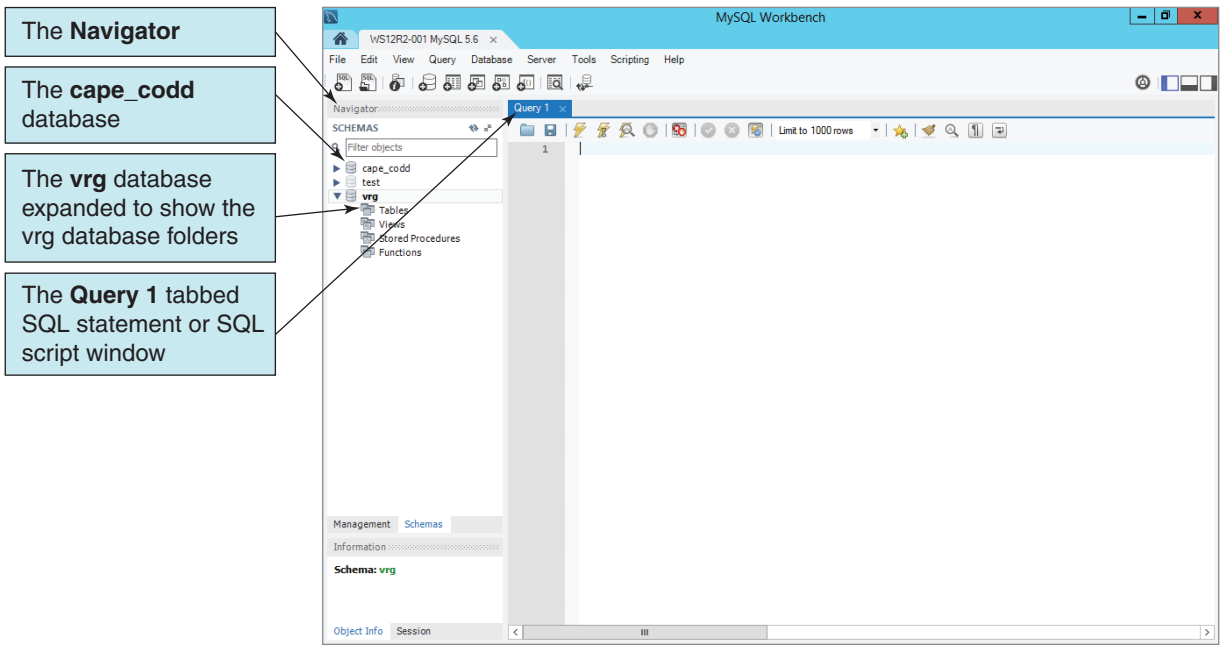
Using SQL Scripts to Create and Populate Database Tables

Now that we have created the VRG database, we will set up a folder in the *MySQL Workbench/Schemas* folder to store our SQL scripts, and review creating and saving an SQL script.

Creating and Saving an SQL Script to Create the VRG Tables

1. MySQL Workbench always keeps an SQL query window open. If we close all our open tabbed windows, it will automatically open a new SQL query window. When

FIGURE 10C-27
The VRG Database in the Navigator



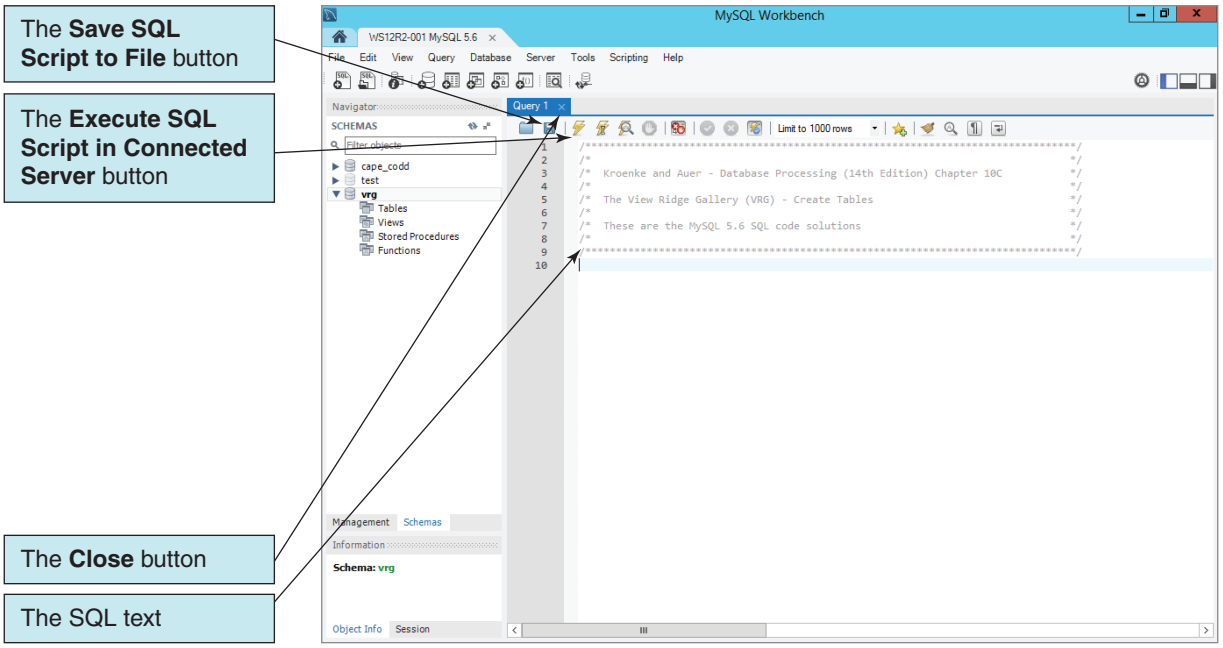


FIGURE 10C-28

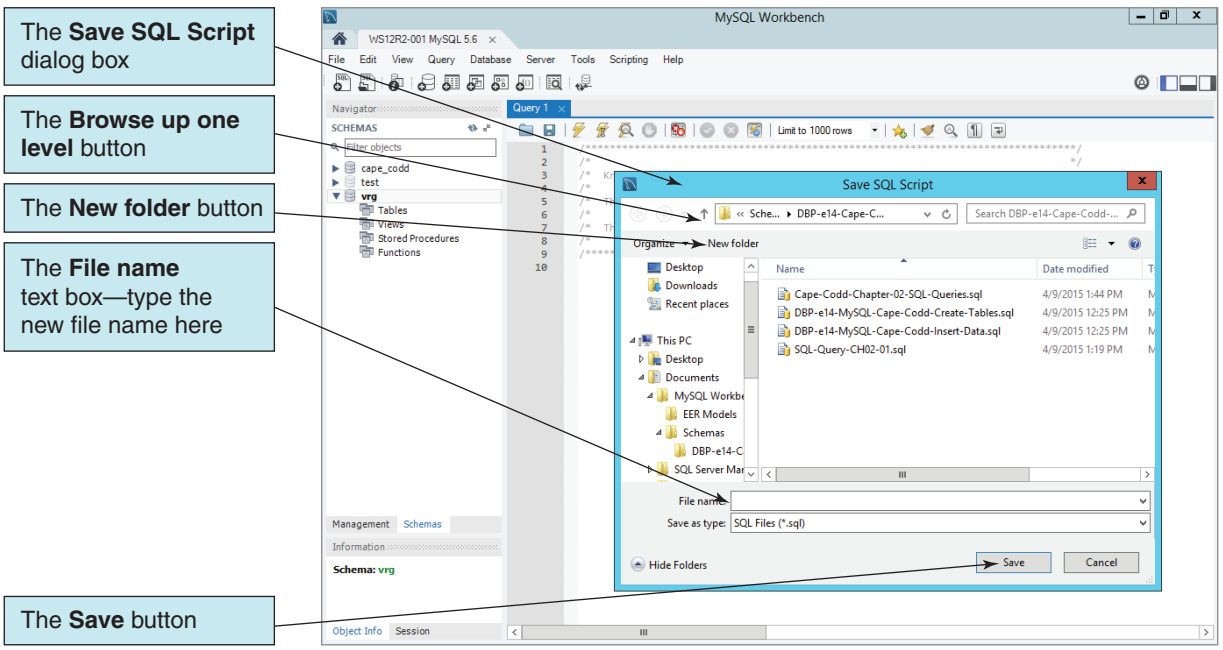
The VRG Database Script in the MySQL Workbench

we closed the Cape-Codd-Chapter-02-SQL-Queries.sql window, it opened a tabbed window name Query 1, as shown in Figure 10C-27.

2. In the open tabbed SQL Query window, type the SQL comments shown in Figure 10C-28.
3. Click the **Save SQL Script to File** button. The Save SQL Script dialog box is displayed, as shown in Figure 10C-29.
4. The Save SQL Script dialog box opens to the *Documents/MySQL Workbench/Schemas/DBP-e14-Cape-Codd-Database* folder. Browse up one level to the *Documents/MySQL Workbench/Schemas* folder.
5. Click the **New folder** button in the Save SQL Script dialog box. A new folder object is displayed.
6. Type the folder name **DBP-e14-View-Ridge-Gallery-Database** as the new folder name.

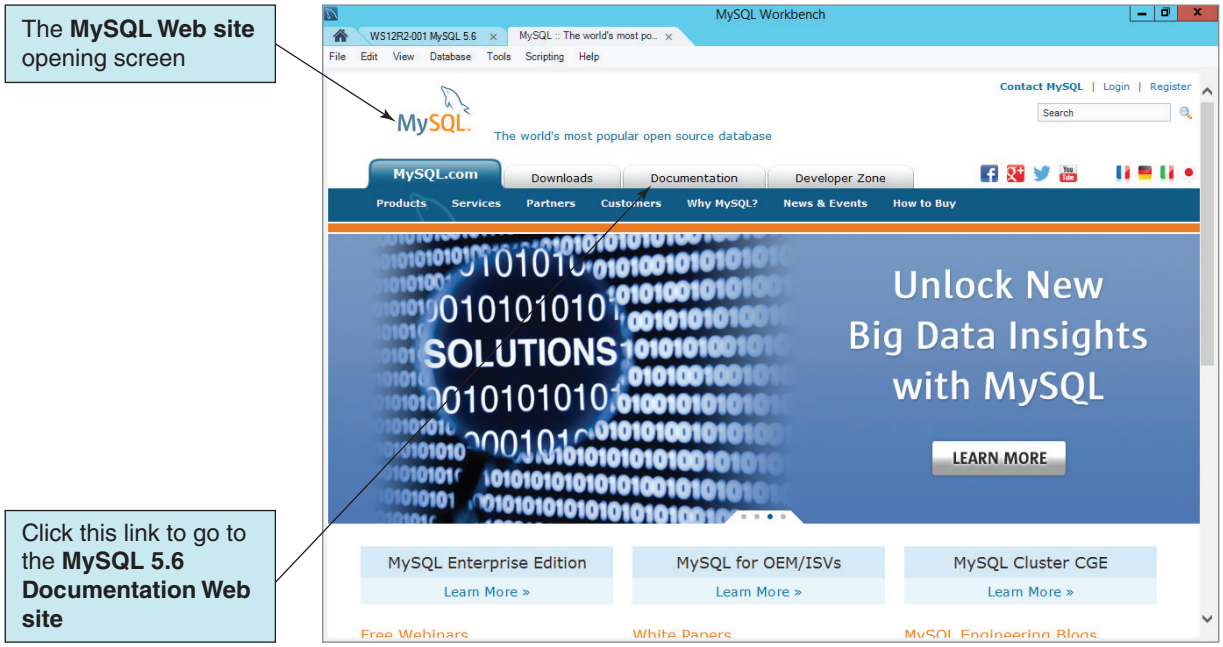
FIGURE 10C-29

Saving the SQL Script

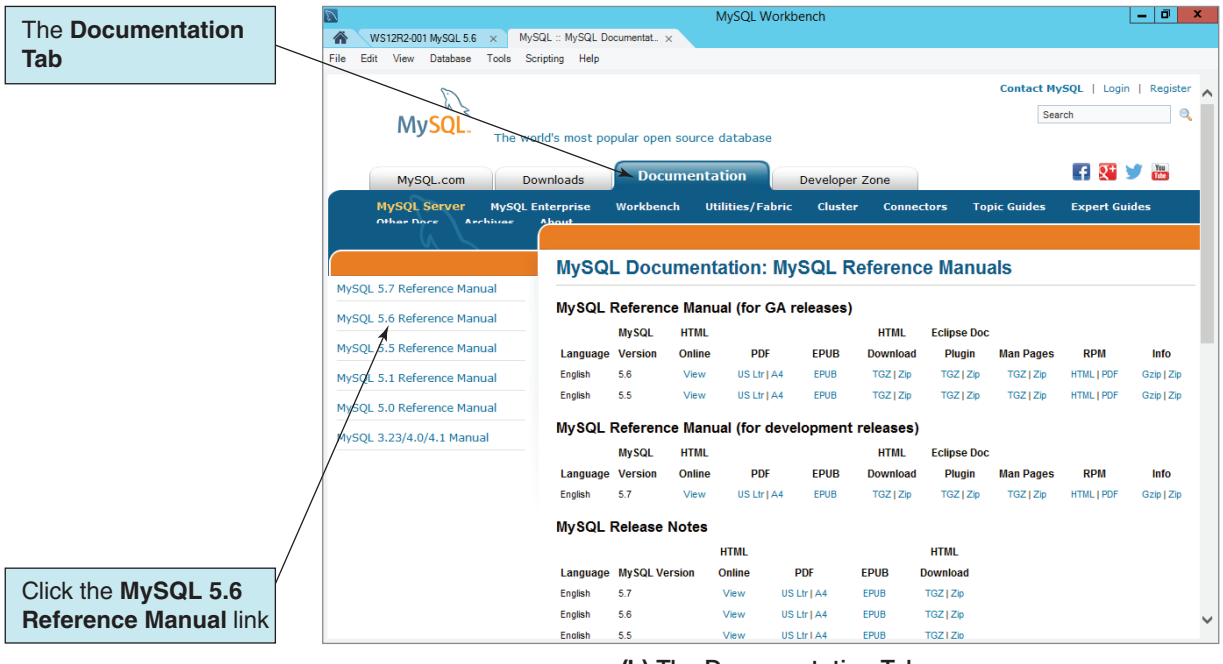


7. Click the **Open folder** button in the Save Script to File dialog box. The new folder is created.
8. Type the file name **VRG-Create-Tables** in the File name text box of the Save Script to File dialog box.
9. Click the **Save** Button on the Save Script to File dialog box. The script is saved.
   - One of the nice features of the MySQL Workbench is that we can open a documentation Web browser as a **tabbed MySQL.com Web site window** by using the **Help | MySQL.com Web site** command, as shown in Figure 10C-30(a). To get to the MySQL 5.6 documentation, click the Documentation tab to display the MySQL Documentation: MySQL Reference Manuals page as shown

FIGURE 10C-30
The MySQL Workbench Help command

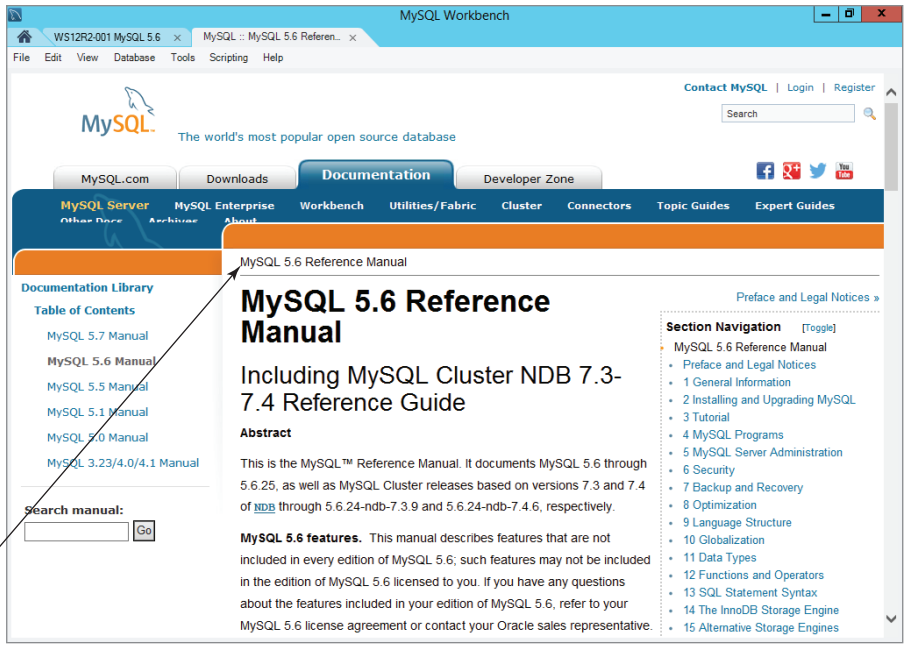


(a) The Help | MySQL.com Website



(b) The Documentation Tab

(*continued*)



The **MySQL 5.6 Reference Manual** opening screen

(c) The MySQL 5.6 Reference Manual

FIGURE 10C-30
Continued

in Figure 10C-30(b). On the Documentation tab, click MySQL 5.6 Reference Manual, which is displayed as shown in Figure 10C-30(c). Click the **X [Close]** button to close the MySQL 5.6 Reference Manual.

Creating the View Ridge Database Table Structure

The MySQL 5.6 version of the SQL CREATE TABLE statements for the View Ridge Gallery database in Chapter 7 is shown in Figure 10C-31. Unlike some other DBMS products, TRANSACTION is not a **reserved word** in MySQL 5.6,[4] but it is an ODBC reserved word. Therefore, and for consistency within this book, we will continue to use the table name TRANS instead of TRANSACTION in the VRG database.

MySQL supports surrogate keys, and the surrogate key columns are created using the **AUTO_INCREMENT attribute** with the primary key. Note that by default AUTO_INCREMENT starts at 1 and then increments by adding 1 to the previous surrogate key value each time a new key is created. We can change the starting value by using an ALTER TABLE statement, but you cannot change the value of the increment. This somewhat limits the usefulness of AUTO_INCREMENT, but it should be usable in many cases.

Unfortunately, AUTO_INCREMENT does not work well when we are trying to enter data with nonsequential surrogate keys. This is exactly the case with the View Ridge Gallery database data–for example, see the VRG CUSTOMER table data in Figure 7-15(a). In this case, we need to create the tables without AUTO_INCREMENT, insert the existing data, and then alter the tables to use AUTO_INCREMENT starting at the next available surrogate key value.

Creating the VRG Table Structure Using SQL Statements

1. The **VRG-Create-Tables.sql** script should till be open in MySQL Workbench. If not, open it.
2. Type in the SQL statements shown in Figure 10C-31 (this SQL script is available on the book's Web site at *www.pearsonhighered.com/kroenke*). Be sure to save the script often, and when you have completed entering all the SQL statements, save the script a final time.
3. Scroll to the top of the script.

[4]For a complete list of MySQL 5.6 reserved keywords, see the MySQL 5.6 documentation "Reserved Words in MySQL 5.6" at *http://dev.mysql.com/doc/mysqld-version-reference/en/mysqld-version-reference-reservedwords-5-6.html*.

```
/*************************************************************************************/
/*                                                                                   */
/*     Kroenke and Auer - Database Processing (14th Edition) Chapter 10C             */
/*                                                                                   */
/*     The View Ridge Gallery Database (VRG) - Create Tables                         */
/*                                                                                   */
/*     These are the MySQL 5.6 SQL code solutions                                    */
/*                                                                                   */
/*************************************************************************************/


CREATE TABLE ARTIST(
       ArtistID            Int              NOT NULL,
       LastName            Char(25)         NOT NULL,
       FirstName           Char(25)         NOT NULL,
       Nationality         Char(30)         NULL,
       DateOfBirth         Numeric(4,0)     NULL,
       DateDeceased        Numeric(4,0)     NULL,
       CONSTRAINT    ArtistPK            PRIMARY KEY(ArtistID),
       CONSTRAINT    ArtistAK1           UNIQUE(LastName, FirstName),
       CONSTRAINT    NationalityValues   CHECK
                           (Nationality IN ('Canadian', 'English', 'French',
                            'German', 'Mexican', 'Russian', 'Spanish',
                            'United States')),
       CONSTRAINT    BirthValuesCheck    CHECK (DateOfBirth < DateDeceased),
       CONSTRAINT    ValidBirthYear      CHECK
                           (DateOfBirth LIKE '[1-2][0-9][0-9][0-9]'),
       CONSTRAINT    ValidDeathYear      CHECK
                           (DateDeceased LIKE '[1-2][0-9][0-9][0-9]')
       );

CREATE TABLE WORK(
       WorkID              Int              NOT NULL,
       Title               Char(35)         NOT NULL,
       Copy                Char(12)         NOT NULL,
       Medium              Char(35)         NULL,
       Description         Varchar(1000)    NULL DEFAULT 'Unknown provenance',
       ArtistID            Int              NOT NULL,
       CONSTRAINT    WorkPK              PRIMARY KEY(WorkID),
       CONSTRAINT    WorkAK1             UNIQUE(Title, Copy),
       CONSTRAINT    ArtistFK            FOREIGN KEY(ArtistID)
                              REFERENCES ARTIST(ArtistID)
                                 ON UPDATE NO ACTION
                                 ON DELETE NO ACTION
       );

CREATE TABLE CUSTOMER(
       CustomerID          Int              NOT NULL,
       LastName            Char(25)         NOT NULL,
       FirstName           Char(25)         NOT NULL,
       EmailAddress        Varchar(100)     NULL,
       EncryptedPassword   Varchar(50)      NULL,
       Street              Char(30)         NULL,
       City                Char(35)         NULL,
       State               Char(2)          NULL,
       ZIPorPostalCode     Char(9)          NULL,
       Country             Char(50)         NULL,
       AreaCode            Char(3)          NULL,
       PhoneNumber         Char(8)          NULL,
       CONSTRAINT    CustomerPK          PRIMARY KEY(CustomerID),
       CONSTRAINT    EmailAdressAK1      UNIQUE(EmailAddress)
       );
```

FIGURE 10C-31

The SQL Statements to Create the VRG Table Structure

(continued)

FIGURE 10C-31
Continued

```
CREATE TABLE TRANS(
       TransactionID       Int              NOT NULL,
       DateAcquired        Date             NOT NULL,
       AcquisitionPrice    Numeric(8,2)     NOT NULL,
       DateSold            Date             NULL,
       AskingPrice         Numeric(8,2)     NULL,
       SalesPrice          Numeric(8,2)     NULL,
       CustomerID          Int              NULL,
       WorkID              Int              NOT NULL,
       CONSTRAINT    TransPK             PRIMARY KEY(TransactionID),
       CONSTRAINT    TransWorkFK         FOREIGN KEY(WorkID)
                           REFERENCES WORK(WorkID)
                              ON UPDATE NO ACTION
                              ON DELETE NO ACTION,
       CONSTRAINT    TransCustomerFK     FOREIGN KEY(CustomerID)
                           REFERENCES CUSTOMER(CustomerID)
                              ON UPDATE NO ACTION
                              ON DELETE NO ACTION,
       CONSTRAINT    SalesPriceRange  CHECK
                          ((SalesPrice > 0) AND (SalesPrice <=500000)),
       CONSTRAINT    ValidTransDate      CHECK (DateAcquired <= DateSold)
       );

CREATE TABLE CUSTOMER_ARTIST_INT(
       ArtistID            Int              NOT NULL,
       CustomerID          Int              NOT NULL,
       CONSTRAINT    CAIntPK             PRIMARY KEY(ArtistID, CustomerID),
       CONSTRAINT    CAInt_ArtistFK      FOREIGN KEY(ArtistID)
                           REFERENCES ARTIST(ArtistID)
                              ON UPDATE NO ACTION
                              ON DELETE CASCADE,
       CONSTRAINT    CAInt_CustomerFK    FOREIGN KEY(CustomerID)
                           REFERENCES CUSTOMER(CustomerID)
                              ON UPDATE NO ACTION
                              ON DELETE CASCADE
       );
```

4. Click the **Execute SQL Script in Connected Server** button shown in Figure 10C-19. The tables are created, and the VRG tables appear as shown in Figure 10C-32.
5. Click the script window **Close** button to close the SQL script.

Reviewing Database Structures in the MySQL GUI Display

Now we have created the VRG table and relationship structure. After building the table structure using SQL statements, we can inspect the results using the MySQL GUI Tools. Let's take a look at the WORK table, particularly at the properties of the WorkID primary key.

Viewing the WORK Table Structure in the GUI Display

1. There are two ways to display the WORK table structure. As shown in Figure 10C-33(a), when we click the **WORK** table object to select it in the Navigator, an **Alter Table** button becomes available.
2. The other option is shown in Figure 10C-33(b), and this is the option we will use in this example. In the Navigator, click the **WORK** table object to select it, and then right-click it again to open the shortcut menu.
3. In the table shortcut menu, click the **Alter Table...** command. The WORK table design is displayed in the tabbed MySQL Table Editor, as shown in Figure 10C-34. In the MySQL Table Editor, click the **Columns** tab, and then select the **WorkID** row. The MySQL Table Editor appears as shown in Figure 10C-34, with the Column properties selected.
4. Click the MySQL Table Editor window **Close** button to close the MySQL Table Editor.

The MySQL GUI Tools do not include the functionality to create a database diagram for the VRG database. We might want to do this, for example, to ensure that the relationships

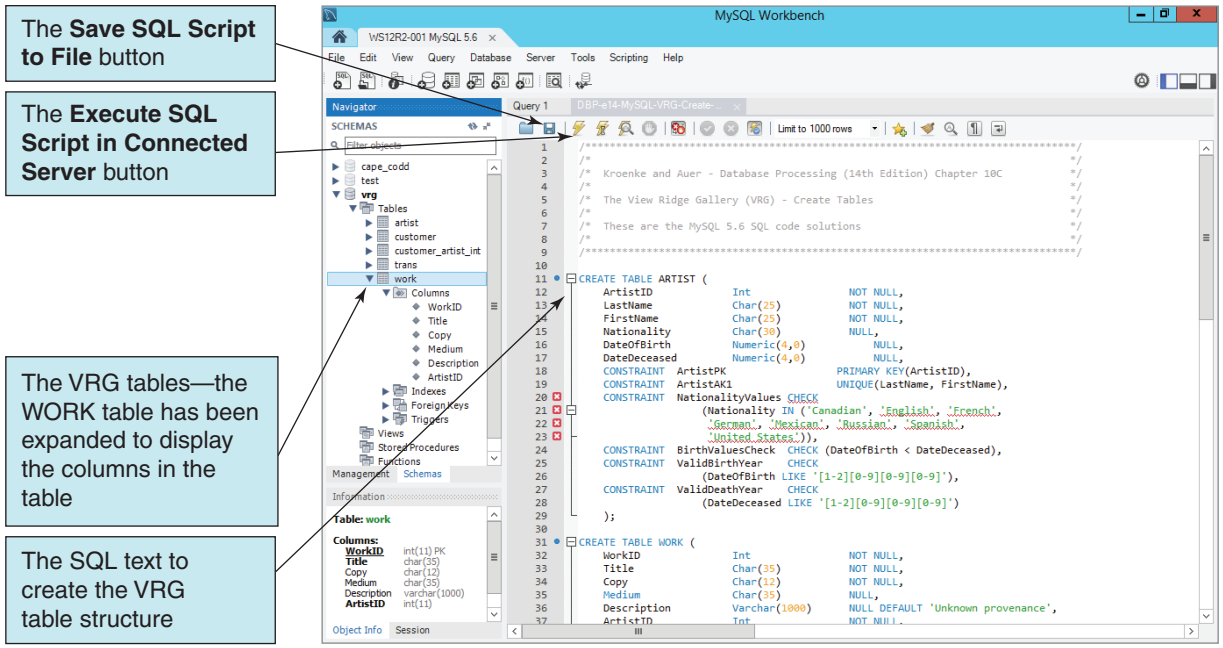


FIGURE 10C-32
The VRG Database Tables

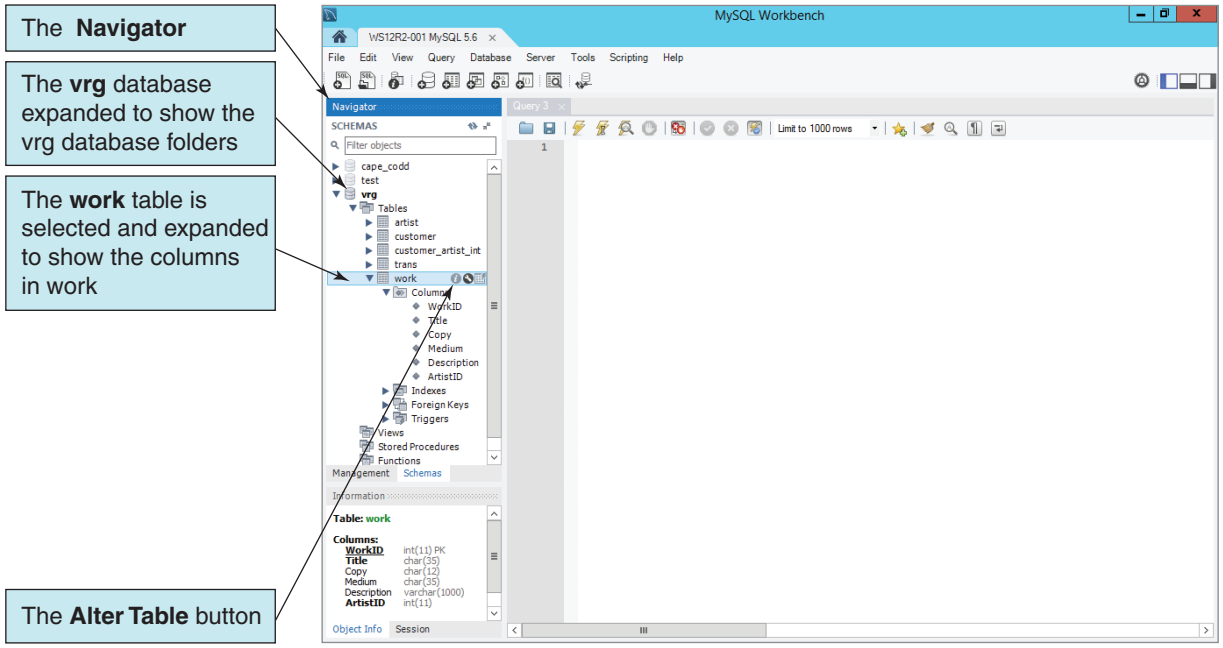
were created correctly. One possibility is to use the MySQL Workbench to reverse-engineer a database design from the database.

We can check foreign keys by using the MySQL Table Editor. For example, let's check the relationship between WORK and ARTIST.

Viewing Relationship Foreign Key Properties

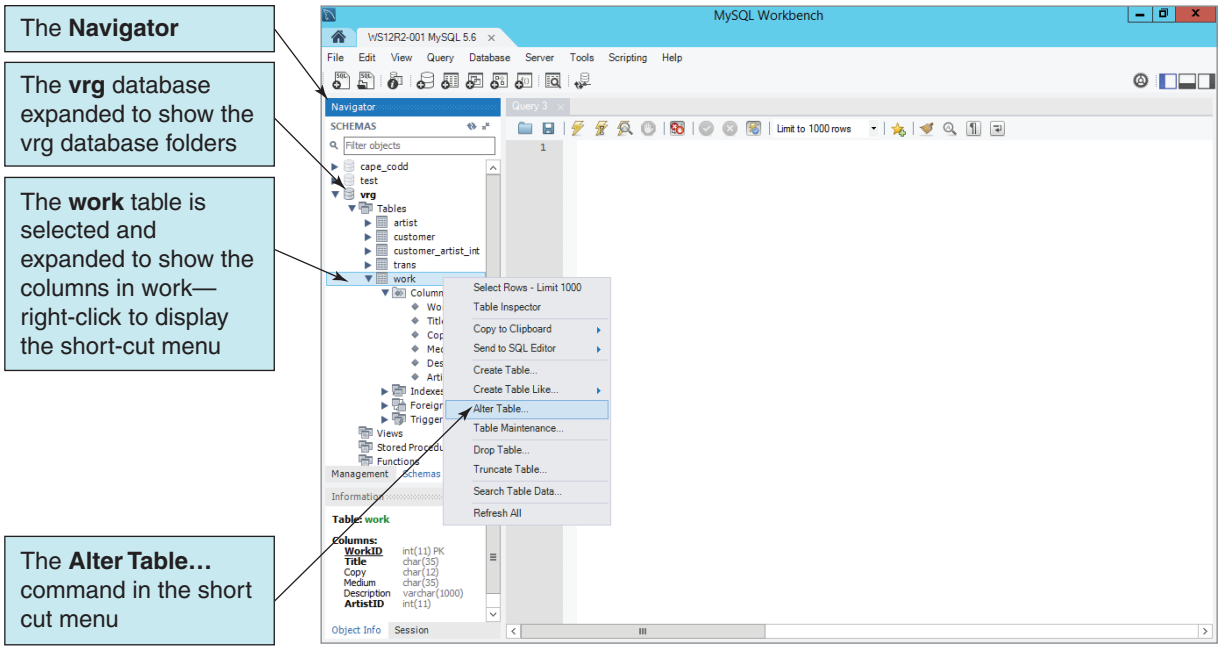
1. In the MySQL Workbench Object Browser, click the **WORK** table object to select it, and then right-click it again to open the shortcut menu.
2. In the table shortcut menu, click the **Alter Table…** command. The WORK table design is displayed in the MySQL Table Editor. Select the **ArtistFK** row.

FIGURE 10C-33
The Alter Table Command



(a) The Alter Table Button

(continued)



(b) The Alter Table… Command

FIGURE 10C-33
Continued

3. Click the **Foreign Keys** tab at the bottom of the screen to display the foreign key settings for the WORK table, as shown in Figure 10C-35. Note that the proper settings are shown, including the lack of cascading updates and deletes.
4. Click the MySQL Table Editor **Close** button to close the MySQL Table Editor.

Indexes

As discussed in Appendix G–*Data Structure for Database Processing*, an **index** is a special data structure that is created to improve database performance. MySQL automatically creates an index on all primary keys and foreign keys and on UNIQUE constraints. Additionally, full-text and spatial indexes can be created, and a developer can also

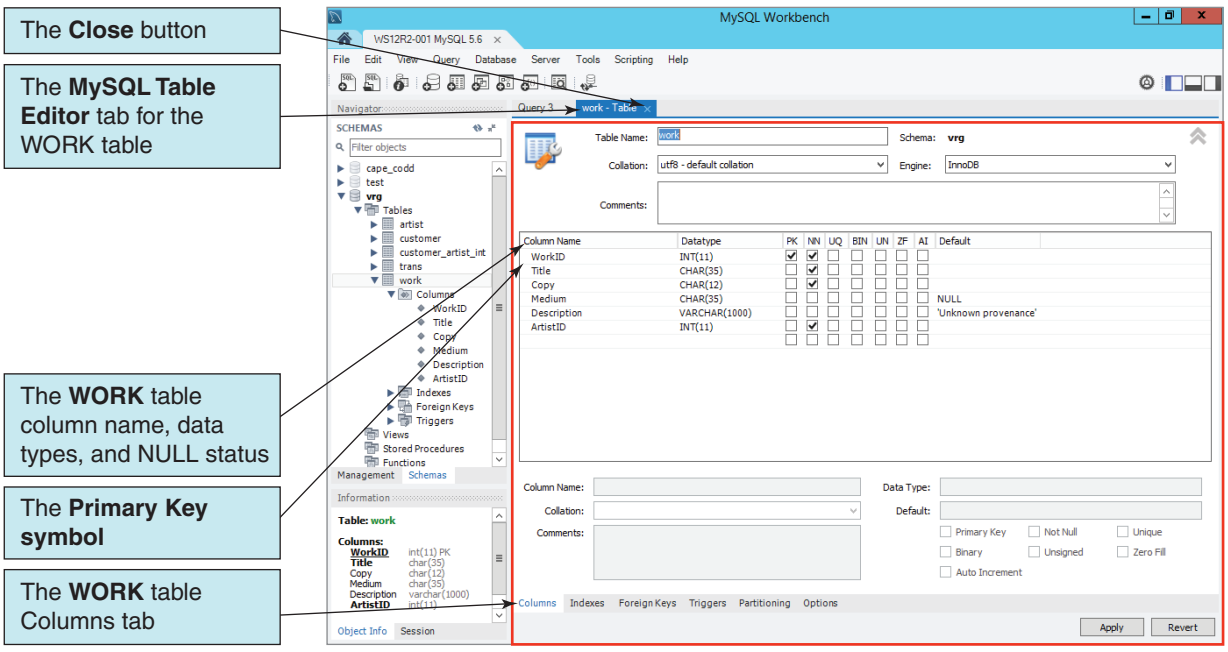


FIGURE 10C-34
The WORK Table Columns and Column Properties

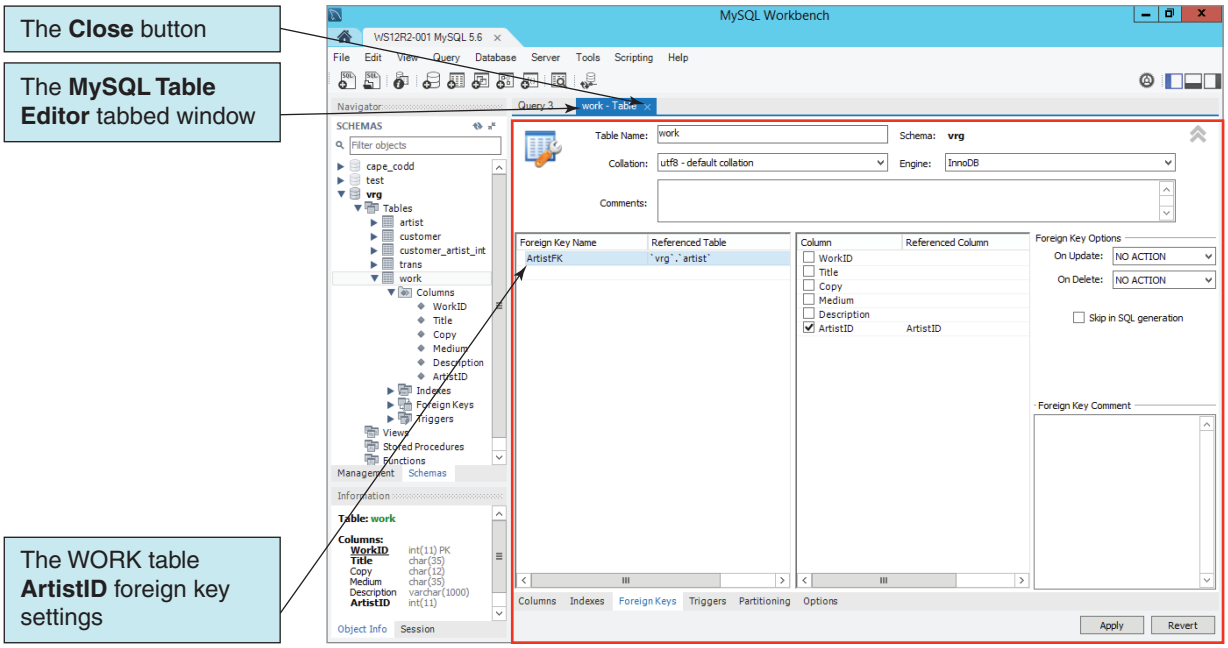


FIGURE 10C-35

The WORK Table Foreign Key Settings

direct MySQL to create an index on other columns that are frequently used in WHERE clauses or on columns that are used for sorting data when sequentially processing a table for queries and reports. MySQL supports B-tree, Hash, and R-tree index structures for columns. For more information, see the MySQL 5.6 Documentation "Chapter 8.3. Optimization and Indexes" at *http://dev.mysql.com/doc/refman/5.6/en/optimization-indexes.html*. We can view the index data in the MySQL Workbench.

Viewing Relationship Index Properties

1. In the MySQL Workbench Object Browser, click the **WORK** table object to select it, and then right-click it again to open the shortcut menu.
2. In the table shortcut menu, click the **Alter Table…** command. The WORK table design is displayed in the MySQL Table Editor.
3. Click the **Indexes** tab to display the foreign key settings for the WORK table, and then select the **PRIMARY** row, as shown in Figure 10C-36. Note that the proper settings are shown, including the lack of cascading updates and deletes.
4. Click the MySQL Table Editor **Close** button to close the MySQL Table Editor.

We can also create new indexes in the MySQL Workbench. We will create an index for the CAInt_ArtistFK foreign key constraint (i.e., the values of ArtistID). We will create that index using the MySQL Table Editor.

Creating a New Index Using the MySQL Table Editor

1. In the MySQL Workbench, make sure that VRG is the default schema, and then expand VRG so the table objects are visible.
2. Click the **CUSTOMER_ARTIST_INT** table object to select it.
3. Right-click the **CUSTOMER_ARTIST_INT** table object to display the shortcut menu.
4. In the table shortcut menu, click the **Alter Table…** command. The CUSTOMER_ARTIST_INT table design is displayed in the MySQL Table Editor.
5. Click the **Indexes** tab to display the foreign key settings for the CUSTOMER_ARTIST_INT table, and note that a new Index Name text box is open and active, as shown in Figure 10C-37.

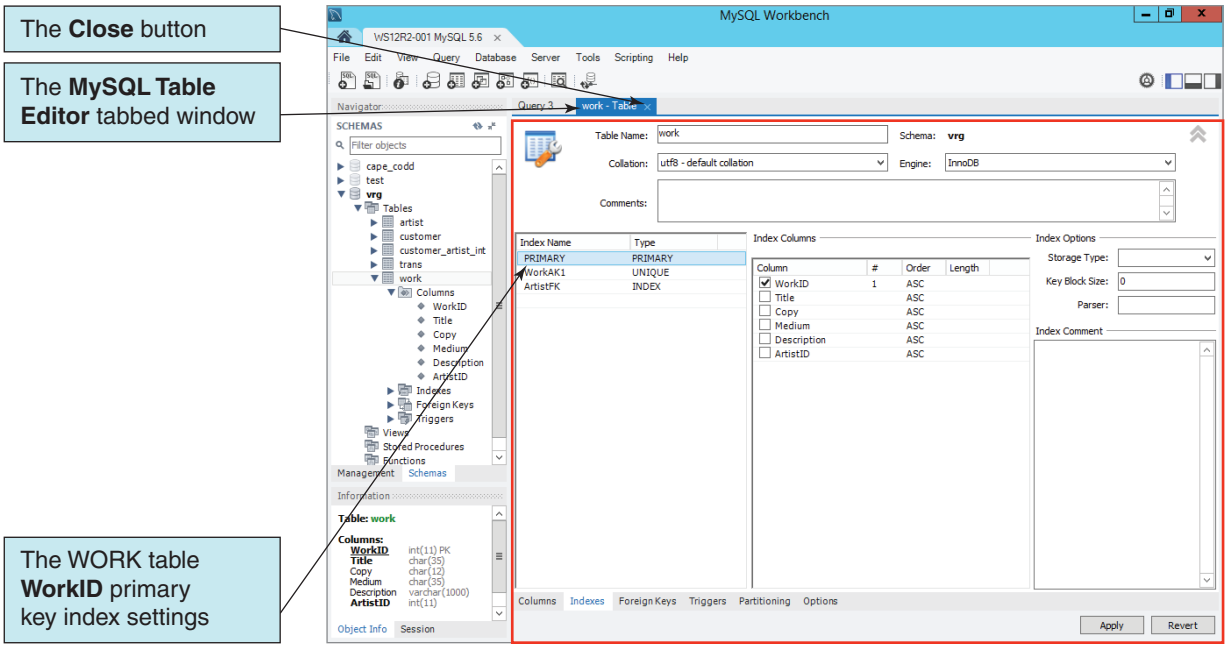


FIGURE 10C-36

The WORK Table Primary Key Index Settings

6. Type the name **CAInt_ArtistFK** in the Index Name text box.
7. Press the **Enter** key. The Type drop-down list is displayed, as shown in Figure 10C-38.
8. Select the index type **INDEX** from the Type drop-down list.
9. In the Index Columns section, check the check box next to the column name **ArtistID**, as shown in Figure 10C-39.
10. Click the **Apply** button. As shown in Figure 10C-40(a), the Apply SQL Script to Database dialog box *Review SQL Script* page is displayed so the user can review the SQL command before it is executed.
11. Click the **Apply SQL** button. As shown in Figure 10C-40(b), The Apply SQL Script to Database dialog box *Apply SQL Script* page is displayed with the results of executing the SQL command.

FIGURE 10C-37

Creating an Index in the MySQL Table Editor

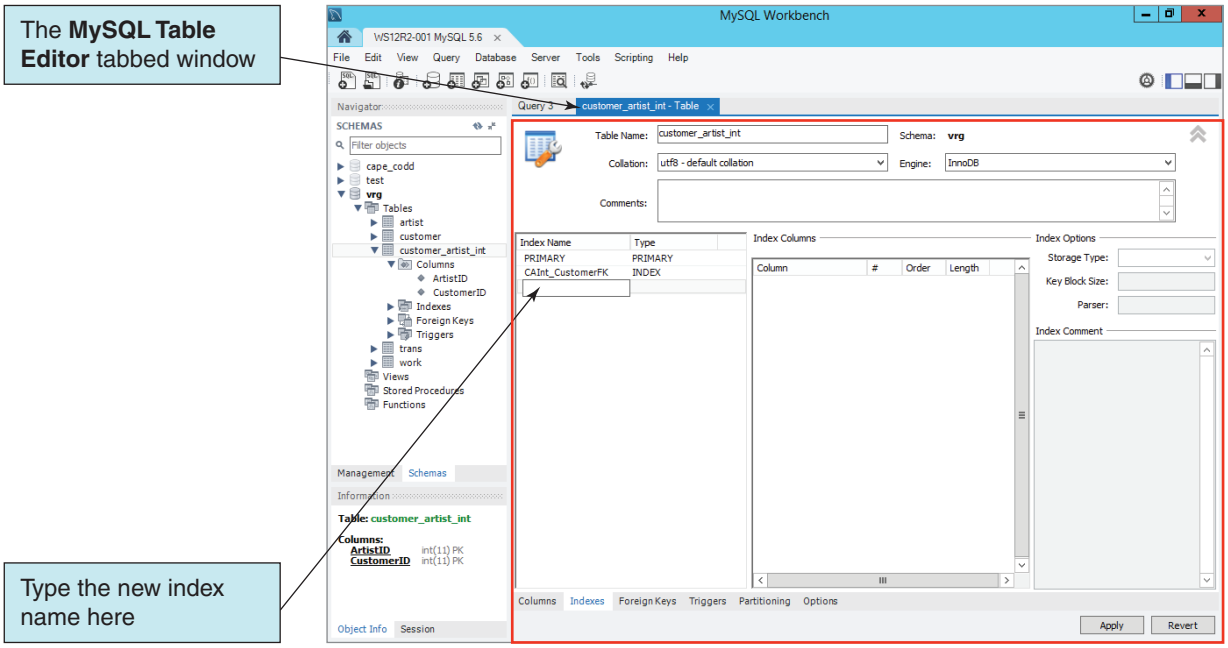
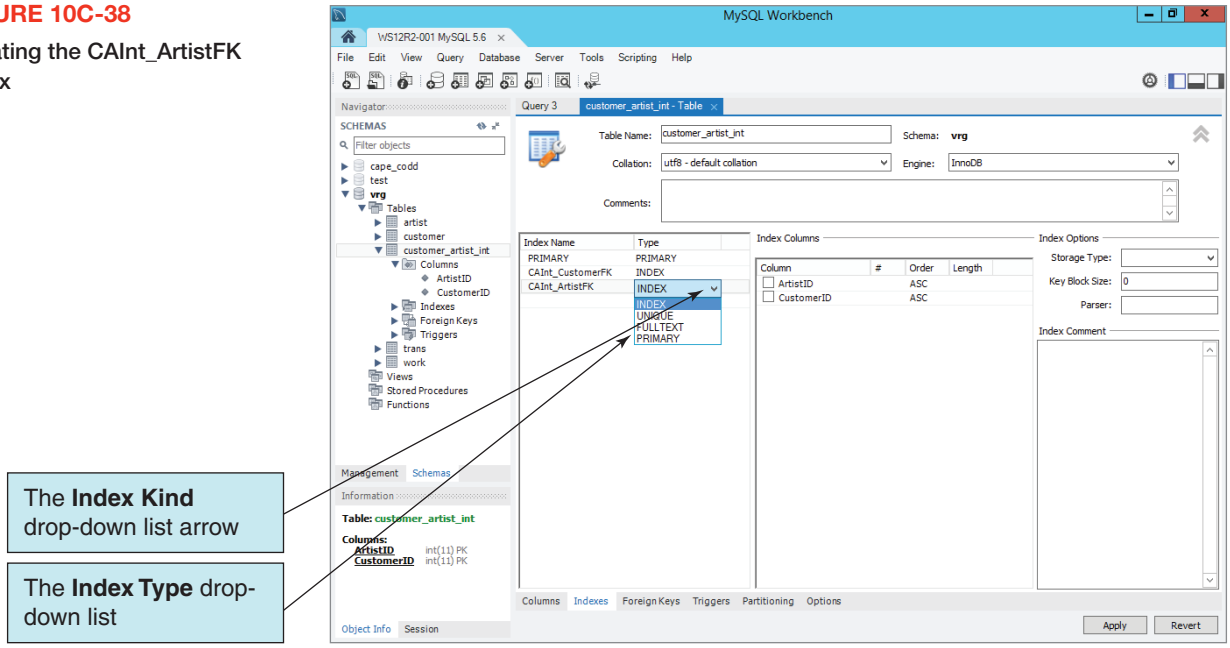


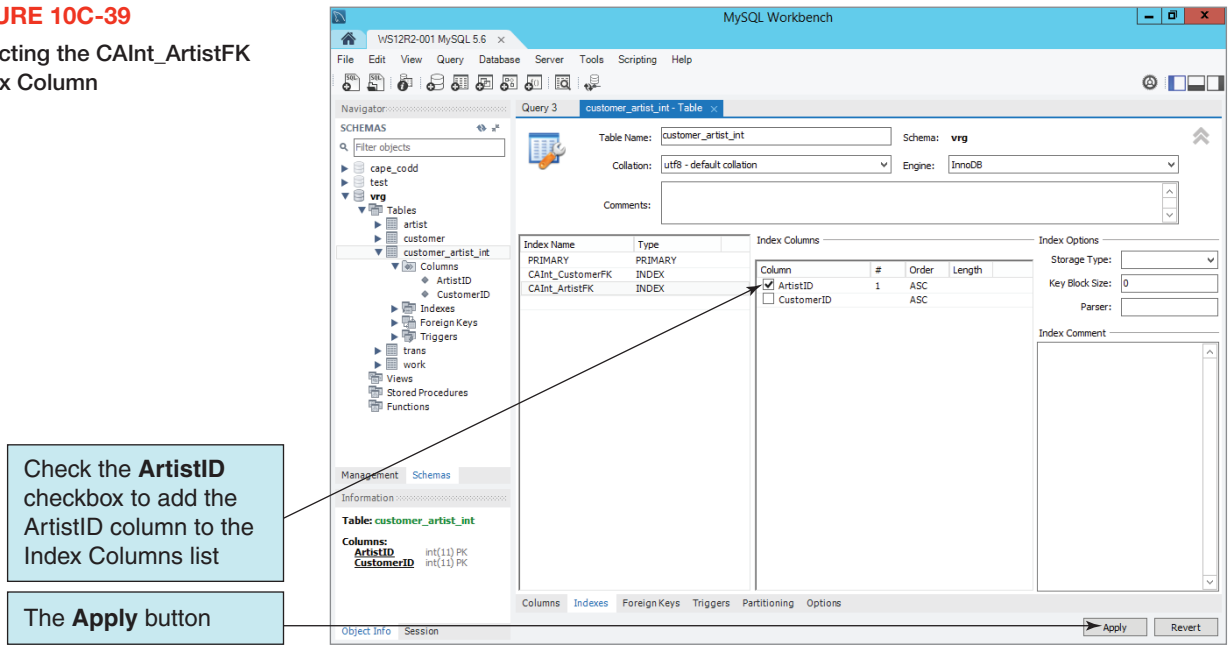
FIGURE 10C-38
Creating the CAInt_ArtistFK Index

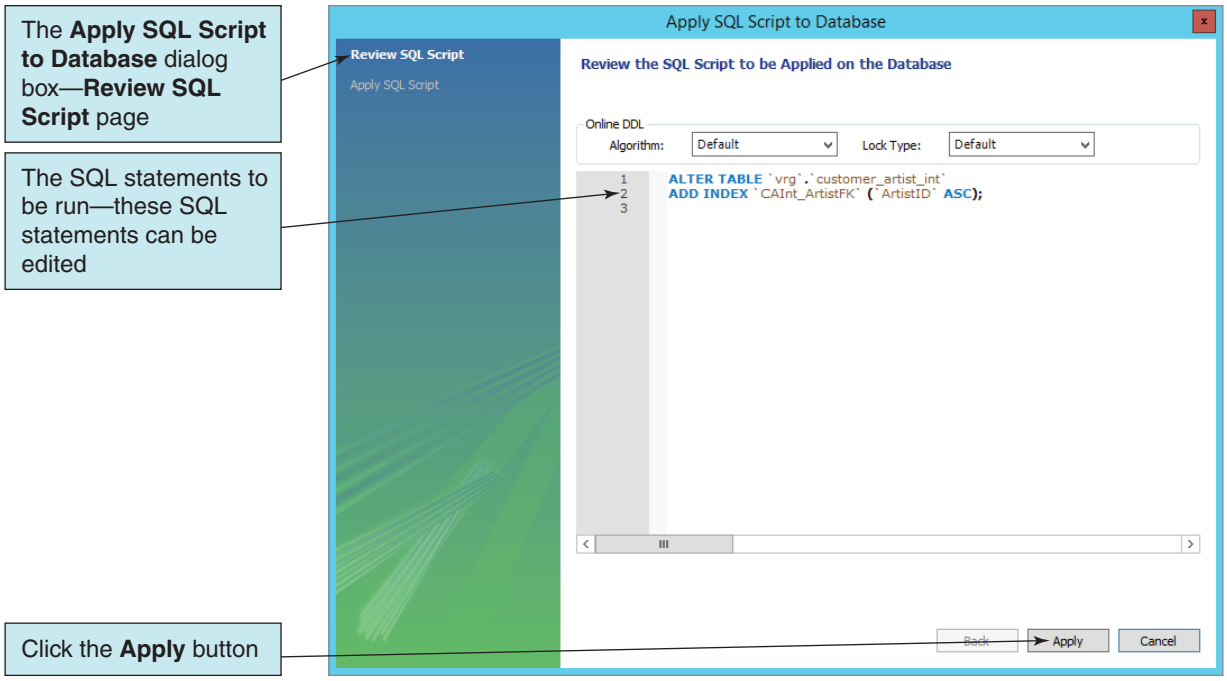


12. Click the **Finish SQL** button. The MySQL Table Editor is displayed again, with the new index, as shown in Figure 10C-41.
13. Click the **Close** button.

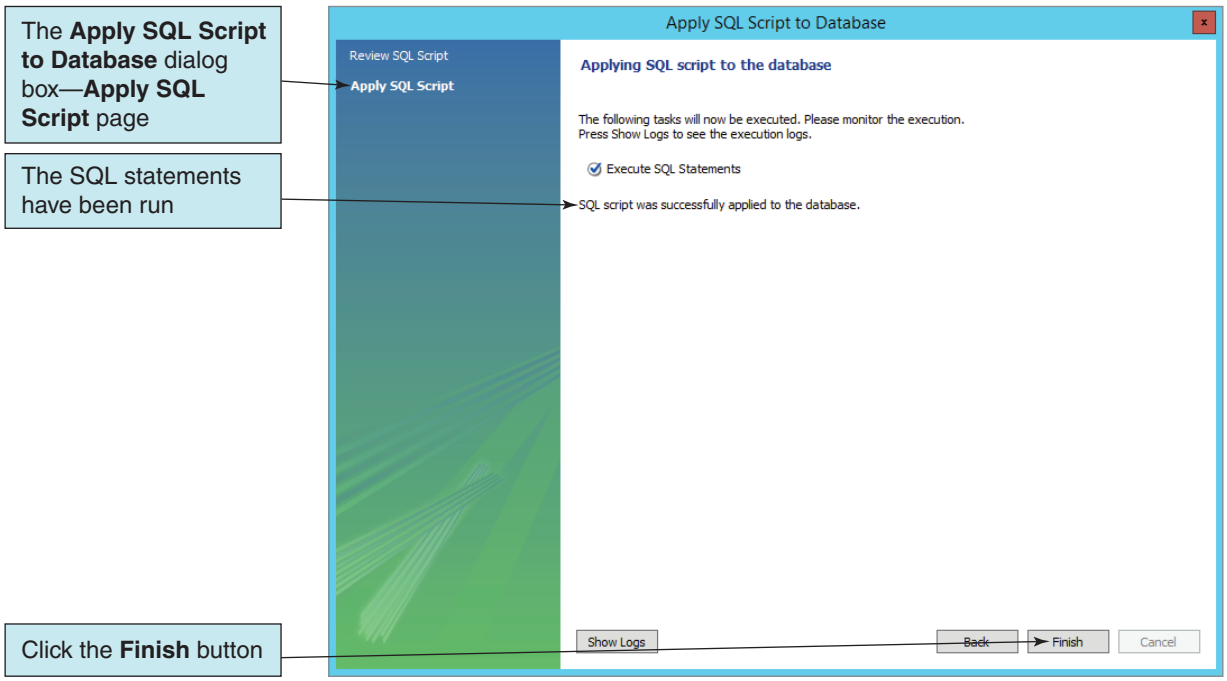
Note that the result of our actions in the GUI tool was an SQL statement that was run to actually create the index. This demonstrates the use of GUI tools as a visual way to build the SQL statements needed to actually change the database. It also demonstrates that we can use SQL statements directly if we know which SQL statement to use. To illustrate this, we will create another index using the **SQL ALTER TABLE ADD INDEX statement** itself. In this case, we will index the ZIPorPostalCode column in the CUSTOMER table to expedite sorts and searches on that column.

FIGURE 10C-39
Selecting the CAInt_ArtistFK Index Column





(a) Review SQL Script



(b) Apply SQL Script

FIGURE 10C-40

The Apply SQL Script to Database Dialog Box for the New Index

Creating a New Index Using an SQL statement

1. In the MySQL Workbench, make sure that VRG is the default schema, and, if necessary, expand VRG so the table objects are visible.
2. Click on the **File | New Script Tab** in the MySQL Workbench menu.
3. A new SQL Query tabbed window is opened in the SQL Editor.

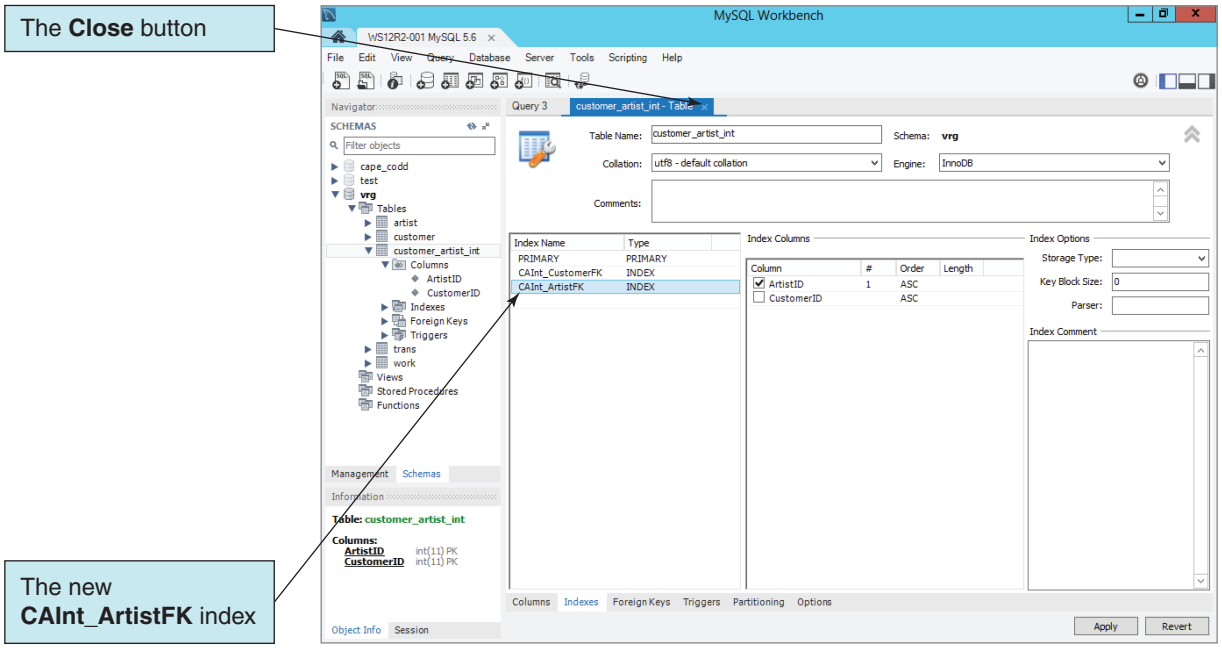


FIGURE 10C-41

The Completed CAInt_ArtistFK Index

4. In the SQL Editor, enter the SQL statement:

```
/* *** SQL-ALTER-TABLE-CH10C-01 *** */
ALTER TABLE CUSTOMER
        ADD INDEX ZIPorPostalCodeIndex
                USING BTREE(ZIPorPostalCode);
```

   as shown in Figure 10C-42.
5. Click the **Execute Current SQL Statement in Connected Server** button to run the SQL statement.
6. In the Object Browser, click the **CUSTOMER** table object to select it.

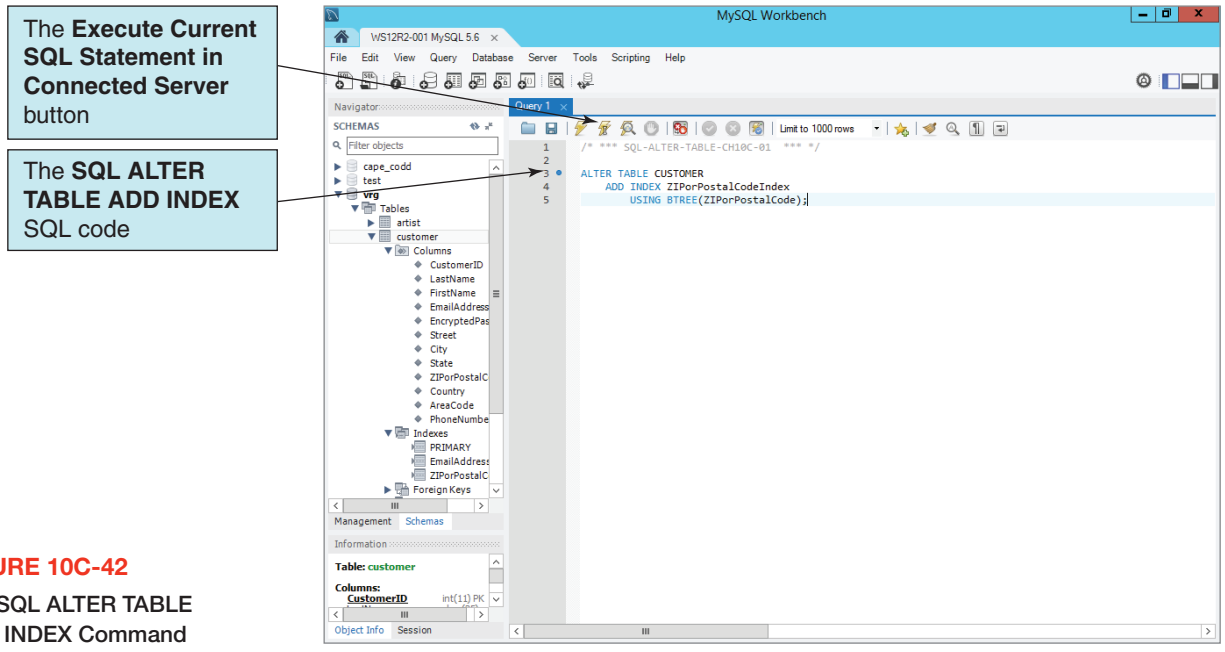


FIGURE 10C-42

The SQL ALTER TABLE ADD INDEX Command

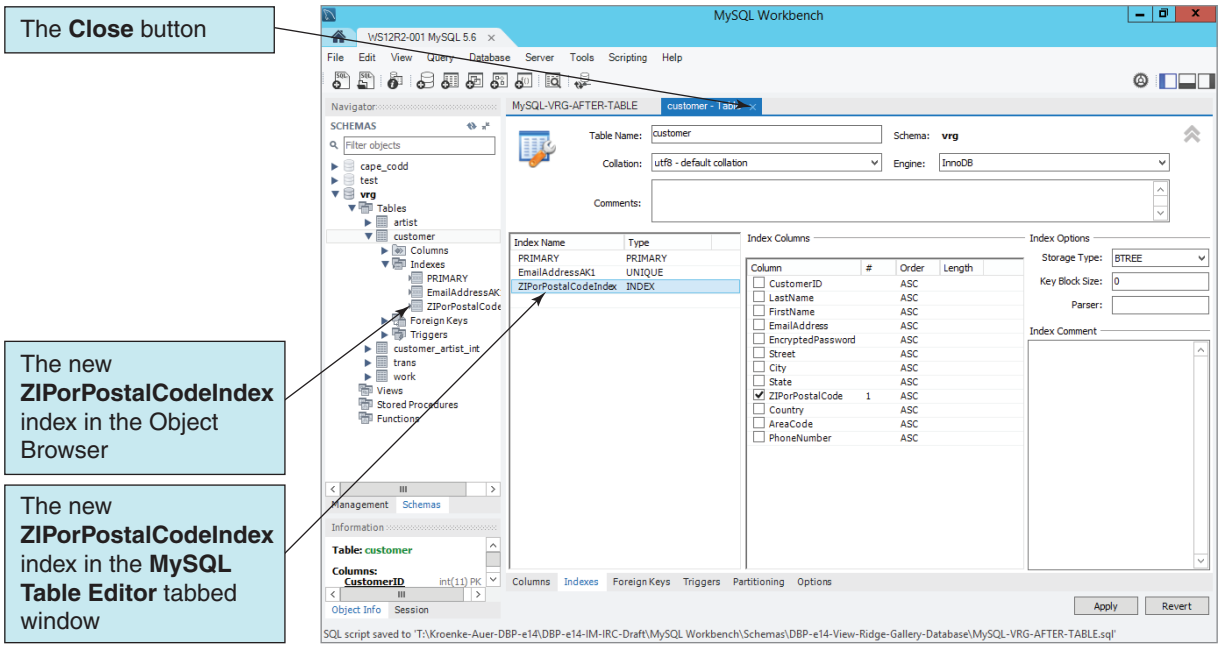


FIGURE 10C-43

The Completed ZIPorPostalCodeIndex Index

7. Right-click the **CUSTOMER** table object to display the shortcut menu.
8. Click the **Edit Table** command on the shortcut menu. The MySQL Table Editor is displayed.
9. Click the **Indexes** tab to display the foreign key settings for the CUSTOMER table, and note that a new index named ZIPorPostalCodeIndex has been added.
10. Click the ZIPorPostalCodeIndex row to see the index details, which appear as shown in Figure 10C-43.
11. Click the **Close** button on the MySQL Table Editor to close the editor.
12. Click the **SQL Query** tab in the SQL Editor window. You do not need to save the SQL ALTER TABLE CREATE INDEX statement, but you may if you want to.

Populating the VRG Tables with Data

MySQL does not have a GUI tool with a table grid for entering data into a table. Therefore, we must use SQL INSERT statements to enter, modify, and delete table data. But before we do that, we need to address the surrogate key values issue raised in Chapter 7. The data shown in Figure 7-15 are sample data, and the primary key values of CustomerID, ArtistID, WorkID, and TransactionID shown in that figure are nonsequential. Yet the AUTO_INCREMENT attribute that we use to populate MySQL surrogate primary keys creates sequential numbering.

This means that if we write and execute SQL INSERT statements to put the artist data shown in Figure 7-15(b) into the ARTIST table, the values of ArtistID that will be added to the table will be (1, 2, 3, 4, 5, 6, 7, 8, 9) instead of the values of (1, 2, 3, 4, 5, 11, 17, 18, 19) listed in the figure. How can we enter the needed nonsequential values?

The answer is to create the tables without the AUTO_INCREMENT attribute, and use this table structure to insert the non-sequential data. When this is complete, we then alter the tables to add the AUTO_INCREMENT attribute so that any additional data are inserted with the correctly sequenced surrogate key values.

However, in order to add the AUTO_INCREMENT attribute using the SQL ALTER statement, we have to temporarily remove any referential integrity constraints and restore them after the table is modified. Thus, instead of using an SQL INSERT statement that automatically enters the surrogate value, such as:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-INSERT-CH10C-01 *** */
INSERT INTO ARTIST VALUES('Miro', 'Joan', 'Spanish', 1893, 1983);
```

we have to use a set of SQL statements similar to the following:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-INSERT-CH10C-02 *** */
INSERT INTO ARTIST
           (ArtistID, LastName, FirstName, Nationality,
            DateOfBirth, DateDeceased)
           VALUES (1, 'Miro', 'Joan', 'Spanish', 1893, 1983);

/*    Set AUTO_INCREMENT for the ARTIST table.                */
/*    This requires six (6) steps.                            */

/* *** SQL-ALTER-TABLE-CH10C-02 *** */
ALTER TABLE WORK
      DROP FOREIGN KEY ArtistFK;

/* *** SQL-ALTER-TABLE-CH10C-03 *** */
ALTER TABLE CUSTOMER_ARTIST_INT
      DROP FOREIGN KEY CAInt_ArtistFK;

/* *** SQL-ALTER-TABLE-CH10C-04 *** */
ALTER TABLE ARTIST
      MODIFY COLUMN ArtistID INTEGER NOT NULL AUTO_INCREMENT;

/* *** SQL-ALTER-TABLE-CH10C-05 *** */
ALTER TABLE ARTIST AUTO_INCREMENT = 20;

/* *** SQL-ALTER-TABLE-CH10C-06 *** */
ALTER TABLE WORK
      ADD CONSTRAINT  ArtistFK              FOREIGN KEY(ArtistID)
                REFERENCES ARTIST(ArtistID)
                     ON UPDATE NO ACTION
                     ON DELETE NO ACTION;

/* *** SQL-ALTER-TABLE-CH10C-07 *** */
ALTER TABLE CUSTOMER_ARTIST_INT
      ADD CONSTRAINT CAInt_ArtistFK FOREIGN KEY(ArtistID)
                          REFERENCES ARTIST(ArtistID)
                               ON UPDATE NO ACTION
                               ON DELETE CASCADE;
```

Now insert the data, and then use ALTER statements to implement the correct AUTO _INCREMENT settings, including setting the next ArtistID in ARTIST value to be used to 20. When using this method, MySQL will correctly set the next value of the surrogate key. Of course, this is a lot of work if we are inserting one row of data at a time, but when used in an SQL script that inserts a lot of data into a table, it makes sense. So, we will use an SQL script (this SQL script is available on the book's Web site at *www.pearsonhighered.com/kroenke*).

The set of SQL INSERT statements needed to populate the VRG database with the View Ridge Gallery data shown in Figure 7-15 is shown in Figure 10C-44. As shown in Figure 10C-45, we create and save a new SQL script named *VRG-Insert-Data.sql* based on Figure 10C-44. Save the corrected script, and then run the script (use the Execute SQL Script in Connect Server button) to populate the tables. Close the script window after the script has been successfully run.

```
/*************************************************************************/
/*                                                                       */
/*     Kroenke and Auer - Database Processing (14th Edition) Chapter 10C     */
/*                                                                       */
/*     The View Ridge Gallery VRG Database - Insert Data                 */
/*                                                                       */
/*     These are the MySQL 5.6 SQL code solutions                        */
/*                                                                       */
/************************************************************************ *****/
/*                                                                       */
/*     This file contains the initial data for each table.               */
/*     This file also sets the AUTO_INCREMENT property for each table, but   */
/*     only after the non-consecutive surrogate key values are entered.      */
/*                                                                       */
/************************************************************************ **/

/********************************************************************** ***/

/*     INSERT data for CUSTOMER                                          */

INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1000, 'Janes', 'Jeffrey', 'Jeffrey.Janes@somewhere.com', 'ng76tG9E',
       '123 W. Elm St', 'Renton', 'WA', '98055', 'USA', '425', '543-2345');
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1001, 'Smith', 'David', 'David.Smith@somewhere.com', 'ttr67i23',
       '813 Tumbleweed Lane', 'Loveland', 'CO', '81201', 'USA', '970', '654-9876');
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1015, 'Twilight', 'Tiffany', 'Tiffany.Twilight@somewhere.com', 'gr44t5uz',
       '88 1st Avenue', 'Langley', 'WA', '98260', 'USA', '360', '765-5566');
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1033, 'Smathers', 'Fred', 'Fred.Smathers@somewhere.com', 'mnF3D00Q',
       '10899 88th Ave', 'Bainbridge Island', 'WA', '98110', 'USA', '206', '876-9911');
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1034, 'Frederickson', 'Mary Beth', 'MaryBeth.Frederickson@somewhere.com', 'Nd5qr4Tv',
       '25 South Lafayette', 'Denver', 'CO', '80201', 'USA', '303', '513-8822');
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1036, 'Warning', 'Selma', 'Selma.Warning@somewhere.com', 'CAe3Gh98',
       '205 Burnaby', 'Vancouver', 'BC', 'V6Z 1W2', 'Canada', '604', '988-0512');
```

FIGURE 10C-44

The SQL Statements to Populate the VRG Database Tables

```
INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1037, 'Wu', 'Susan', 'Susan.Wu@somewhere.com', 'Ues3thQ2',
       '105 Locust Ave', 'Atlanta', 'GA', '30322', 'USA', '404', '653-3465');

INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1040, 'Gray', 'Donald','Donald.Gray@somewhere.com', NULL,
       '55 Bodega Ave', 'Bodega Bay', 'CA', '94923', 'USA', '707', '568-4839');

INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1041, 'Johnson', 'Lynda', NULL, NULL,
       '117 C Street', 'Washington', 'DC', '20003', 'USA', '202', '438-5498');

INSERT INTO CUSTOMER
       (CustomerID, LastName,  FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1051, 'Wilkens', 'Chris', 'Chris.Wilkens@somewhere.com', '45QZjx59',
       '87 Highland Drive', 'Olympia', 'WA', '98508', 'USA', '360', '876-8822');


/*     Set AUTO_INCREMENT for the CUSTOMER table                                */

ALTER TABLE TRANS
       DROP FOREIGN KEY TransCustomerFK;

ALTER TABLE CUSTOMER_ARTIST_INT
       DROP FOREIGN KEY CAInt_CustomerFK;

ALTER TABLE CUSTOMER
       MODIFY COLUMN CustomerID INTEGER NOT NULL AUTO_INCREMENT;

ALTER TABLE CUSTOMER AUTO_INCREMENT = 1052;

ALTER TABLE TRANS
       ADD CONSTRAINT   TransCustomerFK   FOREIGN KEY(CustomerID)
              REFERENCES CUSTOMER(CustomerID)
                     ON UPDATE NO ACTION
                     ON DELETE NO ACTION;

ALTER TABLE CUSTOMER_ARTIST_INT
       ADD CONSTRAINT   CAInt_CustomerFK   FOREIGN KEY(CustomerID)
              REFERENCES CUSTOMER(CustomerID)
                     ON UPDATE NO ACTION
                     ON DELETE CASCADE;
```

FIGURE 10C-44 (*continued*)

Continued

```
/*************************************************************************/

/*     INSERT data for ARTIST                                            */

INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (1, 'Miro', 'Joan', 'Spanish', 1893, 1983);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (2, 'Kandinsky', 'Wassily', 'Russian', 1866, 1944);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (3, 'Klee', 'Paul', 'German', 1879, 1940);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (4, 'Matisse', 'Henri', 'French', 1869, 1954);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (5, 'Chagall', 'Marc', 'French', 1887, 1985);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (11, 'Sargent', 'John Singer', 'United States', 1856, 1925);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (17, 'Tobey', 'Mark', 'United States', 1890, 1976);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (18, 'Horiuchi', 'Paul', 'United States', 1906, 1999);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (19, 'Graves', 'Morris', 'United States', 1920, 2001);


/*     Set AUTO_INCREMENT for the ARTIST table                           */

ALTER TABLE WORK
       DROP FOREIGN KEY ArtistFK;

ALTER TABLE CUSTOMER_ARTIST_INT
       DROP FOREIGN KEY CAInt_ArtistFK;

ALTER TABLE ARTIST
       MODIFY COLUMN ArtistID INTEGER NOT NULL AUTO_INCREMENT;

ALTER TABLE ARTIST AUTO_INCREMENT = 20;

ALTER TABLE WORK
       ADD CONSTRAINT   ArtistFK         FOREIGN KEY(ArtistID)
                 REFERENCES ARTIST(ArtistID)
                        ON UPDATE NO ACTION
                        ON DELETE NO ACTION;


ALTER TABLE CUSTOMER_ARTIST_INT
       ADD CONSTRAINT   CAInt_ArtistFK       FOREIGN KEY(ArtistID)
                 REFERENCES ARTIST(ArtistID)
                        ON UPDATE NO ACTION
                        ON DELETE CASCADE;
```

FIGURE 10C-44
Continued

```
/*************************************************************************/

/*     INSERT data for CUSTOMER_ARTIST_INT                               */


INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1051);


/*************************************************************************/

/*     INSERT data for WORK                                              */

INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       500, 'Memories IV', 'Unique', 'Casein rice paper collage',
       '31 x 24.8 in.', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       511, 'Surf and Bird', '142/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       521, 'The Tilled Field', '788/1000', 'High Quality Limited Print',
       'Early Surrealist style', 1);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       522, 'La Lecon de Ski', '353/500', 'High Quality Limited Print',
       'Surrealist style', 1);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       523, 'On White II', '435/500', 'High Quality Limited Print',
       'Bauhaus style of Kandinsky', 2);
```

FIGURE 10C-44 (*continued*)

Continued

```
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       524, 'Woman with a Hat', '596/750', 'High Quality Limited Print',
       'A very colorful Impressionist piece', 4);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       537, 'The Woven World', '17/750', 'Color lithograph',
       'Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       548, 'Night Bird', 'Unique', 'Watercolor on Paper',
       '50 x 72.5 cm. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       551, 'Der Blaue Reiter', '236/1000', 'High Quality Limited Print',
       'The Blue Rider-Early Pointilism influence', 2);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       552, 'Angelus Novus', '659/750', 'High Quality Limited Print',
       'Bauhaus style of Klee', 3);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       553, 'The Dance', '734/1000', 'High Quality Limited Print',
       'An Impressionist masterpiece', 4);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       554, 'I and the Village', '834/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       555, 'Claude Monet Painting', '684/1000', 'High Quality Limited Print',
       'Shows French Impressionist influence of Monet', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       561, 'Sunflower', 'Unique', 'Watercolor and ink',
       '33.3 x 16.1 cm. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       562, 'The Fiddler', '251/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       563, 'Spanish Dancer', '583/750', 'High Quality Limited Print',
       'American realist style - From work in Spain', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       564, 'Farmer''s Market #2',       '267/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       565, 'Farmer''s Market #2',       '268/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       566, 'Into Time', '323/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       570, 'Untitled Number 1', 'Unique', 'Monotype with tempera',
       '4.3 x 6.1 in. Signed', 17);
```

FIGURE 10C-44

Continued

```
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       571, 'Yellow Covers Blue', 'Unique', 'Oil and collage',
       '71 x 78 in. - Signed', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       578, 'Mid-Century Hibernation', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       580, 'Forms in Progress I', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       581, 'Forms in Progress II', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       585, 'The Fiddler', '252/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       586, 'Spanish Dancer', '588/750', 'High Quality Limited Print',
       'American Realist style - From work in Spain', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       587, 'Broadway Boggie', '433/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       588, 'Universal Field', '114/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       589, 'Color Floating in Time', '487/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       590, 'Blue Interior', 'Unique', 'Tempera on card','43.9 x 28 in.', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       593, 'Surf and Bird', 'Unique', 'Gouache', '26.5 x 29.75 in. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       594, 'Surf and Bird', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       595, 'Surf and Bird', '365/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       596, 'Surf and Bird', '366/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
```

FIGURE 10C-44 (*continued*)

Continued

```
/*     Set AUTO_INCREMENT for the WORK table                                  */

ALTER TABLE TRANS
       DROP FOREIGN KEY TransWorkFK;

ALTER TABLE WORK
        MODIFY COLUMN WorkID INTEGER NOT NULL AUTO_INCREMENT;

ALTER TABLE WORK AUTO_INCREMENT = 597;

ALTER TABLE TRANS
       ADD CONSTRAINT   TransWorkFK     FOREIGN KEY(WorkID)
             REFERENCES WORK(WorkID)
                    ON UPDATE NO ACTION
                    ON DELETE NO ACTION;


/*************************************************************************/

/*     INSERT data for TRANS                                                 */


INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       100, '2011-11-04', 30000.00, 45000.00, '2011-12-14', 42500.00, 1000, 500);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       101, '2011-11-07', 250.00, 500.00, '2011-12-19', 500.00, 1015, 511);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       102, '2011-11-17', 125.00, 250.00, '2012-01-18', 200.00, 1001, 521);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       103, '2011-11-17', 250.00, 500.00, '2012-12-12', 400.00, 1034, 522);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       104, '2011-11-17', 250.00, 250.00, '2012-01-18', 200.00, 1001, 523);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       105, '2011-11-17', 200.00, 500.00, '2012-12-12', 400.00, 1034, 524);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       115, '2012-03-03', 1500.00, 3000.00, '2012-06-07', 2750.00, 1033, 537);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       121, '2012-09-21', 15000.00, 30000.00, '2012-11-28', 27500.00,  1015, 548);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       125, '2012-11-21', 125.00, 250.00, '2012-12-18', 200.00, 1001, 551);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       126, '2012-11-21', 200.00, 400.00, 552);
```

FIGURE 10C-44
Continued

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       127, '2012-11-21', 125.00, 500.00, '2012-12-22', 400.00, 1034, 553);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       128, '2012-11-21', 125.00, 250.00, '2013-03-16', 225.00, 1036, 554);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       129, '2012-11-21', 125.00, 250.00, '2013-03-16', 225.00, 1036, 555);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       151, '2013-05-07', 10000.00, 20000.00, '2013-06-28', 17500.00, 1036, 561);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       152, '2013-05-18', 125.00, 250.00, '2013-08-15', 225.00, 1001, 562);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       153, '2013-05-18', 200.00, 400.00, '2013-08-15', 350.00, 1001, 563);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       154, '2013-05-18', 250.00, 500.00, '2013-09-28', 400.00, 1040, 564);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       155, '2013-05-18', 250.00, 500.00, 565);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       156, '2013-05-18', 250.00, 500.00, '2013-09-27', 400.00, 1040, 566);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       161, '2013-06-28', 7500.00, 15000.00, '2013-09-29', 13750.00, 1033, 570);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       171, '2013-08-23', 35000.00, 60000.00, '2013-09-29', 55000.00, 1000, 571);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       175, '2013-08-23', 40000.00, 75000.00, '2013-12-18', 72500.00, 1036, 500);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       181, '2013-10-11', 250.00, 500.00, 578);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       201, '2014-02-28', 2000.00, 3500.00, '2014-04-26', 3250.00, 1040, 580);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       202, '2014-02-28', 2000.00, 3500.00, '2014-04-26', 3250.00, 1040, 581);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       225, '2014-06-08', 125.00, 250.00, '2014-09-27', 225.00, 1051, 585);
```

FIGURE 10C-44 (*continued*)

Continued

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       226, '2014-06-08', 200.00, 400.00, 586);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, Wo rkID)
       VALUES (
       227, '2014-06-08', 250.00, 500.00, '2014-09-27', 475.00, 1051, 587);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       228, '2014-06-08', 250.00, 500.00, 588);
INSERT INTO TRANS (Transaction ID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       229, '2014-06-08', 250.00, 500.00, 589);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       241, '2014-08-29', 2500.00, 5000.00, '2014-09-27', 4750.00, 1015, 590);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       251, '2014-10-25', 25000.00, 50000.00, 593);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       252, '2014-10-27', 250.00, 500.00, 594);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       253, '2014-10-27', 250.00, 500.00, 595);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       254, '2014-10-27', 250.00, 500.00, 596);


/*     Set AUTO_INCREMENT for the TRANS table                                  */

ALTER TABLE TRANS
       MODIFY COLUMN TransactionID INTEGER NOT NULL AUTO_INCREMENT;

ALTER TABLE TRANS AUTO_INCREMENT = 255;

/*************************************************************************/
```

FIGURE 10C-44

Continued

Transaction COMMIT in MySQL

The MySQL-VRG-Insert-Data SQL script shown in Figure 10C-45 contains callouts to a **Commit button**, to a **Rollback button**, and to **Toggle autocommit mode button**. In Chapter 9, we discussed transaction processing, where to ensure database integrity, a transaction is initiated with (in MYSQL) an **SQL START TRANSACTION statement**, and then either (1) committed to the database with an **SQL COMMIT statement** if the transaction is successful or BEGIN or (2) removed entirely from the database with an **SQL ROLLBACK statement** if there was an error in the transaction processing.

Transaction processing statements are used with DML commands such as INSERT and UPDATE. MySQL 5.6 enables an **autocommit mode** (implicit COMMIT mode) by default, where a successful transaction is committed to the database automatically. This is shown in Figure 10C-45 by the fact that the Toggle autocommit mode button is shown in color, indicating that autocommit mode is enabled. However, and as the name of the Toggle autocommit mode button implies, autocommit mode can be disabled by using the Toggle autocommit mode button.

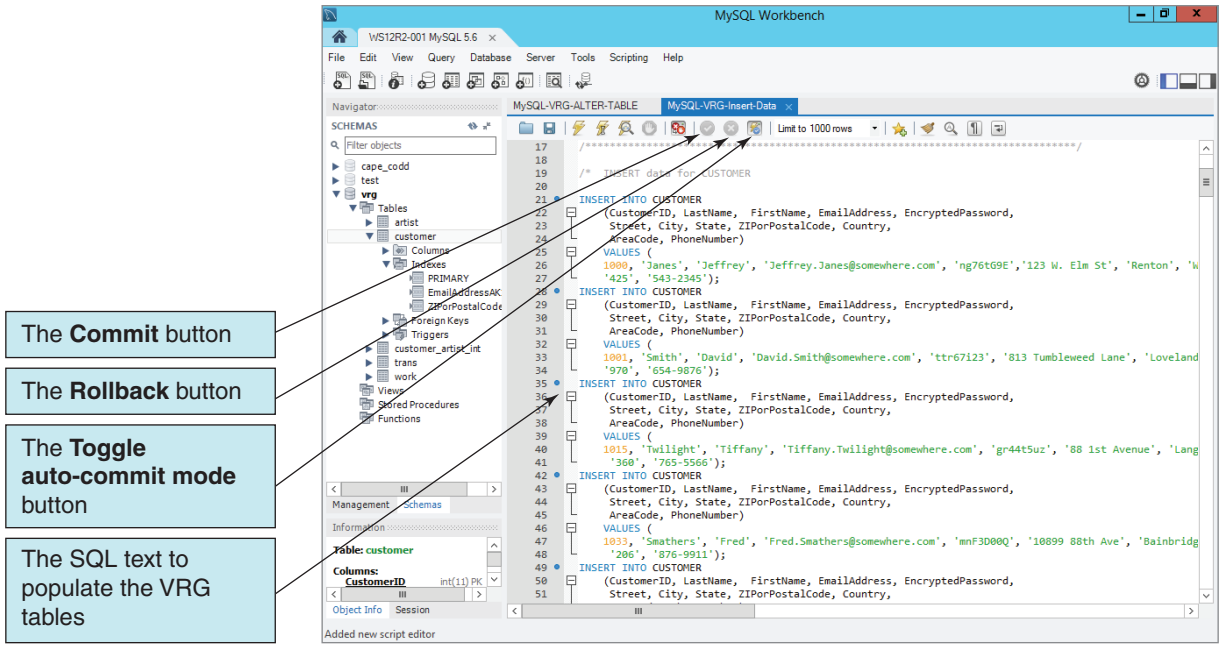


FIGURE 10C-45
Populating the VRG Database

When MySQL autocommit mode is disabled, the Commit and Rollback buttons will become functional and appear in color rather than grayed out as they are shown in Figure 10C-45.

Disabling MySQL autocommit mode means that explicit transaction COMMIT actions must be done, either by (1) running a COMMIT statement or (2) clicking the Commit button after the DML transaction is completed.

Creating Views

SQL views were discussed in Chapter 7; one view we created there was the view *CustomerInterestsView*. In MySQL, views are created with an SQL statement in MySQL Workbench. This view can be created with the SQL statement:

```
/* *** SQL-CREATE-VIEW-CH07-05 *** */
CREATE VIEW CustomerInterestsView AS
      SELECT      LastName AS CustomerLastName,
                  C.FirstName AS CustomerFirstName,
                  A.LastName AS ArtistName
      FROM        CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CAI
            ON    C.CustomerID = CAI.CustomerID
                        JOIN        ARTIST AS A
                                    ON    CAI.ArtistID = A.ArtistID;
```

As usual, it is best to store our code for creating database structures such as views in a script file. Figure 10C-46 shows this SQL CREATE VIEW statement in an SQL script named *VRG-Create-Views.sql* in the MySQL Workbench.

Creating a New View

1. In the MySQL Workbench, click the **File | New Script Tab** command to open a new tabbed script window.
   - **NOTE:** We could also just use the SQL Editor Scratch tab area.
2. In the Object Browser, select **vrg** from the Default Schema drop-down list to make it the active database.
3. Type the SQL statements shown in Figure 10C-46 into the script.

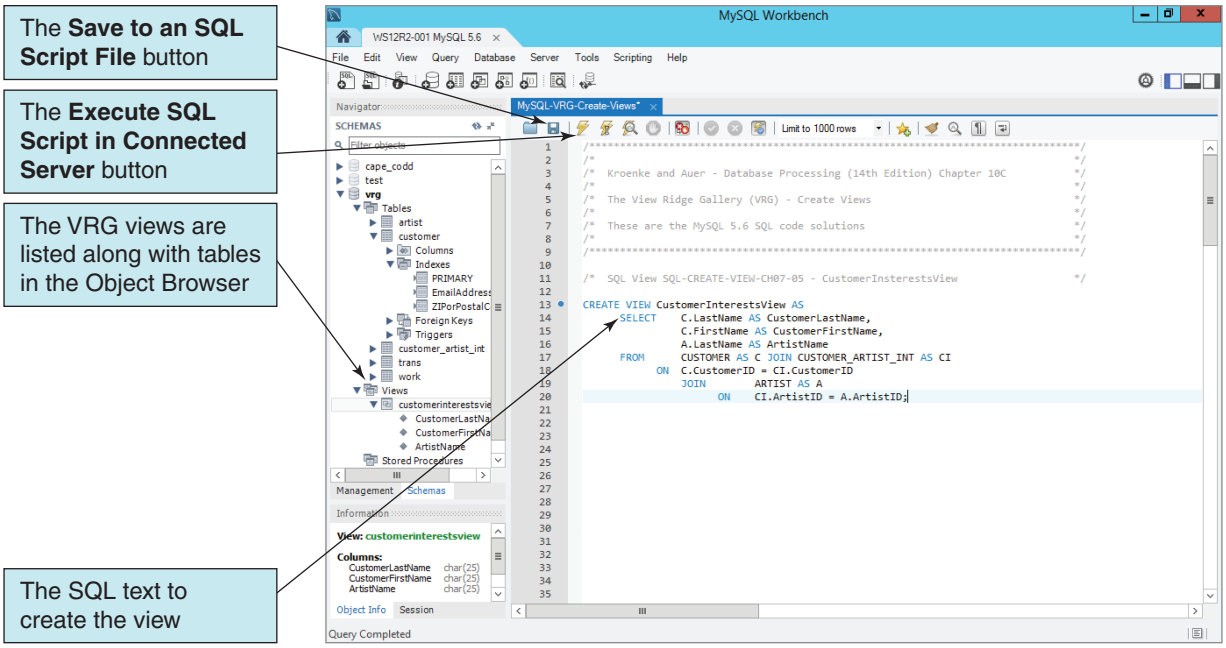


FIGURE 10C-46
Creating an SQL View

4. Save the script with the file name *VRG-Create-Views.sql*.
5. Click the **Execute SQL Script in Connected Server** button shown in Figure 10C-46. The view is created and appears along with the VRG tables in the Object Browser.
6. Click the script window **Close** button to close the SQL script.

As explained in Chapter 7, SQL views are used like tables in other SQL statements. For example, to see all the data in the view, we use the SQL SELECT statement:

```
/* *** SQL-Query-View-CH07-05 *** */
SELECT          *
FROM            CustomerInterestsView
ORDER BY        CustomerLastName, CustomerFirstName;
```

The SQL query and the result of running it are shown in Figure 10C-47. Note that in Figure 10C-47 we are also illustrating how to run a single SQL command within an SQL script–highlight the SQL command to be run, and then click the **Execute Current SQL Command in Connected Server** button.

The full output of the SQL-Query-View-CH07-05 query is shown in Figure 10C-48. At this point, you should add the other views shown in Chapter 7 to the script and run and test them. Note that MySQL, like all DBMS products, uses some different operators from other DBMS products–and different from those shown in Chapter 7. For example, in MySQL, the concatenation in the view CustomerPhoneView is coded using the **MySQL CONCAT string function**, as follows:

```
/* *** SQL-CREATE-VIEW-CH07-04 *** */
CREATE VIEW CustomerPhoneView AS
SELECT          LastName AS CustomerLastName,
                FirstName AS CustomerFirstName,
                CONCAT('(',AreaCode,') ',PhoneNumber) As CustomerPhone
FROM            CUSTOMER;
```

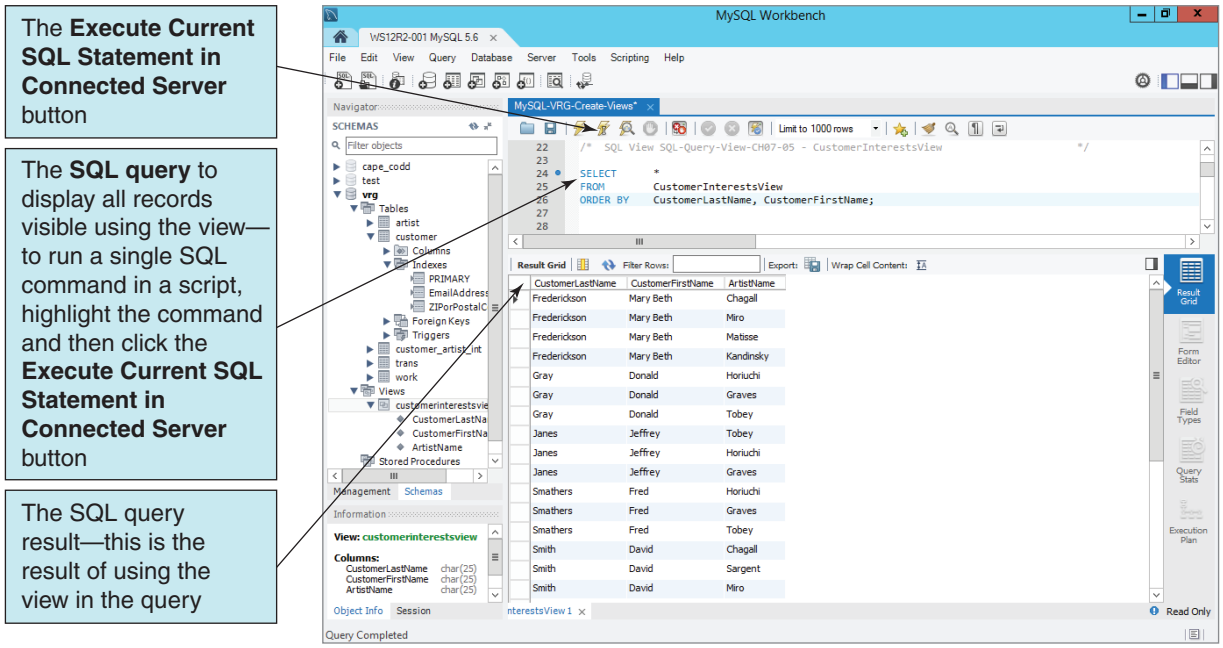


FIGURE 10C-47

Result of Using the View CustomerInterestsView

Importing Microsoft Excel Data into a MySQL 5.6 Database Table

When developing a database to support an application, it is very common to find that some (if not all) of the data needed in the database exists as data in user **worksheets** (also called **spreadsheets**). A typical example of this is a Microsoft Excel 2013 worksheet that a user has been maintaining, and which must now be converted to data stored in the database.

If we are really lucky, the worksheet will already be organized like a database table, with appropriate column labels and unique data in each row. And if we are *really, really lucky*, there will be one or more columns that can be used as the primary key in the new database table. In that case, we can easily import the data into the database. More likely, we will have to modify the worksheet, and organize and clean up the data in it before we can import the data. In essence, we are following a procedure that we will encounter again in Chapter 12 in our discussion of data warehouses known as **extract, transform and load (ETL)**.

As an example, let's consider the problem of postcards sold by the View Ridge Gallery. These postcards are pictures of the art work sold by the View Ridge Gallery (postcards of other popular works of art by well-known artists such as Claude Monet's *Water Lilies* are also stocked and sold, but to simplify the problem we will consider only postcards of art work at View Ridge Gallery). The inventory of postcards is currently kept in a Microsoft Excel 2013 worksheet named POSTCARDS, and shown in Figure 10C-49.

This worksheet breaks our basic rule of one theme per table, and combines artist, art work, and inventory into the same worksheet. Fortunately, there is no multivalue, multicolumn problem (as discussed in Chapter 4) in this worksheet. Even as it stands, this worksheet would need normalizing into BCNF to create the proper set of tables for the VRG database.

However, we will deal with the normalization of this data in the VRG database itself in the Review Questions at the end of this chapter. For now, we will simply import the worksheet into the VRG database and use it as a temporary table and source of data for populating the properly normalized tables.

For MySQL, we will create the POSTCARDS_TEMP Table using the **MySQL for Excel Add-In**. Install this utility using the MySQL Installer, and when Microsoft Excel is launched, it will appear on the DATA tab in the Microsoft Excel 2013 ribbon. The MySQL for Excel Add-In does a good job of letting users create a new table, set a primary key and specify most

FIGURE 10C-48
Complete Output of SQL Query on the View CustomerInterestsView

CustomerLastName	CustomerFirstName	ArtistName
Frederickson	Mary Beth	Chagall
Frederickson	Mary Beth	Miro
Frederickson	Mary Beth	Matisse
Frederickson	Mary Beth	Kandinsky
Gray	Donald	Horiuchi
Gray	Donald	Graves
Gray	Donald	Tobey
Janes	Jeffrey	Tobey
Janes	Jeffrey	Horiuchi
Janes	Jeffrey	Graves
Smathers	Fred	Horiuchi
Smathers	Fred	Graves
Smathers	Fred	Tobey
Smith	David	Chagall
Smith	David	Sargent
Smith	David	Miro
Smith	David	Matisse
Smith	David	Kandinsky
Twilight	Tiffany	Tobey
Twilight	Tiffany	Sargent
Twilight	Tiffany	Horiuchi
Twilight	Tiffany	Graves
Warning	Selma	Chagall
Warning	Selma	Sargent
Warning	Selma	Graves
Wilkens	Chris	Horiuchi
Wilkens	Chris	Graves
Wilkens	Chris	Tobey

column characteristics. We will have to use SQL ALTER TABLE statements when we integrate the data in this table into the VRG database. However, MySQL does not support some common SQL ALTER TABLE features, so we will have to use MySQL specific syntax (see: *http://dev.mysql.com/doc/refman/5.6/en/alter-table.html*).

Our solution is to:

- Use MySQL for Excel to import the data in a temporary table, then
- Use an SQL CREATE TABLE statement to create the actual table we want in the database, then
- Use an SQL INSERT statement to copy the data from the temporary table to the actual table, then
- Delete the temporary table from the database.

Note that in these steps we will use a new variant of the SQL INSERT statement, a **bulk INSERT statement**. We use this form of the SQL INSERT statement when we want to copy

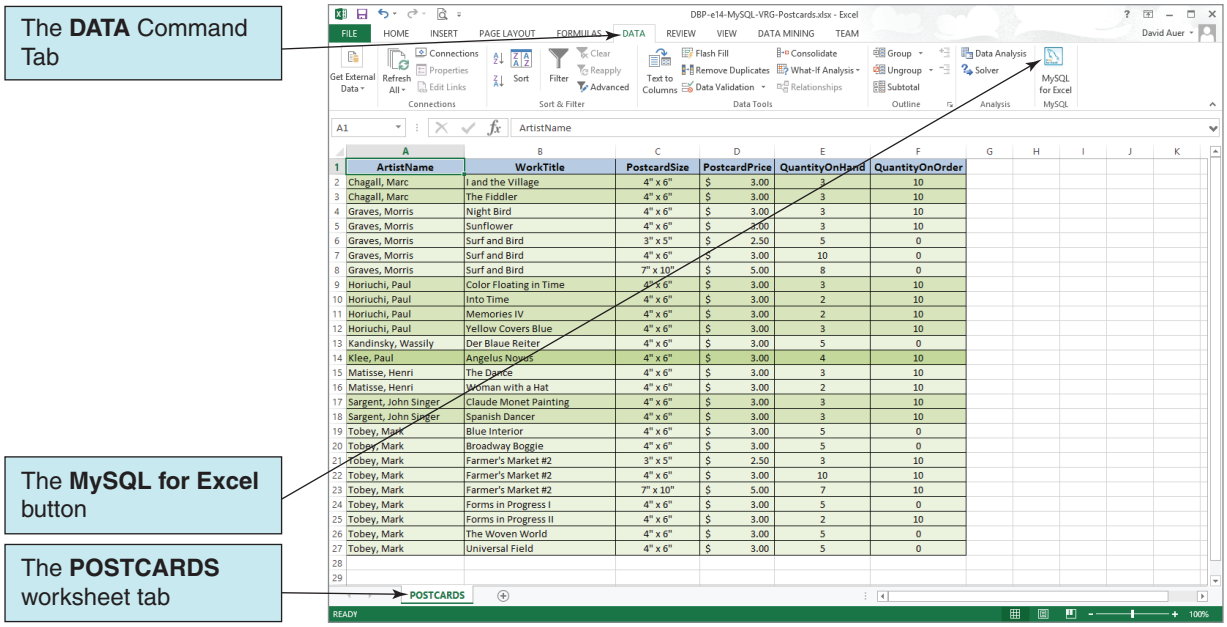


FIGURE 10C-49

The View Ridge Gallery POSTCARD Worksheet

a lot of data from one table to another, and copying from a temporary table to a final table is a great place to use this statement.

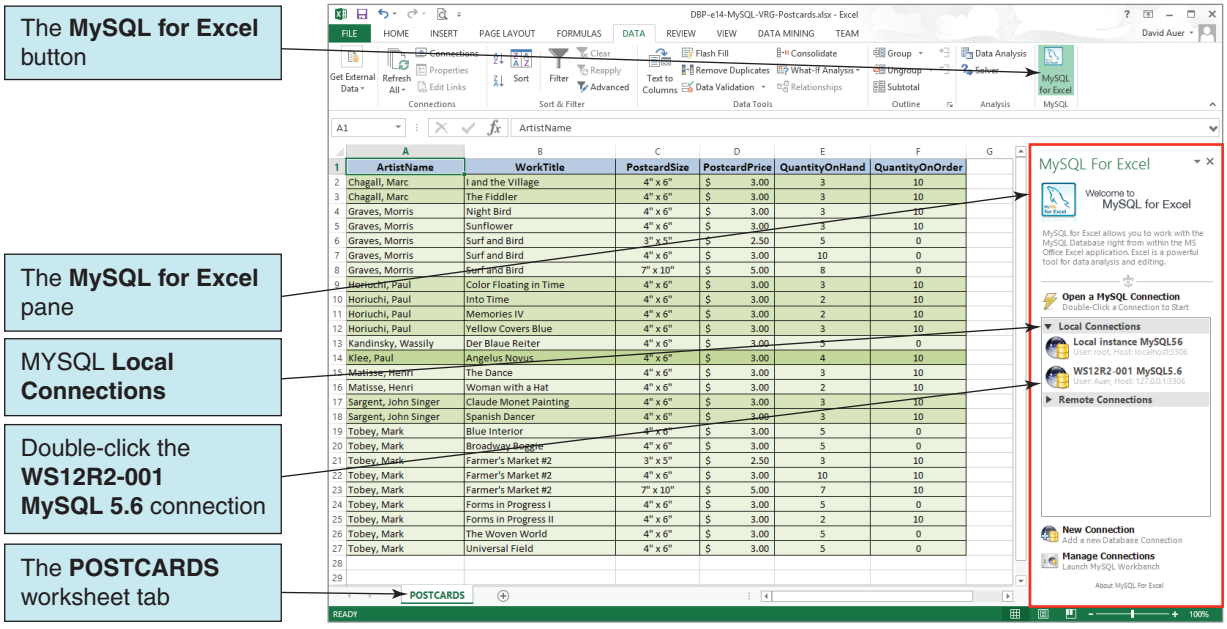
Here are the actual steps:

Importing the Microsoft Excel Data into a MySQL 5.6 Database Table

1. Open the POSTCARDS worksheet in Microsoft Excel 2013, click the **DATA** command tab in the Ribbon. The MySQL for Excel button is displayed, as shown in Figure 10C-49. Click the **MySQL for Excel** button to launch the MySQL for Excel *Open a MySQL Connection* pane, as shown in Figure 10C-50.
2. Open a MySQL connection by double-clicking **Local Connection link**. We will use the **WS12R2-001 MySQL 5.6** connection we created earlier in this chapter, but we could also use the default *Local instance MySQL56* link. The Connection Password dialog box is displayed, as shown in Figure 10C-51.

FIGURE 10C-50

The MySQL for Excel Open a MySQL Connection Pane



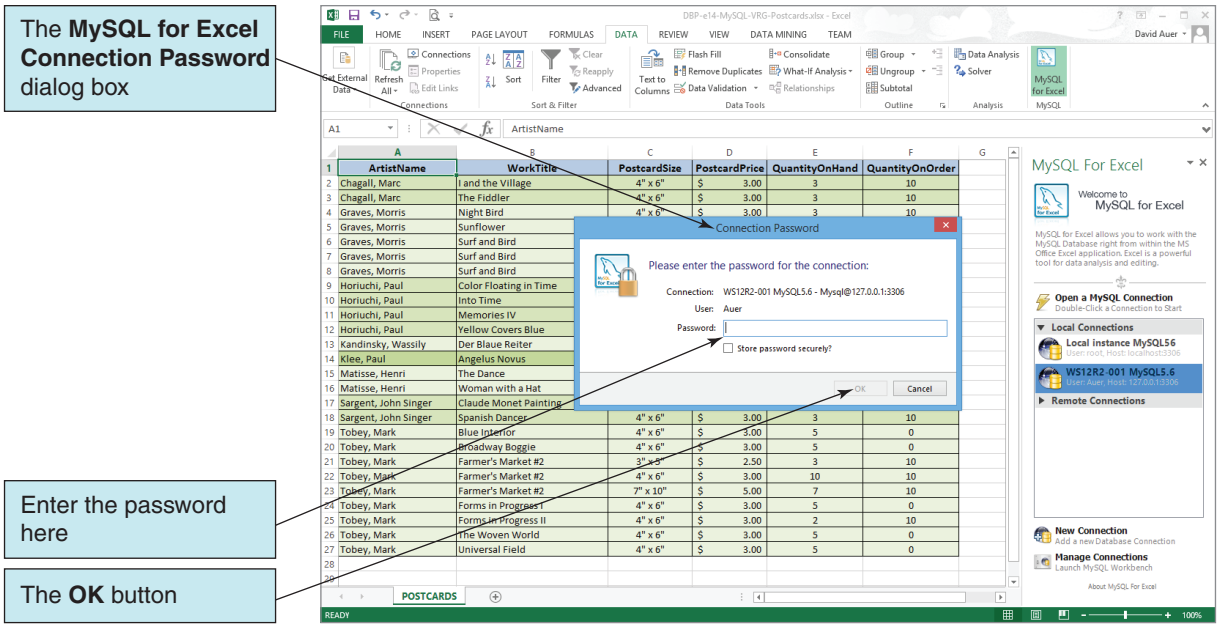


FIGURE 10C-51

The MySQL for Excel Connection Password Dialog Box

3. Enter the password and click the **OK** button. The MySQL for Excel *Select a Database Schema* pane is displayed, as shown in Figure 10C-52. Click the vrg schema name to select it, and then click the **Next** button.
4. In Microsoft Excel, select (highlight) the entire POSTCARDS table range!
5. As shown in Figure 10C-53, click the **Export Excel Data to New Table** command. The Export Data dialog box is displayed, as shown in Figure 10C-54, labeled with the name of the selected Microsoft Excel sheet and the selected range (POSTCARDS [A1:F27]).
6. The Export Data dialog box allows us to set the table specifications for the new table. As shown in Figure 10C-54, we enter the table name to be used in the database here–in this case we will use **postcards_temp** (note that MySQL does not allow

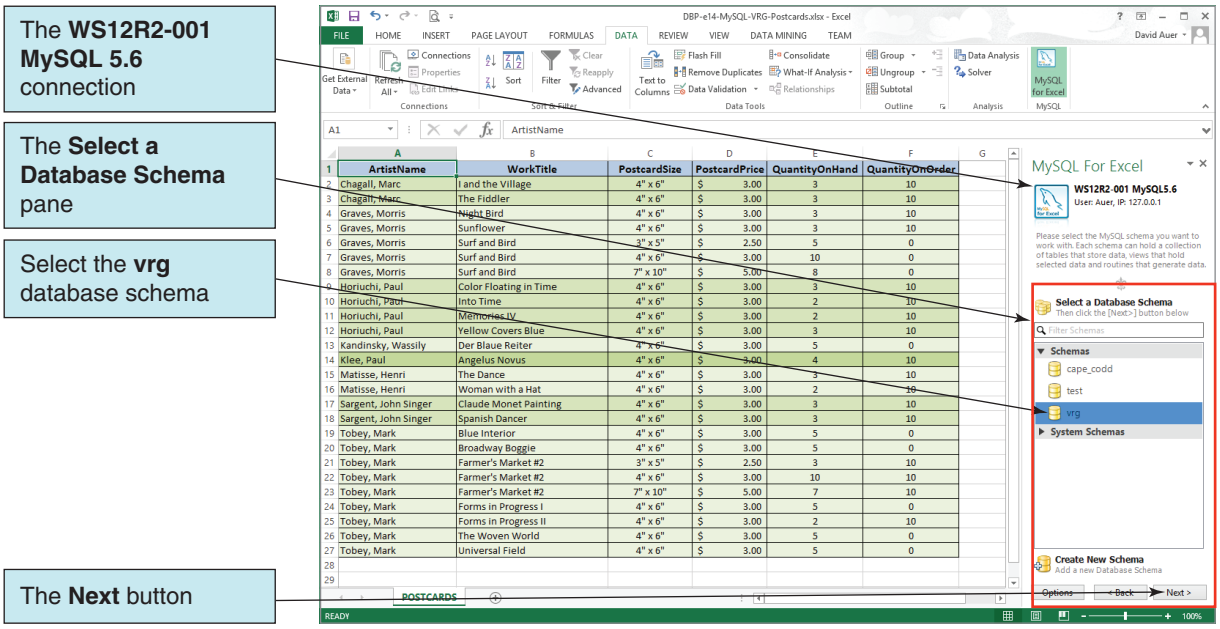


FIGURE 10C-52

The MySQL for Excel Select a Database Schema Pane

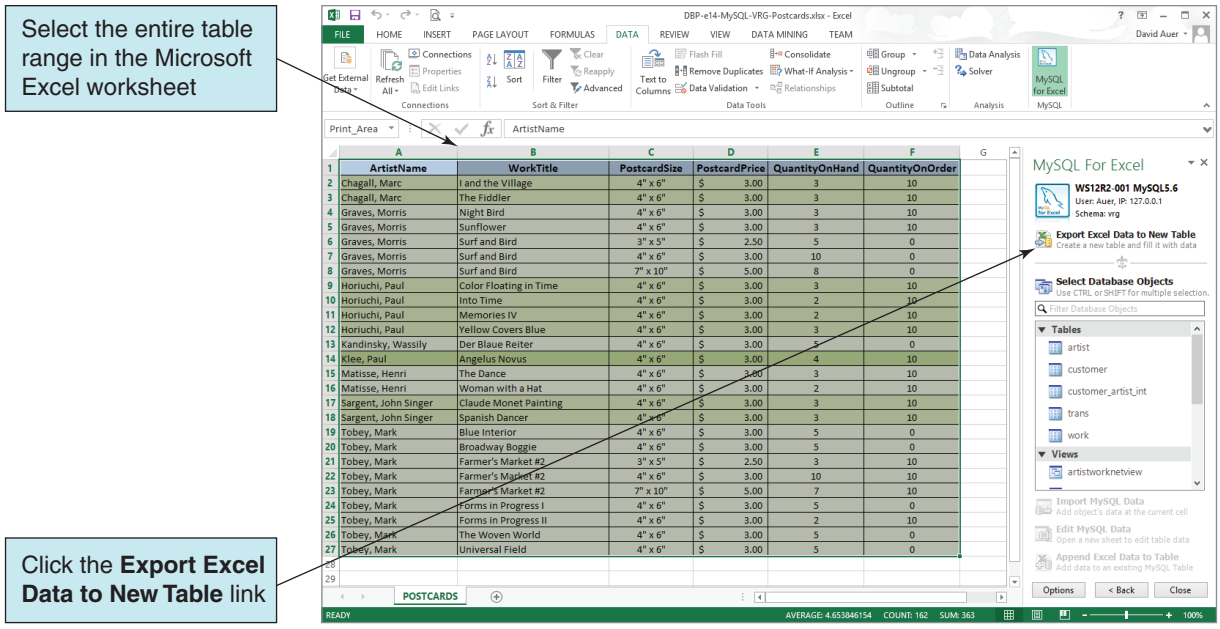


FIGURE 10C-53

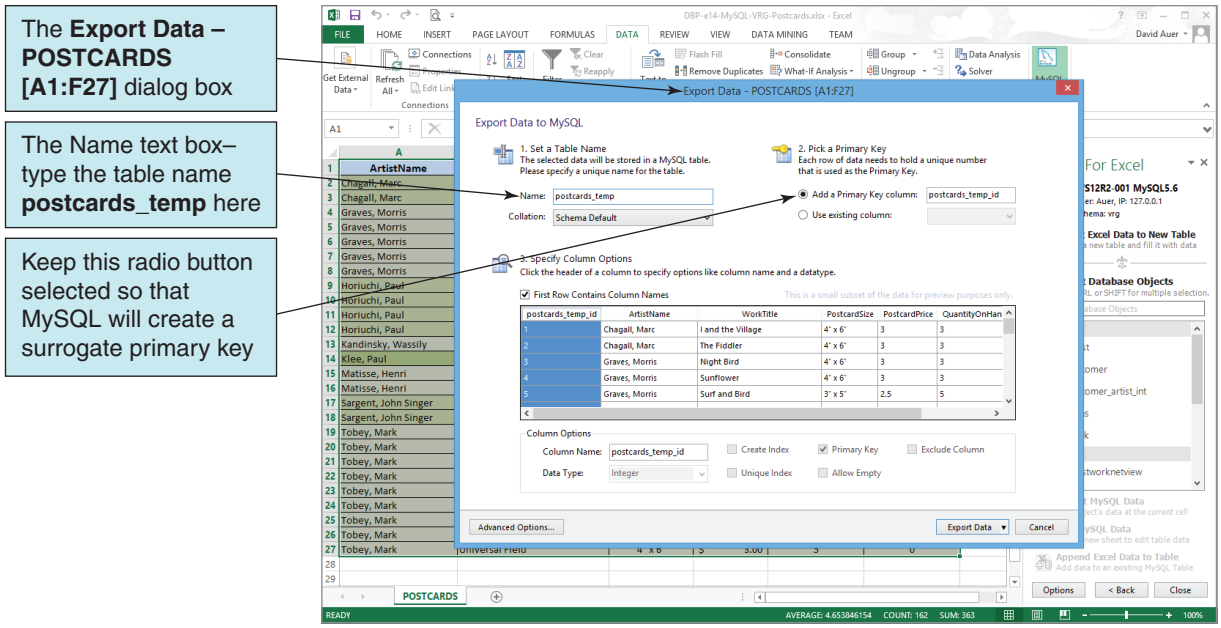
The Export Excel Database to New Table Button

upper case letters in the table name actually used in MySQL). MySQL also will create a surrogate primary key. Alternately, we could specific a primary key based on existing columns in the table. In this case we need to add a surrogate primary key, and will let MySQL do that.

7. As shown in Figure 10C-55, note that we can adjust data types and NULL/NOT NULL (shown as "Allow Empty") for each column. Because this is a temporary table, we will use the MySQL defaults.
8. Click the **Export Data** button. The new table is created and populated, as shown in the Success dialog box seen in Figure 10C-56.
9. In the Success dialog box, click the **OK** button.
10. In the Microsoft Excel *MySQL For Excel* pane, click the **Close** button to close MySQL for Excel.

FIGURE 10C-54

The Export Data – POSTCARDS [A1:F27] Dialog Box



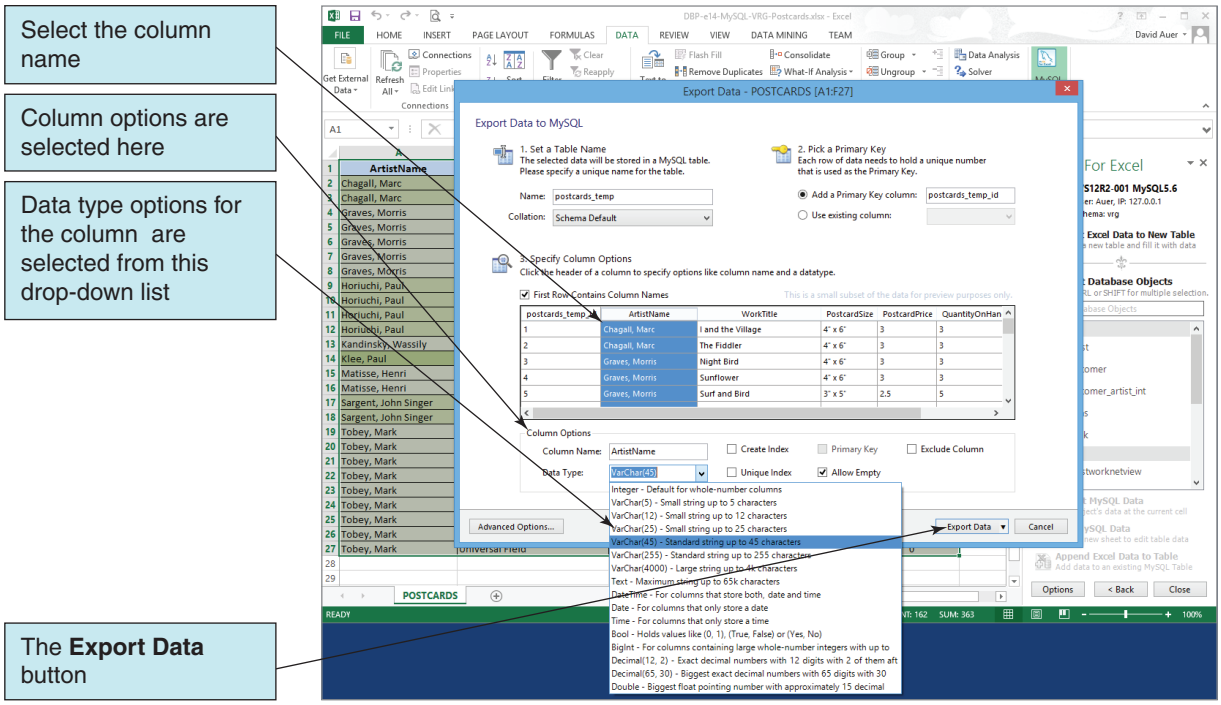
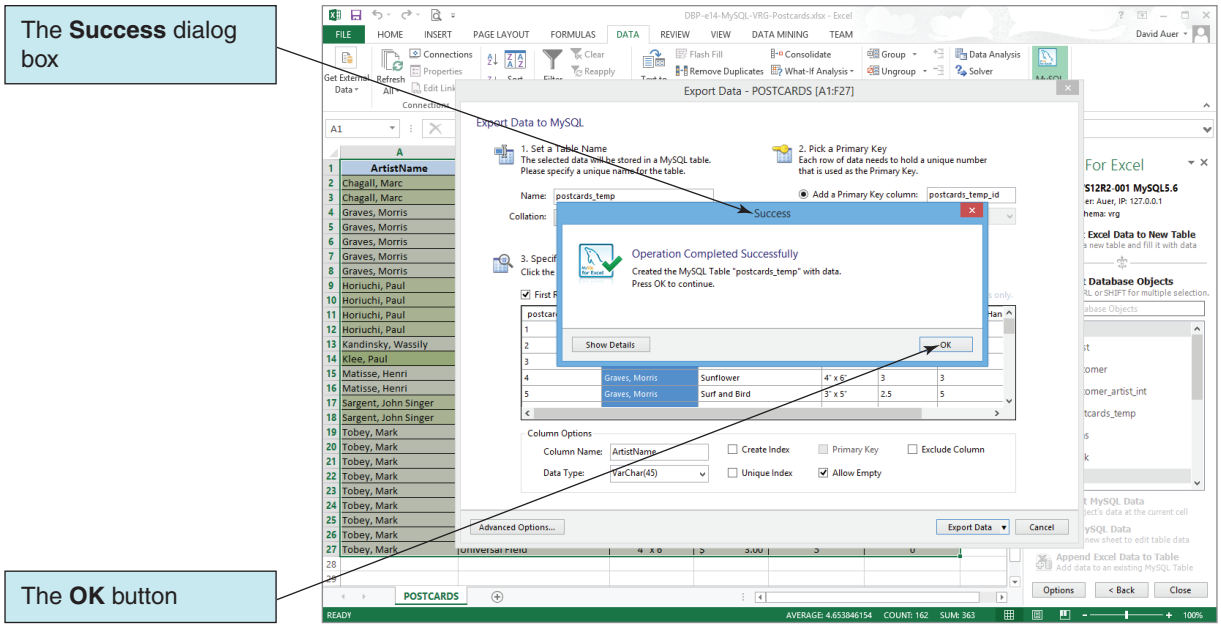


FIGURE 10C-55
Column Options

11. Save the Microsoft Excel workbook. If a dialog box appears warning about macro features that cannot be saved, ignore it and click the **Yes** button.
12. Close the Microsoft Excel workbook.
13. Open MySQL Workbench, and refresh the **vrg** schema. Expand the Tables object and the *postcards_temp* table object Columns. The imported table data is shown in Figure 10C-57.

Now that we have successfully imported the temporary *postcards_temp* table, we will need to design and implement the actual final table or tables in the VRG database to store this data in the form we want, and then populate those tables. We will deal with these steps in Project Question 10C.37.

FIGURE 10C-56
The Success Dialog Box



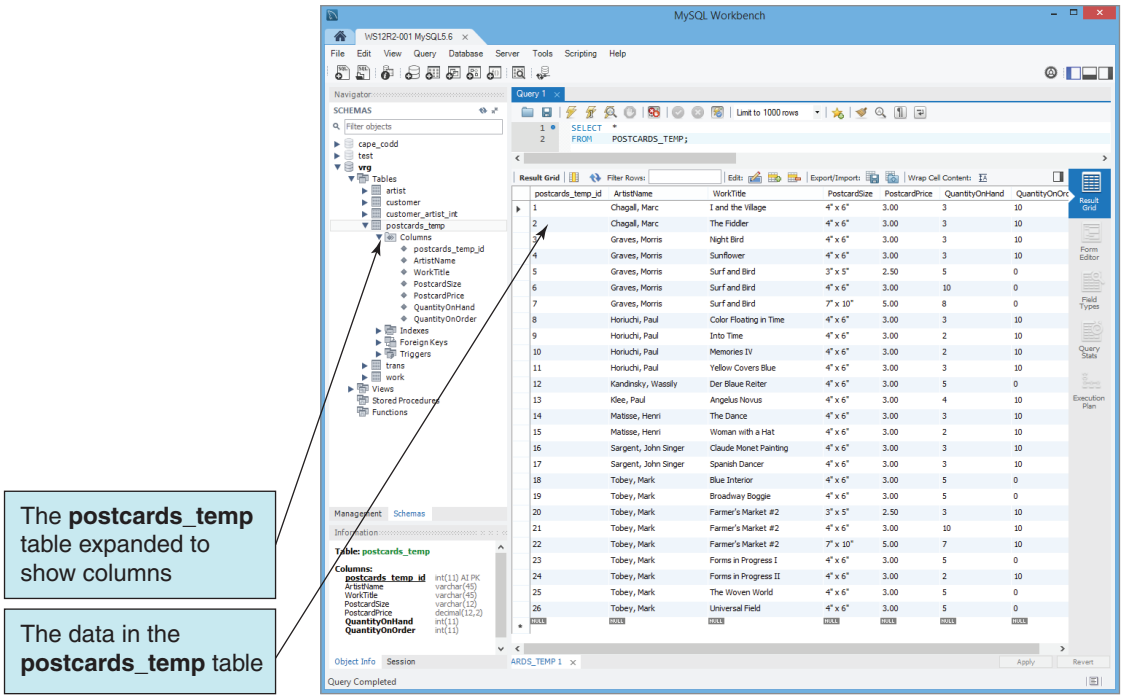


FIGURE 10C-57

The POSTCARDS_TEMP Data in the VRG Database

MySQL Application Logic

A MySQL database can be processed from an application in a number of different ways. The first is one that we are already familiar with: the use of SQL scripts. For security, however, such files should be used only during application development and testing and never on an operational database.

Another way is to create application code using a programming language such as Java or C++ and then invoke MySQL DBMS commands from those programs. The modern way to do this is to use a library of object classes, create objects that accomplish database work, and then process those objects by setting object properties and invoking object methods.

Application logic can be embedded in triggers. As you learned in Chapter 7, triggers can be used to set default values, to enforce data constraints, to update views, and to enforce referential integrity constraints. In this chapter, we will describe four triggers, one for each of the four trigger uses. These triggers should be invoked by MySQL when the specified actions occur.

Finally, another way of processing a MySQL database is to create stored procedures, as described in Chapter 7. These stored procedures can then be invoked from application programs or from Web pages using languages such as JavaScript or PHP. Stored procedures can also be executed from the MySQL Workbench. This should be done only when the procedures are being developed and tested, however. As described in Chapter 9, for security reasons, no one other than authorized members of the DBA staff should be allowed to interactively process an operational database.

In this chapter, we will describe and illustrate two stored procedures. We will test those procedures by invoking them from the MySQL Workbench, and some of our output will be designed specifically for this environment. Again, this should be done only during development and testing. You will learn how to invoke those stored procedures from application code in Chapter 11.

MySQL SQL/PSM Procedural Statements

MySQL's variant of SQL contains additional procedural language added for use in stored procedures and triggers. We will use certain elements of this SQL in such code, and therefore we need to discuss them at this point. Information on these and other SQL language components

can be found in the MySQL 5.6 documents "Chapter 13. SQL Syntax Reference" at *http://dev.mysql.com/doc/refman/5.6/en/sql-syntax.html* and "Chapter 20. Stored Programs and Views" at *http://dev.mysql.com/doc/refman/5.6/en/stored-programs-views.html*.

Basic MySQL SQL/PSM Structure and Statement Delimiters

Transact-SQL uses the same basic structure for user-defined functions, triggers and stored procedures, although each varies according to what it needs to accomplish. This basic structure is:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10C-01 *** */
CREATE  {FUNCTION, PROCEDURE, TRIGGER} {PSM_module_name}
        {Additional TRIGGER declarations here for TRIGGER only}
(
-- Define input parameters if any here
)
BEGIN
-- DECLARE local variables here
-- PSM module statements here
END
```

Note the use of the **BEGIN…END keywords**, which defines a block of Transact-SQL statements so more than one statement can be executed, to contain all procedural statements in the user-defined function, stored procedure, or trigger.

SQL statements in procedures and triggers are written with a **delimiter**, the same semicolon (;) required in nonprocedural SQL statements. However, when working with stored procedures in a command-line editor, MYSQL needs to use a different delimiter to allow input of semicolons *within* the procedure. For consistency, we will continue to use the semicolons for SQL statements but will switch to the **delimiter // (slash slash)** for creating stored procedures. This is done by using the code statements:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10C-01 *** */
DELIMITER //
{Stored Procedure code here, including lines that end with a semicolon}
//
DELIMITER;
```

MySQL SQL/PSM Variables

MySQL variables and parameters do not use any special symbol to identify them. Thus, *WorkID* can be both a column name and a MySQL variable or parameter. It is good coding practice to avoid such confusions by adopting naming conventions for parameters and variables. We will put the prefix *var* at the start of our variable names and the prefix *new* at the start of input parameters. A **parameter** is a value that is passed to the stored procedure when it is called. A **variable** is a value used within the stored procedure itself. Comments in MySQL code are either enclosed in **/* (slash asterisk) and */ (asterisk slash) signs** or preceded by a **# (pound sign)** if they are restricted to one line.

MySQL Control-of-Flow Statements

Control-of-flow statements contain procedural language components that let you control exactly which parts of your code are used and the conditions required for their use. These components include BEGIN…END, IF…ELSE…END IF, WHILE, RETURN, and other keywords that can be used to direct the operations of a block of code. The **BEGIN…END block** construction

is required to define a block of MySQL statements in a procedure or function. We can also use BEGIN…END blocks within another BEGIN…END block if doing so will help us control our code or clarify our logic. The **IF…ELSE…END IF keywords** are used to test for a condition and then direct which blocks of code are used based on the result of that test. Note that the END IF keyword *is* used as part of this construct in MySQL, and this is common in many programming languages. The **REPEAT keyword** and **WHILE keyword** are used to create loops in MySQL, where a section of code is repeated as long as some condition is true. The **ROLLBACK keyword** is used to exit a block of code and terminate a running procedure or stored procedure (but not a trigger—ROLLBACK cannot be used in a trigger). In this case, we "roll back" the transaction to its starting point and make sure the database is unchanged by the attempted actions.

As an example, let's consider a new customer at the View Ridge Gallery who needs to have customer data entered into the CUSTOMER table and artist interest data entered into the CUSTOMER_ARTIST_INT table. The new customer is Michael Bench, with phone number 206-876-8822, email address Michael.Bench@somewhere.com, and an interest in French artists.

Before we enter Michael's data, we need to check to see whether he is already in the database. To do this, we could use programming code similar to the following SQL code in a stored procedure:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10C-02 *** */
DELIMITER //
BEGIN
      DECLARE varRowCount INT;
      # Check to see if Customer already exists in database
      SELECT      varRowCount = COUNT(*)
      FROM        CUSTOMER
      WHERE       LastName = 'Bench'
            AND   FirstName = 'Michael'
            AND   AreaCode = '206'
            AND   PhoneNumber = '876-8822'
            AND   Email = 'Michael.Bench@somewhere.com';
      # IF varRowCount > 0 THEN Customer already exists.
      IF (varRowCount > 0)
      THEN ROLLACK;
      END IF;
      # IF varRowCount = 0 THEN add Customer to CUSTOMER table.
      INSERT INTO CUSTOMER
            (LastName, FirstName, AreaCode, PhoneNumber, Email)
            VALUES('Bench', 'Michael', '206', '876-8822',
                  'Michael.Bench@somewhere.com');
END
//
DELIMITER;
```

This block of SQL code illustrates the use of all of the control-of-flow keywords we have discussed except those used for looping. The WHILE and REPEAT keywords are used in code loops, and one use of a code loop is in an SQL cursor.

MySQL Cursor Statements

As we discussed in Chapter 7, a cursor is used so SQL results stored in a table can be processed one row at a time. Cursor-related keywords include DECLARE, OPEN, FETCH, CLOSE, and

DEALLOCATE. The **DECLARE CURSOR keywords** are used to create a cursor; the **OPEN keyword** actually starts the use of the cursor. The **FETCH keyword** is used to retrieve row data, and the **CLOSE keyword** is used to exit the use of a cursor. When using a cursor, the REPEAT and WHILE keywords are used to control how long the cursor is active.

Let's consider Michael Bench's interest in French artists. The ARTIST table has two French artists: Henri Matisse and Marc Chagall. Therefore, we need to add new rows to CUSTOMER_ARTIST_INT, both of which will contain Michael's CustomerID number (now that he has one) and each of which contains the ArtistID for one of these artists. To do this, we can use the following *section* of SQL code in a stored procedure (this in *not* a complete stored procedure by itself).

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-Code-Example-CH10C-03 *** */
DECLARE    varArtistID     Int;
DECLARE    varCustomerID   Int;
DECLARE    done            Int DEFAULT 0;
DECLARE    ArtistCursor       CURSOR FOR
                  SELECT   ArtistID
                  FROM     ARTIST
                  WHERE    Nationality = newNationality;
DECLARE    continue        HANDLER FOR NOT FOUND SET done = 1;
# GET the CustomerID surrogate key value.
SET varCustomerID = LAST_INSERT_ID();
# Create intersection record for each appropriate Artist.
OPEN ArtistCursor
REPEAT
      FETCH ArtistCursor INTO varArtistID;
      IF NOT done THEN
            INSERT INTO CUSTOMER_ARTIST_INT (ArtistID, CustomerID)
                  VALUES(varArtistID, varCustomerID);
      END IF;
UNTIL done END REPEAT;
CLOSE ArtistCursor;
```

In this code, the ArtistCursor loops through the set of ArtistID values for French artists as long as the value of the variable *done* is equal to 0. When the cursor has covered all the appropriate data, an SQL error occurs ("NOT FOUND" in the code), which is handled by the handler named *continue*, which sets the value of *done* to 1. This ends the repeat, and the cursor is closed.

SQL Output Statements

Unlike SQL Server and Oracle Database, MySQL has no output commands, such as PRINT, that we can use to construct visible output for our procedures. However, MySQL does allow us to use a SELECT statement followed by a character string enclosed in single quotes (') as an output device. We can also select a variable, which will return the current value of that variable.

User-Defined Functions

As we discussed in Chapter 7, a **user-defined function** (also known as a **stored function**) is a stored set of SQL statements that:

- is *called by name* from another SQL statement,
- may have *input parameters* passed to it by the calling SQL statement, and
- *returns an output value* to the SQL statement that called the function.

The logical process flow of a user-defined function is illustrated in Figure 7-20. SQL/PSM user-defined functions are very similar to the SQL built-in functions (COUNT, SUM, AVG, MAX, and MIN that we discussed and used in Chapter 2, except that, as the name implies, we create them ourselves to perform specific tasks that we need to do.

In Chapter 7, we used as our example the common problem of needing a name in the format *LastName, FirstName* (including the comma) in a report when the database stores the basic data in two fields named *FirstName* and *LastName*. Here we will build a user-defined function for a similar task: concatenating the *FirstName* and *LastName* fields into a name in the format *FirstName LastName* (including the space!). This construct is commonly used, for example, in mailing labels.

Using the data in the VRG database, we could, of course, simply include the code to do this in an SQL statement (similar to SQL-Query-CH02-37 in Chapter 2) such as:

```
/* *** SQL-Query-CH10C-01 *** */
SELECT      CONCAT(FirstName, ' ', LastName) AS CustomerName,
            Street, City, State, ZIPorPostalCode
FROM        CUSTOMER
ORDER BY    CustomerName;
```

This produces the desired results, but at the expense of working out some cumbersome coding:

CustomerName	Street	City	State	ZIPorPostalCode
Chris Wilkens	87 Highland Drive	Olympia	WA	98508
David Smith	813 Tumbleweed Lane	Loveland	CO	81201
Donald Gray	55 Bodega Ave	Bodega Bay	CA	94923
Fred Smathers	10899 88th Ave	Bainbridge Island	WA	98110
Jeffrey Janes	123 W. Elm St	Renton	WA	98055
Lynda Johnson	117 C Street	Washington	DC	20003
Mary Beth Frederickson	25 South Lafayette	Denver	CO	80201
Selma Warning	205 Burnaby	Vancouver	BC	V6Z 1W2
Susan Wu	105 Locust Ave	Atlanta	GA	30322
Tiffany Twilight	88 1st Avenue	Langley	WA	98260

The alternative is to create a user-defined function to store this code. Not only does this make it easier to use, but it also makes it available for use in other SQL statements. Figure 10C-58 shows a user-defined function named *FirstNameFirst* written in MySQL SQL for use with MySQL 5.6, and the SQL code for the function uses, as we would expect, specific syntax requirements for MySQL 5.6 SQL:

- The function is created and stored in the database by using the **MySQL CREATE FUNCTION statement**.
- The variable names of both the input parameters and the returned output value start with *var* to indicate that these are variable names.
- The concatenation syntax is MySQL SQL syntax.

Now that we have created and stored the user-defined function, we can use it in SQL-Query-CH10C-02:

```
/* *** SQL-Query-CH10C-02 *** */
SELECT      FirstNameFirst(FirstName, LastName) AS CustomerName,
            Street, City, State, ZIPorPostalCode
FROM        CUSTOMER
ORDER BY    CustomerName;
```

FIGURE 10C-58

The SQL Statements for the FirstNameFirst User-Defined Function

```
DELIMITER //

CREATE FUNCTION FirstNameFirst
-- These are the input parameters
       (
        varFirstName    Char(25),
        varLastName     Char(25)
       )
RETURNS Varchar(60) DETERMINISTIC
BEGIN
    -- This is the variable that will hold the value to be returned
    DECLARE   varFullName   Varchar(60);

    -- SQL statements to concatenate the names in the proper order
    SET varFullName = CONCAT(varFirstName, ' ', varLastName);

    -- Return the concatenated name
    RETURN varFullName;

END
//

DELIMITER ;
```

Now we have a function that produces the results we want, which of course are identical to the results for SQL-QUERY-CH10C-01 above:.

CustomerName	Street	City	State	ZIPorPostalCode
Chris Wilkens	87 Highland Drive	Olympia	WA	98508
David Smith	813 Tumbleweed Lane	Loveland	CO	81201
Donald Gray	55 Bodega Ave	Bodega Bay	CA	94923
Fred Smathers	10899 88th Ave	Bainbridge Island	WA	98110
Jeffrey Janes	123 W. Elm St	Renton	WA	98055
Lynda Johnson	117 C Street	Washington	DC	20003
Mary Beth Frederickson	25 South Lafayette	Denver	CO	80201
Selma Warning	205 Burnaby	Vancouver	BC	V6Z 1W2
Susan Wu	105 Locust Ave	Atlanta	GA	30322
Tiffany Twilight	88 1st Avenue	Langley	WA	98260

The advantage of having a user-defined function is that we can now use it whenever we need to without having to re-create the code. For example, our previous query used data in the View Ridge Gallery CUSTOMER table, but we could just as easily use the function with the data in the ARTIST table:

```
/* *** SQL-Query-CH10C-03 *** */
SELECT      FirstNameFirst(FirstName, LastName) AS ArtistName,
            DateOfBirth, DateDeceased
FROM        ARTIST
ORDER BY    ArtistName;
```

This query produces the expected results:

ArtistName	DateofBirth	DateDeceased
Henri Matisse	1869	1954
Joan Miro	1893	1983
John Singer Sargent	1856	1925
Marc Chagall	1887	1985
Mark Tobey	1890	1976
Morris Graves	1920	2001
Paul Horiuchi	1906	1999
Paul Klee	1879	1940
Wassily Kandinsky	1866	1944

We can even use the function multiple times in the same SQL statement, as shown in SQL-Query-CH10C-04, which is a variant on the SQL query we used to create the SQL view CustomerInterestsView in our discussion of SQL views in this chapter:

```
/* *** SQL-Query-CH10C-04 *** */
SELECT      FirstNameFirst(C.FirstName, C.LastName) AS CustomerName,
            FirstNameFirst(A.FirstName, A.LastName) AS ArtistName
FROM        CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CAI
      ON    C.CustomerID = CAI.CustomerID
                JOIN    ARTIST AS A
                  ON    CAI.ArtistID = A.ArtistID
ORDER BY    CustomerName, ArtistName;
```

This query produces the expected large result that is shown in Figure 10C-59, where we see both CustomerName and ArtistName display the names in the FirstName LastName syntax produced by the *FirstNameFirst* user-defined function. Compare the results in this figure to those in Figure 10C-57, which present similar results, but without the formatting provided by the *FirstNameFirst* function.

Having dealt with the problem of concatenating two separate name values into one, let's consider the opposite problem: separating a combined name into separate elements. This is a problem that commonly occurs when using data provided to us in a Microsoft Excel worksheet, where the user simply put an entire name into one cell. As a practical example of this, consider the VRG *postcards_temp* table we imported in our discussion of how to import Microsoft Excel data into a MySQL table. Looking at Figure 10C-57, we can see that the ArtistName column contains the combined artist name in *LastName, FirstName* format.

Because a best practice of database design is to decompose data like this into its separate elements, we have the problem of breaking this data into *LastName* and *FirstName*. We can use a user-defined function to do this.

Note that the delineator or separator between LastName and FirstName is a comma. We can search for the comma using the MySQL built-in character string function **LOCATE**, which will return the numeric position of the comma. The full syntax of the function is:

```
LOCATE (ExpressionToFind, ExpressionToSearch [, StartLocation])
```

In the *postcards_temp* table, we want to find the comma (",") in ArtistName starting at the default location of 1 (character strings are counted starting at 1, not 0).

FIGURE 10C-59

Results of SQL Query Using the FirstNameFirst User-Defined Function

CustomerName	ArtistName
Chris Wilkens	Mark Tobey
Chris Wilkens	Morris Graves
Chris Wilkens	Paul Horiuchi
David Smith	Henri Matisse
David Smith	Joan Miro
David Smith	John Singer Sargent
David Smith	Marc Chagall
David Smith	Wassily Kandinsky
Donald Gray	Mark Tobey
Donald Gray	Morris Graves
Donald Gray	Paul Horiuchi
Fred Smathers	Mark Tobey
Fred Smathers	Morris Graves
Fred Smathers	Paul Horiuchi
Jeffrey Janes	Mark Tobey
Jeffrey Janes	Morris Graves
Jeffrey Janes	Paul Horiuchi
Mary Beth Frederickson	Henri Matisse
Mary Beth Frederickson	Joan Miro
Mary Beth Frederickson	Marc Chagall
Mary Beth Frederickson	Wassily Kandinsky
Melinda Bench	John Singer Sargent
Michael Bench	Henri Matisse
Michael Bench	Marc Chagall
Selma Warning	John Singer Sargent
Selma Warning	Marc Chagall
Selma Warning	Morris Graves
Tiffany Twilight	John Singer Sargent
Tiffany Twilight	Mark Tobey
Tiffany Twilight	Morris Graves
Tiffany Twilight	Paul Horiuchi

Once we have found the comma we can retrieve the last name by using the MySQL built-in character string function **SUBSTRING**, which will return a subset of the characters from a character string. The full syntax of the function is:

```
SUBSTRING (ExpressionToSearch, StartLocation, Length)
```

In the *postcards_temp* table, we want to return the subset of ArtistName starting at 1 and ending one character below the comma ([Value returned by LOCATE] - 1). We put these together into a user-defined function named *GetLastNameCommaSeparated* as shown in Figure 10C-60.

Now that we have created and stored the user-defined function, we can use it in SQL-Query-CH10C-05:

```
/* *** SQL-Query-CH10C-05 *** */
SELECT      ArtistName,
            GetLastNameCommaSeparated(ArtistName) AS ArtistLastName
FROM        POSTCARDS_TEMP
ORDER BY    ArtistName;
```

FIGURE 10C-60
The SQL Statements for the GetLastNameCommaSeparated User-Defined Function

```
DELIMITER //

CREATE FUNCTION GetLastNameCommaSeparated

-- These are the input parameters
(
    varName          VARCHAR(25)
)
RETURNS VARCHAR(25) DETERMINISTIC
BEGIN
    -- This is the variable that will hold the value to be returned
    DECLARE varLastName     VARCHAR(25);

    -- This is the variable that will hold the position of the comma
    DECLARE varIndexValue   INT;

    -- SQL statement to find the comma separator

    SET varIndexValue = LOCATE(',', varName);

    -- SQL statement to determine last name
    SET varLastName = SUBSTRING(varName, 1, (varIndexValue  - 1));

    -- Return the last name
    RETURN varLastName;
END
//

DELIMITER ;
```

The results are shown in Figure 10C-61, and are exactly what we want, with the ArtistLastName column containing only the last name.

Now that we can determine the last name of the artists in the *postcards_temp* table, let's return to our discussion of that table, and how we will integrate the data in it into the VRG database. Because the ARTIST table uses ArtistID as the primary key, and WORK uses WorkID for the primary key, we have to find some way of associating these primary key values with the data in *postcards_temp*. We can use the *GetLastNameCommaSeparated* user-defined function to help us do this.

First, we need to alter the *postcards_temp* table by adding a column named ArtistLastName to hold the last name values. The full discussion of how to do this is in Chapter 8 on database redesign, where we discuss how to use the **SQL ALTER TABLE statement**. Here, we will use it to add an ArtistLastName column and an ArtistID column.

```
/* *** SQL-ALTER-TABLE-CH10C-08 *** */
ALTER TABLE POSTCARDS_TEMP
   ADD      ArtistLastName Char(25) NULL;
/* *** SQL-ALTER-TABLE-CH10C-09 *** */
ALTER TABLE POSTCARDS_TEMP
   ADD      ArtistID Int NULL;
```

Note that we are allowing the values of both of these columns to be NULL, because we have not inserted any data and therefore we cannot create them as NOT NULL even if that is what we ultimately want them to be (see Chapter 8 for a discussion of the steps to add a NOT NULL column to a table). The *postcards_temp* table now appears as shown in Figure 10C-62, with the two new columns displayed at the right side of the table.

But before we create the SQL statements to modify the postcards_temp table data, we need to discuss a default setting used in MySQL Workbench that will affect what we are trying to do. By default, MySQL Workbench does not allow the execution of SQL UPDATE and DELETE statements unless these statements have an SQL WHERE clause.

FIGURE 10C-61
The Results for SQL-Query-CH10C-05

ArtistName	ArtistLastName
Chagall, Marc	Chagall
Chagall, Marc	Chagall
Graves, Morris	Graves
Graves, Morris	Graves
Graves, Morris	Graves
Graves, Morris	Graves
Graves, Morris	Graves
Horiuchi, Paul	Horiuchi
Horiuchi, Paul	Horiuchi
Horiuchi, Paul	Horiuchi
Horiuchi, Paul	Horiuchi
Kandinsky, Wassily	Kandinsky
Klee, Paul	Klee
Matisse, Henri	Matisse
Matisse, Henri	Matisse
Sargent, John Singer	Sargent
Sargent, John Singer	Sargent
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey
Tobey, Mark	Tobey

This setting is called **safe updates**. As we learned in our Chapter 7 discussion of SQL DML actions, an UPDATE statement without a WHERE clause will affect *every* row in the table, and a DELETE statement without a WHERE clause will remove *every* row in a table. Not a good idea!

To prevent this from inadvertently happening, MySQL simply won't let you do it. However, when altering the structure of a table it can be necessary to UPDATE every row in the table. Therefore, before modifying the postcards_temp table data, we need to disable safe updates.

Figure 10C-63 shows the SQL statements we will use to alter the postcards_temp table data, which we will discuss below. It also shows the location of the **Reconnect to DBMS button** and the **Edit | Preferences command**, which is used to open the MySQL Workbench Preference dialog box.

Use the Edit | Preferences command to open the MySQL Workbench Preference dialog box, which appears as shown in Figure 10C-64. In the MySQL Workbench Preference dialog box, click SQL Editor tab, then uncheck the **Forbid UPDATEs and DELETEs with no key in WHERE clause or no LIMIT clause checkbox**, and then click the **OK** button. Finally, click the Reconnect to DBMS button to reset the connection between the MySQL Workbench and the DBMS.

postcards_temp_id	ArtistName	WorkTitle	PostcardSize	PostcardPrice	QuantityOnHand	QuantityOnOrder	ArtistLastName	ArtistID
1	Chagall, Marc	I and the Village	4" x 6"	3.00	3	10	NULL	NULL
2	Chagall, Marc	The Fiddler	4" x 6"	3.00	3	10	NULL	NULL
3	Graves, Morris	Night Bird	4" x 6"	3.00	3	10	NULL	NULL
4	Graves, Morris	Sunflower	4" x 6"	3.00	3	10	NULL	NULL
5	Graves, Morris	Surf and Bird	3" x 5"	2.50	5	0	NULL	NULL
6	Graves, Morris	Surf and Bird	4" x 6"	3.00	10	0	NULL	NULL
7	Graves, Morris	Surf and Bird	7" x 10"	5.00	8	0	NULL	NULL
8	Horiuchi, Paul	Color Floating in Time	4" x 6"	3.00	3	10	NULL	NULL
9	Horiuchi, Paul	Into Time	4" x 6"	3.00	2	10	NULL	NULL
10	Horiuchi, Paul	Memories IV	4" x 6"	3.00	2	10	NULL	NULL
11	Horiuchi, Paul	Yellow Covers Blue	4" x 6"	3.00	3	10	NULL	NULL
12	Kandinsky, Wassily	Der Blaue Reiter	4" x 6"	3.00	5	0	NULL	NULL
13	Klee, Paul	Angelus Novus	4" x 6"	3.00	4	10	NULL	NULL
14	Matisse, Henri	The Dance	4" x 6"	3.00	3	10	NULL	NULL
15	Matisse, Henri	Woman with a Hat	4" x 6"	3.00	2	10	NULL	NULL
16	Sargent, John Singer	Claude Monet Painting	4" x 6"	3.00	3	10	NULL	NULL
17	Sargent, John Singer	Spanish Dancer	4" x 6"	3.00	3	10	NULL	NULL
18	Tobey, Mark	Blue Interior	4" x 6"	3.00	5	0	NULL	NULL
19	Tobey, Mark	Broadway Boggie	4" x 6"	3.00	5	0	NULL	NULL
20	Tobey, Mark	Farmer's Market #2	3" x 5"	2.50	3	10	NULL	NULL
21	Tobey, Mark	Farmer's Market #2	4" x 6"	3.00	10	10	NULL	NULL
22	Tobey, Mark	Farmer's Market #2	7" x 10"	5.00	7	10	NULL	NULL
23	Tobey, Mark	Forms in Progress I	4" x 6"	3.00	5	0	NULL	NULL
24	Tobey, Mark	Forms in Progress II	4" x 6"	3.00	2	10	NULL	NULL
25	Tobey, Mark	The Woven World	4" x 6"	3.00	5	0	NULL	NULL
26	Tobey, Mark	Universal Field	4" x 6"	3.00	5	0	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

FIGURE 10C-62

The POSTCARDS_TEMP Table with the Added Columns

Now we are ready populate the postcards_temp table. The SQL statements to do this are:

```
/* *** SQL-UPDATE-CH10C-01 *** */
UPDATE      POSTCARDS_TEMP
   SET      ArtistLastName = GetLastNameCommaSeparated(ArtistName);

/* *** SQL-UPDATE-CH10C-02 *** */
UPDATE      POSTCARDS_TEMP
   SET      ArtistID =
            (SELECT  ArtistID
             FROM    ARTIST
             WHERE   ARTIST.LastName = POSTCARDS_TEMP.ArtistLastName);
```

Note the use of the *GetLastNameCommaSeparated* user-defined function in the SQL-UPDATE-CH10C-01. Also note the use of an SQL subquery in SQL-UPDATE-CH10C-02, which illustrates that SQL subqueries can be used in SQL statements beyond just the SQL SELECT statement. In fact, we can use an SQL subquery in INSERT, UPDATE, DELETE and MERGE statements, and it is often exactly what we need! The updated *postcards_temp* table is shown in Figure 10C-65, with the new column data displayed in the new columns. We still have work to do to integrate the *postcards_temp* table data into the VRG database, and we will continue that work in Review Question 10C.37.

At this point, we need to reset the safe updates setting back to its default–leaving it disabled is asking for trouble! Repeat the steps discussed above, but this time check the **Forbid UPDATEs and DELETEs with no key in WHERE clause or no LIMIT clause** checkbox.

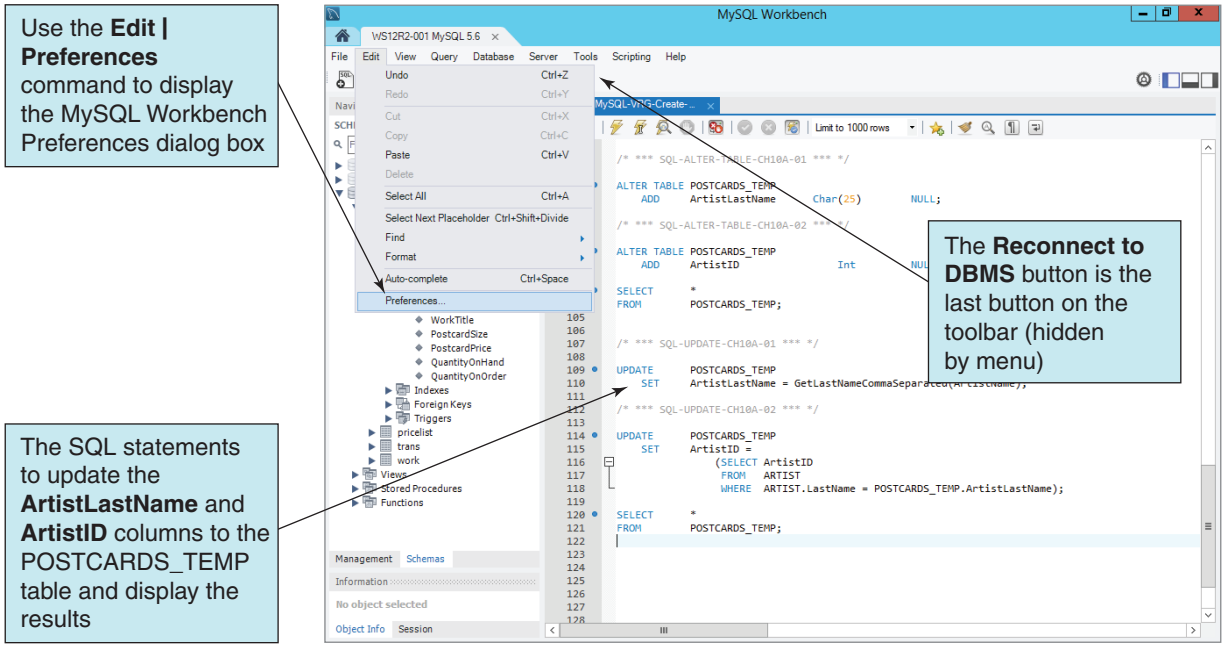


FIGURE 10C-63

The Edit | Preferences Command

Stored Procedures

As with other database structures, you can write a stored procedure in an SQL script text file and process the commands using the MySQL Workbench. However, there is one little gotcha. The first time you create a stored procedure in an SQL script, use the **SQL CREATE PROCEDURE statement**. Subsequently, if you change the procedure, you use the **SQL ALTER PROCEDURE statement** for partial modifications. For large changes, use the **SQL DROP PROCEDURE statement** to delete the procedure and then re-create it.

The Stored Procedure InsertCustomerAndInterests

In our discussion of MySQL procedures, we used as our example the need to enter data for a new customer and the artists of interest to that customer. The code segments we wrote were

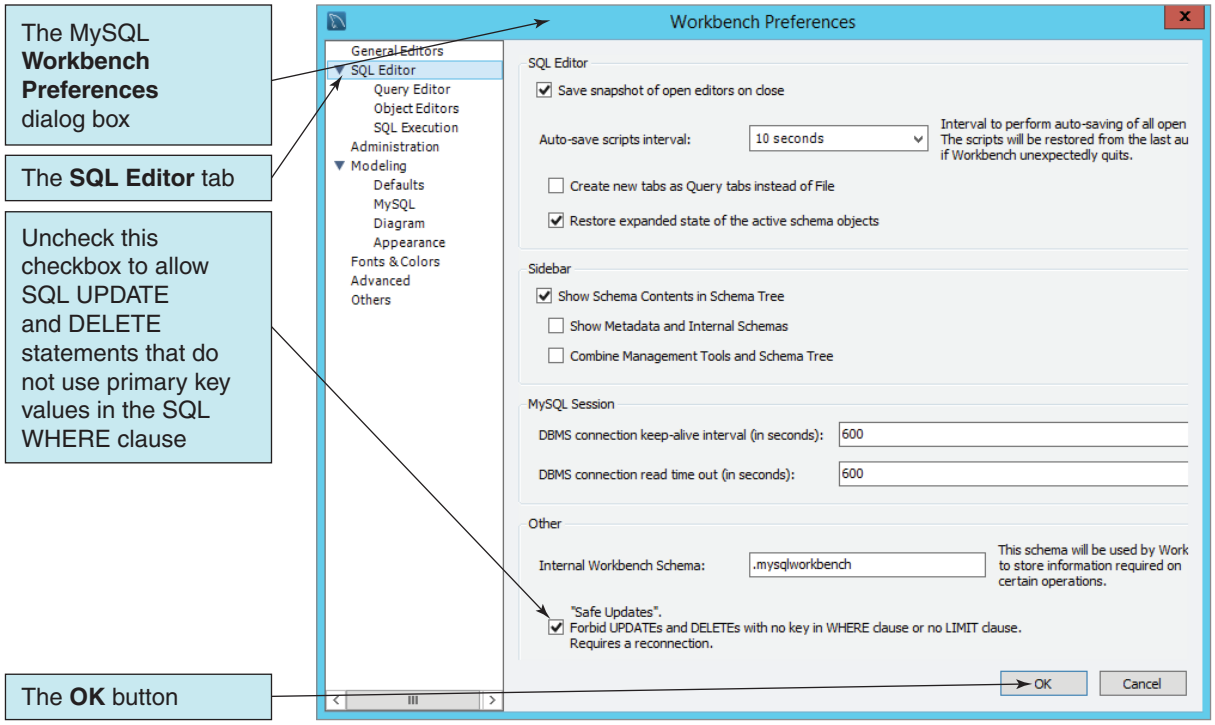


FIGURE 10C-64

Allowing UPDATES without an SQL WHERE Clause

postcards_temp_id	ArtistName	WorkTitle	PostcardSize	PostcardPrice	QuantityOnHand	QuantityOnOrder	ArtistLastName	ArtistID
1	Chagall, Marc	I and the Village	4" x 6"	3.00	3	10	Chagall	5
2	Chagall, Marc	The Fiddler	4" x 6"	3.00	3	10	Chagall	5
3	Graves, Morris	Night Bird	4" x 6"	3.00	3	10	Graves	19
4	Graves, Morris	Sunflower	4" x 6"	3.00	3	10	Graves	19
5	Graves, Morris	Surf and Bird	3" x 5"	2.50	5	0	Graves	19
6	Graves, Morris	Surf and Bird	4" x 6"	3.00	10	0	Graves	19
7	Graves, Morris	Surf and Bird	7" x 10"	5.00	8	0	Graves	19
8	Horiuchi, Paul	Color Floating in Time	4" x 6"	3.00	3	10	Horiuchi	18
9	Horiuchi, Paul	Into Time	4" x 6"	3.00	2	10	Horiuchi	18
10	Horiuchi, Paul	Memories IV	4" x 6"	3.00	2	10	Horiuchi	18
11	Horiuchi, Paul	Yellow Covers Blue	4" x 6"	3.00	3	10	Horiuchi	18
12	Kandinsky, Wassily	Der Blaue Reiter	4" x 6"	3.00	5	0	Kandinsky	2
13	Klee, Paul	Angelus Novus	4" x 6"	3.00	4	10	Klee	3
14	Matisse, Henri	The Dance	4" x 6"	3.00	3	10	Matisse	4
15	Matisse, Henri	Woman with a Hat	4" x 6"	3.00	2	10	Matisse	4
16	Sargent, John Singer	Claude Monet Painting	4" x 6"	3.00	3	10	Sargent	11
17	Sargent, John Singer	Spanish Dancer	4" x 6"	3.00	3	10	Sargent	11
18	Tobey, Mark	Blue Interior	4" x 6"	3.00	5	0	Tobey	17
19	Tobey, Mark	Broadway Boggie	4" x 6"	3.00	5	0	Tobey	17
20	Tobey, Mark	Farmer's Market #2	3" x 5"	2.50	3	10	Tobey	17
21	Tobey, Mark	Farmer's Market #2	4" x 6"	3.00	10	10	Tobey	17
22	Tobey, Mark	Farmer's Market #2	7" x 10"	5.00	7	10	Tobey	17
23	Tobey, Mark	Forms in Progress I	4" x 6"	3.00	5	0	Tobey	17
24	Tobey, Mark	Forms in Progress II	4" x 6"	3.00	2	10	Tobey	17
25	Tobey, Mark	The Woven World	4" x 6"	3.00	5	0	Tobey	17
26	Tobey, Mark	Universal Field	4" x 6"	3.00	5	0	Tobey	17
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

FIGURE 10C-65

The POSTCARDS_TEMP Table with the Populated Columns

very specifically tied to the data we used and thus of limited use. Is there a way to write a general block of code that could be used for more than one customer? Yes, and that block of code is a stored procedure.

Figure 10C-66 shows the SQL code for the InsertCustomerAndInterests stored procedure. This stored procedure generalizes our previous code and can be used to insert data for any new customer into CUSTOMER and then store data for that customer in CUSTOMER_ARTIST_INT, linking the customer to all artists having a particular nationality.

Six parameters are input to the procedure: newLastName, newFirstName, newEmailAddress, newAreaCode, newPhoneNumber, and newNationality. The first five parameters are the new customer data, and the sixth one is the nationality of the artists for which the new customer has an interest. The stored procedure also uses three variables: varRowCount, varArtistID, and varCustomerID. These variables are used to store values of the number of rows, the value of the ArtistID primary key, and the value of the CustomerID primary key, respectively.

The first task performed by this stored procedure is to determine whether the customer already exists. If the value of varRowCount in the first SELECT statement is greater than zero, a row for that customer already exists. In this case, nothing is done, and the stored procedure prints an error message (using the SELECT command) and exits (using the ROLLBACK command). The error message is visible in the MySQL Workbench, but it generally would not be visible to application programs that invoked this procedure. Instead, a parameter or other facility needs to be used to return the error message back to the user via the application program. This topic is beyond the scope of the present discussion, but we will send a message back to the MySQL Workbench to mimic such actions and provide a means to make sure our stored procedures are working correctly.

If the customer does not already exist, the procedure inserts the new data into the table CUSTOMER, and then the new value of CustomerID is read into the variable varCustomerID.

```
DELIMITER //

CREATE PROCEDURE InsertCustomerAndInterests
       (IN newLastName          Char(25),
        IN newFirstName         Char(25),
        IN newEmailAddress      Varchar(100),
        IN newAreaCode          Char(3),
        IN newPhoneNumber       Char(8),
        IN newNationality       Char(30))

BEGIN

  DECLARE     varRowCount          Int;
  DECLARE     varArtistID          Int;
  DECLARE     varCustomerID        Int;
  DECLARE     done                 Int DEFAULT 0;
  DECLARE     ArtistCursor     CURSOR FOR
                    SELECT         ArtistID
                    FROM           ARTIST
                    WHERE          Nationality=newNationality;
  DECLARE     continue         HANDLER FOR NOT FOUND SET done = 1;

  # Check to see if Customer already exists in database

   SELECT     COUNT(*) INTO varRowCount
   FROM       CUSTOMER
   WHERE      LastName = newLastName
      AND     FirstName = newFirstName
      AND     EmailAddress =   newEmailAddress
      AND     AreaCode = newAreaCode
      AND     PhoneNumber = newPhoneNumber;

   # IF (varRowCount > 0) THEN Customer already exists.
   IF (varRowCount > 0) THEN
      ROLLBACK;
      SELECT 'Customer already exists';
      END IF;

   # IF (varRowCount = 0) THEN Customer does not exist.
   # Insert new Customer data.

   IF (varRowCount = 0) THEN
      INSERT INTO CUSTOMER (LastName, FirstName, EmailAddress, AreaCode, PhoneNumber)
         VALUES(newLastName, newFirstName,  newEmailAddress, newAreaCode,
newPhoneNumber);

      # Get new CustomerID surrogate key value.

      SET varCustomerID = LAST_INSERT_ID();

      # Create intersection record for each appropriate Artist.

      OPEN   ArtistCursor;
             REPEAT
             FETCH ArtistCursor INTO varArtistID;
             IF NOT done THEN
                INSERT INTO CUSTOMER_ARTIST_INT (ArtistID, CustomerID)
                   VALUES(varArtistID, varCustomerID);
             END IF;
             UNTIL done END REPEAT;
      CLOSE  ArtistCursor;
      SELECT 'New customer and artist interest data added to database.'
         AS InsertCustomerAndInterstsResults;
      END IF;
END
//

DELIMITER ;
```

FIGURE 10C-66

The SQL Statements for the InsertCustomerAndInterests Stored Procedure

This is done using the **MySQL function LAST_INSERT_ID()**, which provides the value of the most recently created surrogate key value.

To create the appropriate intersection table rows, an SQL cursor named ArtistCursor is created on an SQL statement that obtains all ARTIST rows where Nationality equals the parameter newNationality. The cursor is opened and positioned on the first row by the first FETCH statement, and then the cursor is processed in a REPEAT loop. In this loop, statements between REPEAT and UNTIL done END REPEAT are iterated until MySQL signals the end of data by having the error handler named *continue* set the value of the variable *done* to 1. At each iteration of the REPEAT loop, a new row is inserted into the intersection table CUSTOMER_ ARTIST_INT. The FETCH…INTO statement at the beginning of the block is used to move the cursor to the next row.

To invoke the InsertCustomerAndInterests stored procedure for Michael Bench, we use the following SQL statement:

```
/* *** SQL-CALL-CH10C-01 *** */
CALL InsertCustomerAndInterests ('Bench', 'Michael',
'Michael.Bench@somewhere.com',
'206', '876-8822', 'French');
```

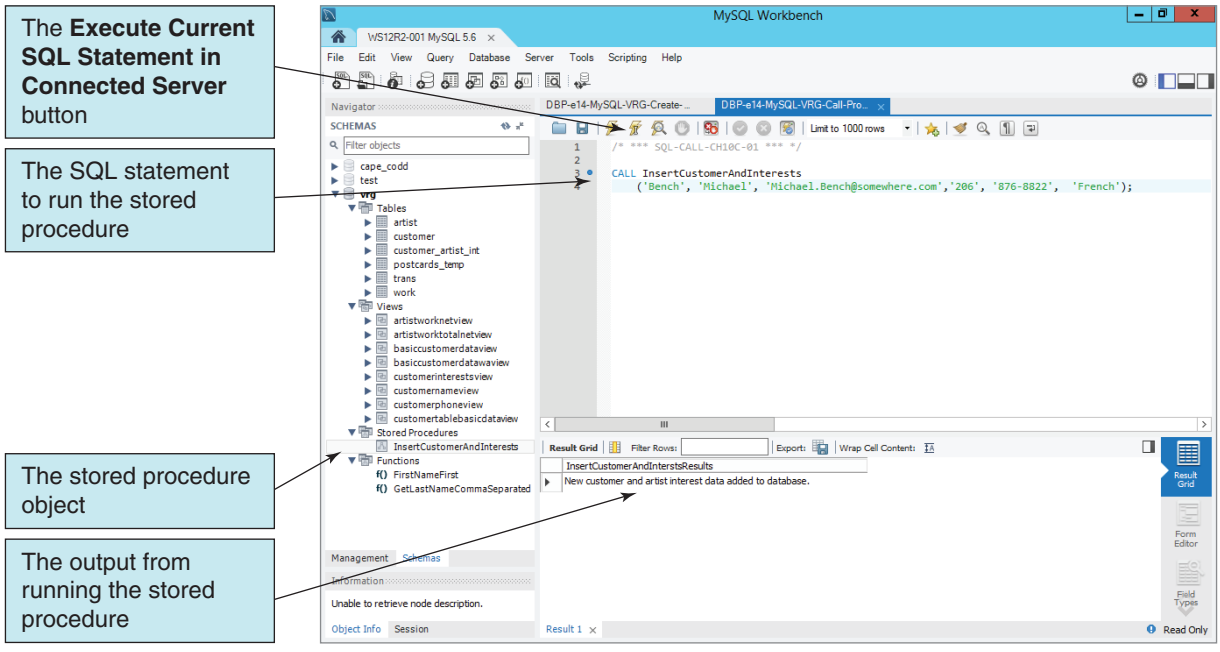
Figure 10C-67 shows the execution of the stored procedure in the MySQL Workbench. Notice how there are two tabbed windows with output from this stored procedure. The first tab, with hidden output, simply displays the calculated value of one of the variables in the procedure. The second tab, with the output shown in Figure 10C-67, shows that our SELECT command has produced the necessary output so we can see what actions were taken. If we now wanted to check the tables themselves, we could do so, but that is not necessary at this point. The output in Figure 10C-67 shows that the customer and artist interest data have been added to the database.

Rather than check each table individually, we can see the results of using this stored procedure by looking at the data (the new records for Michael Bench) as they are viewed in the CustomerInterestsView output using the SQL query:

```
/* *** SQL-Query-View-CH10C-02 *** */
SELECT      *
FROM        CustomerInterestsView
ORDER BY    CustomerLastName, CustomerFirstName;
```

FIGURE 10C-67

Running the InsertCustomerAndInterests Stored Procedure



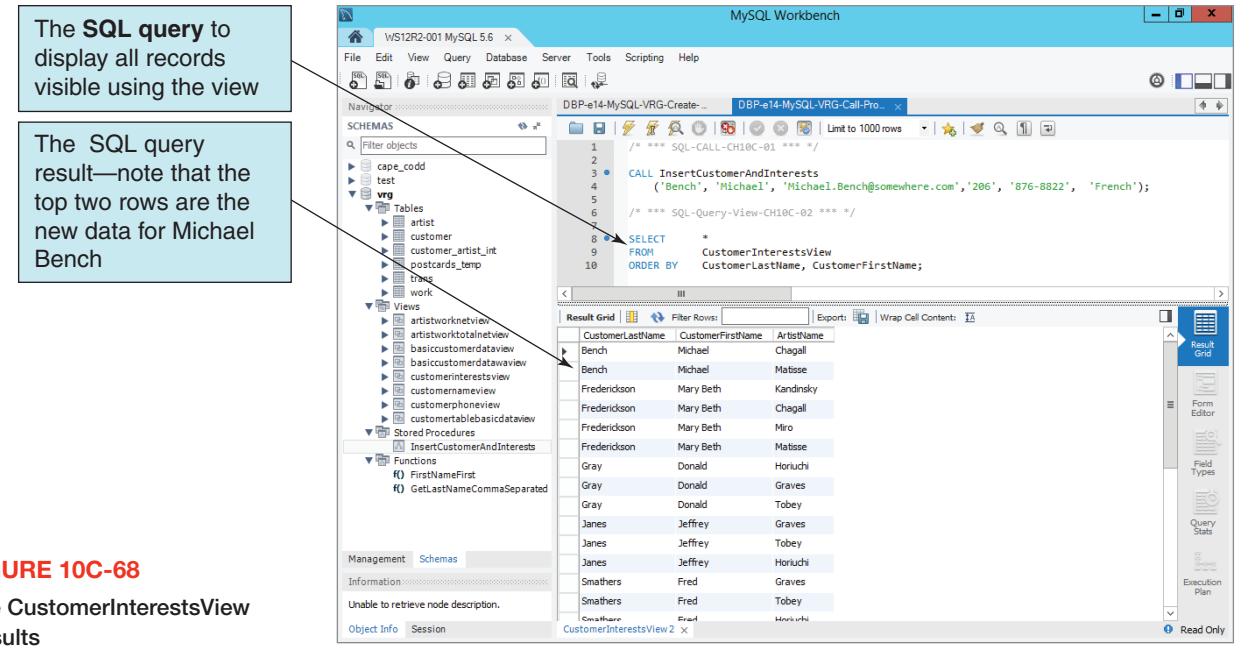


FIGURE 10C-68

The CustomerInterestsView Results

Figure 10C-68 shows the CustomerInterestsView result and clearly shows that Michael Bench and his interest in French artists (Chagall and Matisse) are now in the VRG database.

The Stored Procedure InsertCustomerWithTransaction

Now we will write a stored procedure that inserts data for a new customer, records a purchase, and creates an entry in the CUSTOMER_ARTIST_INT table. We will name this stored procedure *InsertCustomerWithTransaction*, and the necessary SQL code is shown in Figure 10C-69. This procedure receives nine parameters having data about the new customer and about the customer's purchase. We will use this procedure to discuss transaction processing in MySQL.

The first action is to see whether the customer already exists. If so, the procedure exits with an error message. If the customer does not exist, then the procedure starts a transaction with the SQL START TRANSACTION command. Recall from Chapter 9 that transactions ensure that all of the database activity is committed atomically; either all of the updates occur or none of them do. The transaction begins, and the new customer row is inserted. The new value of CustomerID is obtained, as shown in the InsertCustomerWithInterests stored procedure. Next, the procedure checks to determine whether ArtistID, WorkID, and TransactionID are valid. If any are invalid, the transaction is rolled back using the SQL ROLLBACK command.

If all of the surrogate key values are valid, two actions in the transaction are completed. First, an UPDATE statement updates DateSold, SalesPrice, and CustomerID in the appropriate TRANS row. DateSold is set to system date via the **MySQL CURRENT_DATE() function**, SalesPrice is set to the value of transTransSalesPrice, and CustomerID is set to the value of varCustomerID. Second, a row is added to CUSTOMER_ARTIST_INT to record the customer's interest in this artist.

If everything proceeds normally to this point, the transaction is committed using the **SQL COMMIT command**. After, and only after, the transaction is committed, we print our results message.

```
DELIMITER //

CREATE PROCEDURE InsertCustomerWithTransaction
          (IN    newCustomerLastName           Char(25),
           IN    newCustomerFirstName          Char(25),
           IN    newCustomerEmailAddress       Varchar(100),
           IN    newCustomerAreaCode           Char(3),
           IN    newCustomerPhoneNumber        Char(8),
           IN    transArtistLastName           Char(25),
           IN    transWorkTitle                Char(35),
           IN    transWorkCopy                 Char(12),
           IN    transTransSalesPrice          Numeric(8,2))

spicwt:BEGIN

   DECLARE   varRowCount         Int;
   DECLARE   varArtistID         Int;
   DECLARE   varCustomerID       Int;
   DECLARE   varWorkID           Int;
   DECLARE   varTransactionID    Int;

   # Check to see if InsertCustomerWithTransactionCustomer already exists in database

   SELECT COUNT(*) INTO varRowCount
   FROM      CUSTOMER
   WHERE     LastName = newCustomerLastName
      AND    FirstName = newCustomerFirstName
      AND    EmailAddress = newCustomerEmailAddress
      AND    AreaCode = newCustomerAreaCode
      AND    PhoneNumber = newCustomerPhoneNumber;


   # IF (varRowCount > 0) THEN Customer already exists.
   IF (varRowCount > 0)
      THEN
      SELECT 'Customer already exists';
      ROLLBACK;
      LEAVE spicwt;
      END IF;

   # IF varRowCount = 0 THEN Customer does not exist in database.
   IF (varRowCount = 0) THEN
      spicwtif:BEGIN

      # Start transaction - Rollback everything if unable to complete it.
      START TRANSACTION;

      # Insert new Customer data.
      INSERT INTO CUSTOMER (LastName, FirstName, AreaCode, PhoneNumber, EmailAddress)
         VALUES(newCustomerLastName, newCustomerFirstName,
                newCustomerAreaCode, newCustomerPhoneNumber, newCustomerEmailAddress);

      # Get new CustomerID surrogate key value.
      SET varCustomerID = LAST_INSERT_ID();

      # Get ArtistID surrogate key value, check for validity.
      SELECT    ArtistID INTO varArtistID
      FROM      ARTIST
      WHERE     LastName = transArtistLastName;
```

FIGURE 10C-69 (*continued*)

The SQL Statements for the InsertCustomerWithTransaction Stored Procedure

```
            IF (varArtistID IS NULL) THEN
               SELECT 'Invalid ArtistID';
               ROLLBACK;
               LEAVE spicwtif;
               END IF;

         # Get WorkID surrogate key value, check for validity.
            SELECT    WorkID INTO  varWorkID
            FROM      WORK
            WHERE     ArtistID = varArtistID
               AND    Title = transWorkTitle
               AND    Copy = transWorkCopy;

            IF (varWorkID IS NULL) THEN
               SELECT 'Invalid WorkID';
               ROLLBACK;
               LEAVE spicwtif;
               END IF;

            # Get TransID surrogate key value, check for validity.
            SELECT    TransactionID INTO varTransactionID
            FROM      TRANS
            WHERE     WorkID = varWorkID
               AND    SalesPrice IS NULL;

            IF (varTransactionID IS NULL) THEN
               SELECT 'Invalid TransactionID';
               ROLLBACK;
               LEAVE spicwtif;
               END IF;

            # All surrogate key values of OK, complete the transaction
            # Update TRANS row
            UPDATE    TRANS
               SET    DateSold = CURRENT_DATE(),
                      SalesPrice = transTransSalesPrice,
                      CustomerID = varCustomerID
               WHERE  TransactionID = varTransactionID;

            # Commit the Transaction
            COMMIT;

            # Create CUSTOMER_ARTIST_INT row
            INSERT INTO CUSTOMER_ARTIST_INT (CustomerID, ArtistID)
               VALUES(varCustomerID, varArtistID);

            # The transaction is completed. Print message
            SELECT 'The new customer and transaction are now in the database.'
               AS InsertCustomerWithTransactionResults;
            # END spicwtif
            END spicwtif;
      END IF;
   # END spicwt
   END spicwt
   //

   DELIMITER ;
```

FIGURE 10C-69

Continued

To use the InsertCustomerWithTransaction stored procedure, we will record the following purchase by our next new customer, Melinda Gliddens, who just bought a print of John Singer Sargent's *Spanish Dancer* for $350.00. The SQL statement is:

```
/* *** SQL-CALL-CH10C-02 *** */
CALL InsertCustomerWithTransaction ('Gliddens', 'Melinda',
   'Melinda.Gliddens@somewhere.com', '360', '765-8877',
   'Sargent', 'Spanish Dancer', '588/750', 350.00);
```

Figure 10C-70 shows the invocation of this procedure using sample data, and again there are two tabbed windows with output from this stored procedure. The first tab, with hidden output, again displays the calculated value of one of the variables in the procedure, while the second tab, with the output shown in Figure 10C-70, shows that the stored procedure ran successfully. Figure 10C-71 shows that Melinda Gliddens's data are now in the CUSTOMER table by using the SQL query:

```
/* *** SQL-Query-CH10C-06 *** */
SELECT  *
FROM    CUSTOMER;
```

Just as we can use MySQL Workbench to see the tables and views in a MySQL schema, we can also see the stored procedures defined for a schema. In Figures 10C-67 and 10C-70, callouts indicate that the new stored procedures can be displayed in the Stored Procedures folder for the schema.

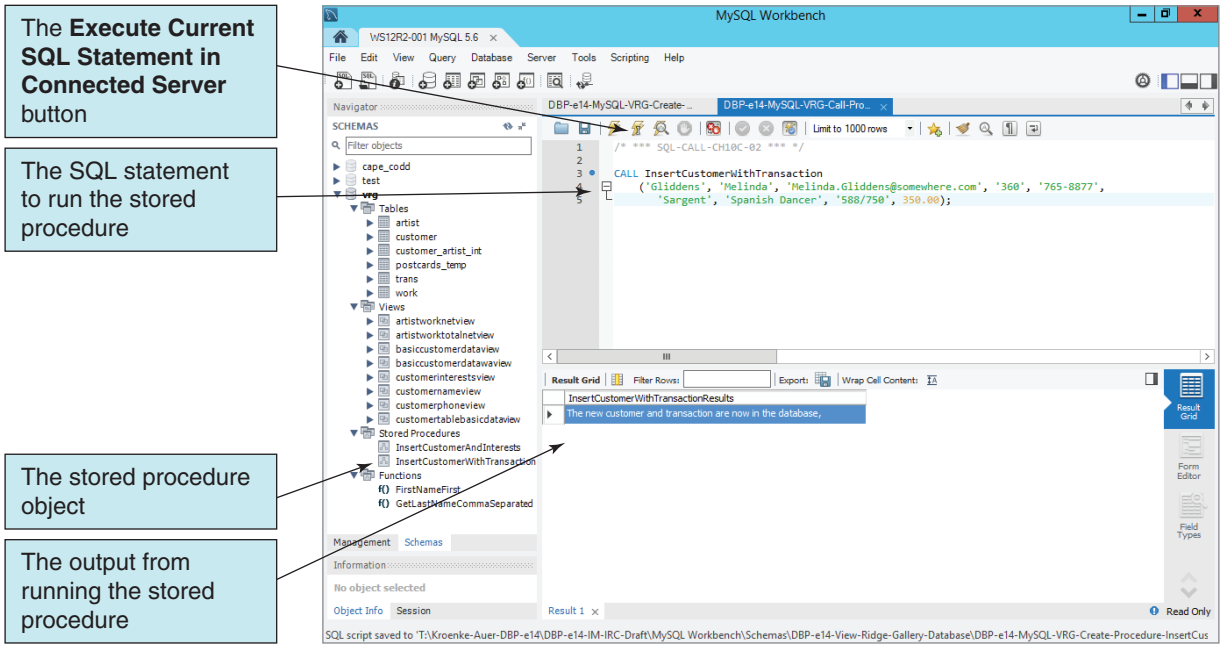
Triggers

MySQL supports the BEFORE and AFTER triggers but not the INSTEAD OF trigger. A table may have one or more BEFORE triggers for insert, update, and delete actions and one or more AFTER triggers for triggering by an insert, an update, or a delete. However, we cannot define two triggers on the same event. For example, we cannot have two BEFORE INSERT triggers.

For insert and update triggers, the new values for every column of the table being processed will be stored in **transition variables** prefixed with the word *NEW*. If, for example,

FIGURE 10C-70

Running the InsertCustomerWithTransaction Stored Procedure



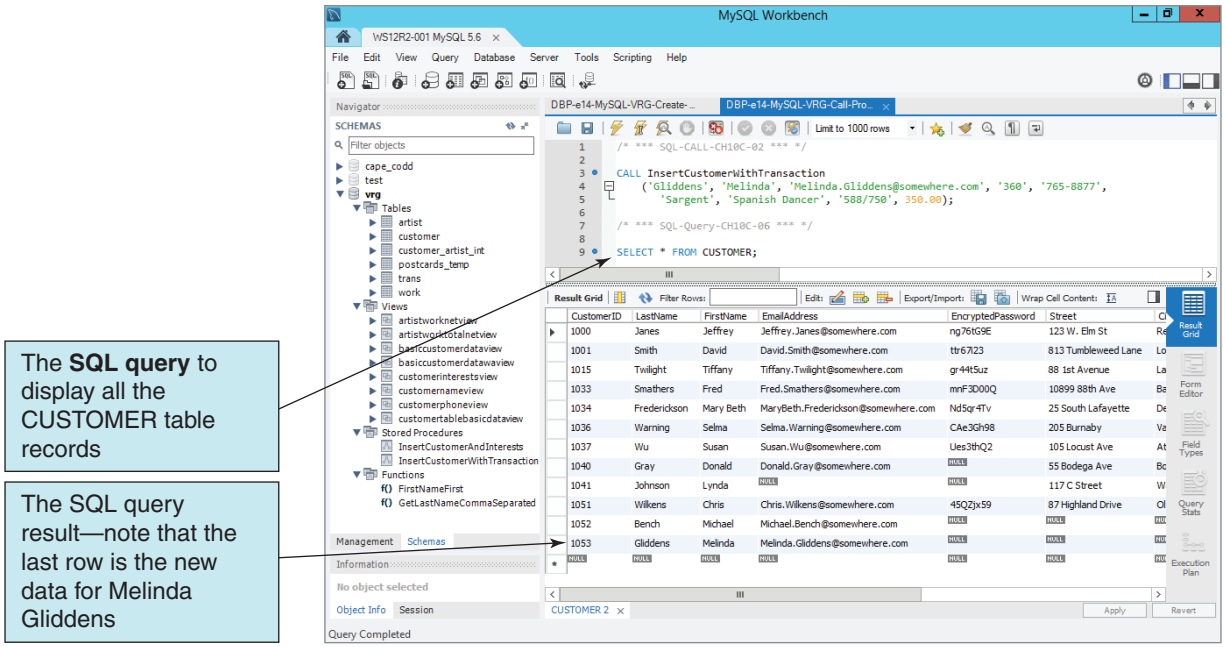


FIGURE 10C-71

The SQL Query Result

a new row is being added to the ARTIST table, the new data can be accessed in five columns named NEW.LastName, NEW.FirstName, NEW.Nationality, NEW.DateOfBirth, and NEW.DateDeceased. Similarly, for update and delete commands, the old values for every column of the table being updated or deleted will be stored in transition variables prefixed with the word *OLD*. You will see how to use these transition variables in the examples that follow.

Chapter 7 described four database functions that can be implemented with triggers. Unfortunately, the trigger capabilities of MySQL have limitations that make it impossible to implement some of the trigger structures discussed in Chapter 7. First, MySQL prohibits using a trigger from making changes to the table that fired the trigger. Second, triggers cannot return an output value—a trigger can use the LEAVE keyword to exit without returning a value, but not the RETURN keyword, which does allow output values. Although there is a work-around, it is not pretty. Third, triggers cannot contain implicit or explicit rollbacks or commits, which means we cannot stop a transaction using a rollback statement in a trigger. Finally, because MySQL does not support INSTEAD OF triggers, we cannot write triggers against views.

The next four sections discuss the four possible trigger functions described in Chapter 7. When we cannot implement a trigger, we will create an equivalent stored procedure as an alternative when possible. We will create these triggers as SQL scripts (using SQL CREATE TRIGGER or SQL ALTER TRIGGER as the first statement) in the MySQL Workbench.

A Trigger for Setting Default Values

Triggers can be used to set default values that are more complex than those that can be set with the Default constraint on a column definition. For example, the View Ridge Gallery has a pricing policy that says that the default AskingPrice of a work of art depends on whether the art has been in the gallery before. If not, the default AskingPrice is twice the AcquisitionPrice. If the work has been in the gallery before, the default price is the larger of twice the AcquisitionPrice or the AcquisitionPrice plus the average net gain of the work in the past.

However, this requires that an action be taken on the same table that fired the trigger—the INSERT into TRANS fires a trigger to calculate the AskingPrice and update the AskingPrice column in TRANS. This is not supported by MySQL triggers. Further, MySQL stored procedures have the same limitation. There is a work-around, but it is somewhat awkward—we will insert the AskingPrice into a separate table named PRICELIST. This work-around creates a duplicate AskingPrice column in the table we do not really want, but it does allow us to use a trigger to make the calculation. Note that this is *not* necessary in SQL Server 2014 or Oracle Database.

The SQL statement to create the PRICELIST table is:

```
/* *** SQL-CREATE-TABLE-CH10C-01 *** */
CREATE TABLE PRICELIST (
          TransactionID  Int                NOT NULL,
          AskingPrice    Numeric(8,2)       NOT NULL,
          CONSTRAINT     PriceListPK        PRIMARY KEY(TransactionID),
          CONSTRAINT     TransPriceListFK   FOREIGN KEY(TransactionID)
                                 REFERENCES TRANS(TransactionID)
                                       ON UPDATE NO ACTION
                                       ON DELETE NO ACTION
          );
```

Now we need to populate the PRICELIST table with our current data from TRANS. We will use a bulk INSERT, as discussed in Chapter 7:

```
/* *** SQL-INSERT-CH10C-03 *** */
INSERT INTO PRICELIST (TransactionID, AskingPrice)
      SELECT  TransactionID, AskingPrice
      FROM    TRANS;
```

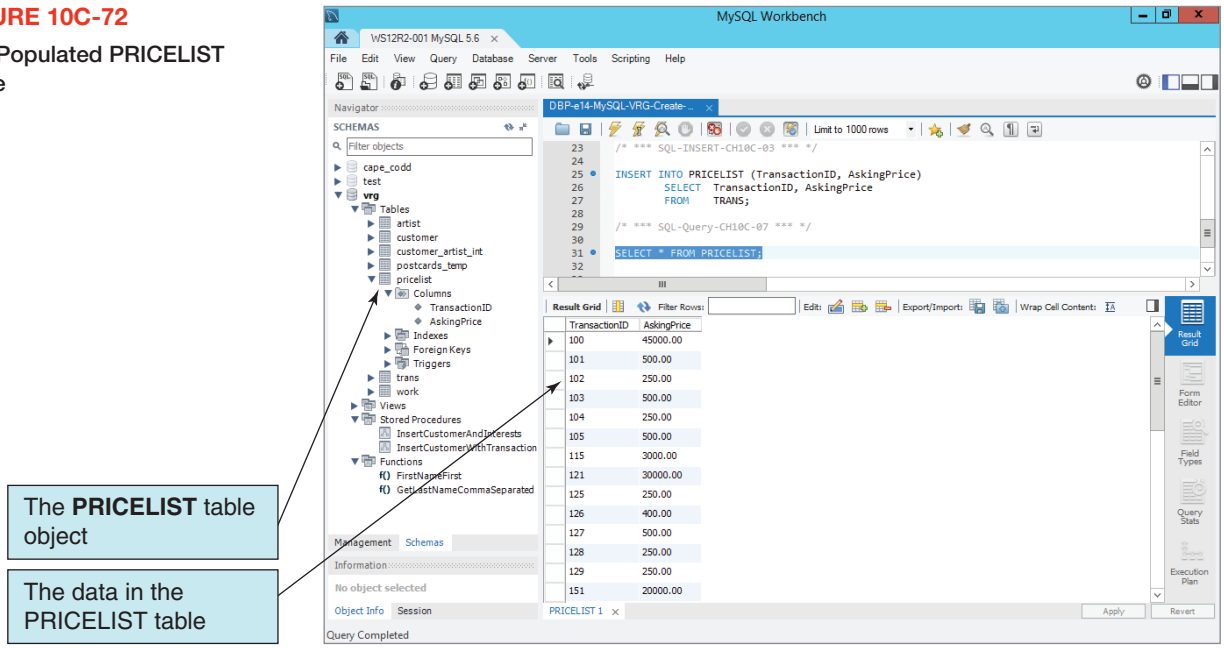
Finally, as shown in Figure 10C-72, we can see the data in the PRICELIST table by using an SQL query:

```
/* *** SQL-Query-CH10C-07 *** */
SELECT      *
FROM        PRICELIST;
```

The TRANS_AfterInsertSetAskingPrice trigger code shown in Figure 10C-73 implements this pricing policy. In the trigger, after the variables are declared, the new values of WorkID and

FIGURE 10C-72

The Populated PRICELIST Table



```
DELIMITER //

CREATE TRIGGER AfterTRANSInsertSetAskingPrice
AFTER INSERT ON TRANS
FOR EACH ROW

BEGIN

   DECLARE   varRowCount               Int;
   DECLARE   varPriorRowCount          Int;
   DECLARE   varWorkID                 Int;
   DECLARE   varTransactionID          Int;
   DECLARE   varAcquisitionPrice       Numeric(8,2);
   DECLARE   varNewAskingPrice         Numeric(8,2);
   DECLARE   varSumNetProfit           Numeric(8,2);
   DECLARE   varAvgNetProfit           Numeric(8,2);

   SET    varTransactionID = NEW.TransactionID;
   SET    varAcquisitionPrice = NEW.AcquisitionPrice;
   SET    varWorkID = NEW.WorkID;

   # First find if work has been here before.
   SELECT    COUNT(*) INTO varRowCount
   FROM      TRANS
   WHERE     WorkID = varWorkID;

   SET varPriorRowCount = (varRowCount - 1);

   # If varPriorRowCount = 0 this is a new acquistion.
   IF (varPriorRowCount = 0) THEN
      # Set varNewAskingPrice to twice the acquisition cost.
      SET varNewAskingPrice = (2 * varAcquisitionPrice);
   ELSE
      # The work has been here before
      # We have to determine the value of varNewAskingPrice
      SELECT    SUM(NetProfit) INTO varSumNetProfit
      FROM      ArtistWorkNetView AS AWNV
      WHERE     AWNV.WorkID = varWorkID
      GROUP BY  AWNV.WorkID;

      SET varAvgNetProfit = (varSumNetProfit / varPriorRowCount);

      # Now choose larger value for the new AskingPrice.
      IF ((varAcquisitionPrice + varAvgNetProfit)
            > (2 * varAcquisitionPrice)) THEN
         SET varNewAskingPrice = (varAcquisitionPrice + varAvgNetProfit);
      ELSE
         SET varNewAskingPrice = (2 * varAcquisitionPrice);
         END IF;
      END IF;

  # Update PRICELIST with the value of AskingPrice
   INSERT INTO PRICELIST VALUES (varTransactionID, 0);

   UPDATE PRICELIST
      SET    AskingPrice = varNewAskingPrice
      WHERE  TransactionID = varTransactionID;

END
//

DELIMITER ;
```

FIGURE 10C-73

The SQL Statements for the TRANS_AfterInsertSetAskingPrice Trigger

AcquisitionPrice are obtained using the **MySQL NEW keyword** to access the the new data stored in the new row created by the INSERT action (similarly, the **MySQL OLD keyword** allows us to access data stored in a row removed by a DELETE action). Then a SELECT COUNT(*) FROM TRANS is executed to count the number of rows with the given WorkID.

The variable varPriorRowCount is set to varRowCount minus one because this is an AFTER trigger, and the new row will already be in the database. The expression (varRowCount-1) is the correct number of qualifying TRANS rows that were in the database prior to the insert.

We test to see if the work is new. If it is new, varNewAskingPrice is then set to twice the AcquisitionPrice. If the work has been in the gallery before, we must calculate which values of varNewAskingPrice to use. Thus, if varPriorRowCount is greater than zero, there were TRANS rows for this work in the database, and the ArtistWorkNetView view (see pages 335-337 in Chapter 7) is used to obtain the sum of the WorkNetProfit for the work. The AVG built-in function cannot be used because it will compute the average using varRowCount rather than varPriorRowCount.

Next, the two possible values of varNewAskingPrice are compared, and varNewAskingPrice is set to the larger value. Finally, an insert and an update are made to PRICELIST to set AskingPrice to the computed value of varNewAskingPrice.

Unfortunately, MySQL does not allow us to send any output messages for triggers, so we cannot use a SELECT statement similar to the ones we used in our stored procedures to display any trigger results—we can only look at the tables (directly or by using a view) for confirmation that the trigger worked correctly.

We create the trigger by writing and executing a MySQL script. To test the trigger, we will begin by obtaining a new work for the View Ridge Gallery. Because Melinda Gliddens just bought the only copy of the print of John Singer Sargent's *Spanish Dancer*, let's replace it:

```
/* *** SQL-INSERT-CH10C-04 *** */
INSERT INTO WORK (Title, Copy, Medium, Description, ArtistID)
     VALUES
     ('Spanish Dancer', '635/750', 'High Quality Limited Print',
      'American Realist style - From work in Spain', 11);
# Obtain the new WorkID
/* *** SQL-Query-CH10C-08 *** */
SELECT     WorkID
FROM       WORK
WHERE      ArtistID = 11
     AND   Title = 'Spanish Dancer'
     AND   Copy = '635/750';
```

The result of SQL-Query-CH10C-08 gives us the WorkID of the new artwork, which in this case is 597:

	WorkID
▶	597
*	NULL

```
# Use the new WorkID value (597 in this case)
# This will fire the trigger TRANS_AfterInsertSetAskingPrice
/* *** SQL-INSERT-CH10C-05 *** */
INSERT INTO TRANS (DateAcquired, AcquisitionPrice, WorkID)
     VALUES ('2014-11-08', 200.00, 597);
```

```
# See the results in PRICELIST
/* *** SQL-Query-CH10C-09 *** */
SELECT      T.TransactionID, DateAcquired,
            AcquisitionPrice, WorkID,
            PL.AskingPrice AS PriceListAskingPrice
FROM        TRANS T, PRICELIST PL
WHERE       T.TransactionID = PL.TransactionID
      AND   T.TransactionID = 255;
```

Figure 10C-74 shows the results of the events triggered by the INSERT statement on TRANS. Note that the asking price for the new work (400.00) has been set to twice the acquisition cost (200.00), which is the correct value for a work that has not previously been in the gallery. This trigger provides useful functionality for the gallery. It saves the gallery personnel considerable manual work in implementing their pricing policy and likely improves the accuracy of the results as well.

A Trigger for Enforcing a Data Constraint

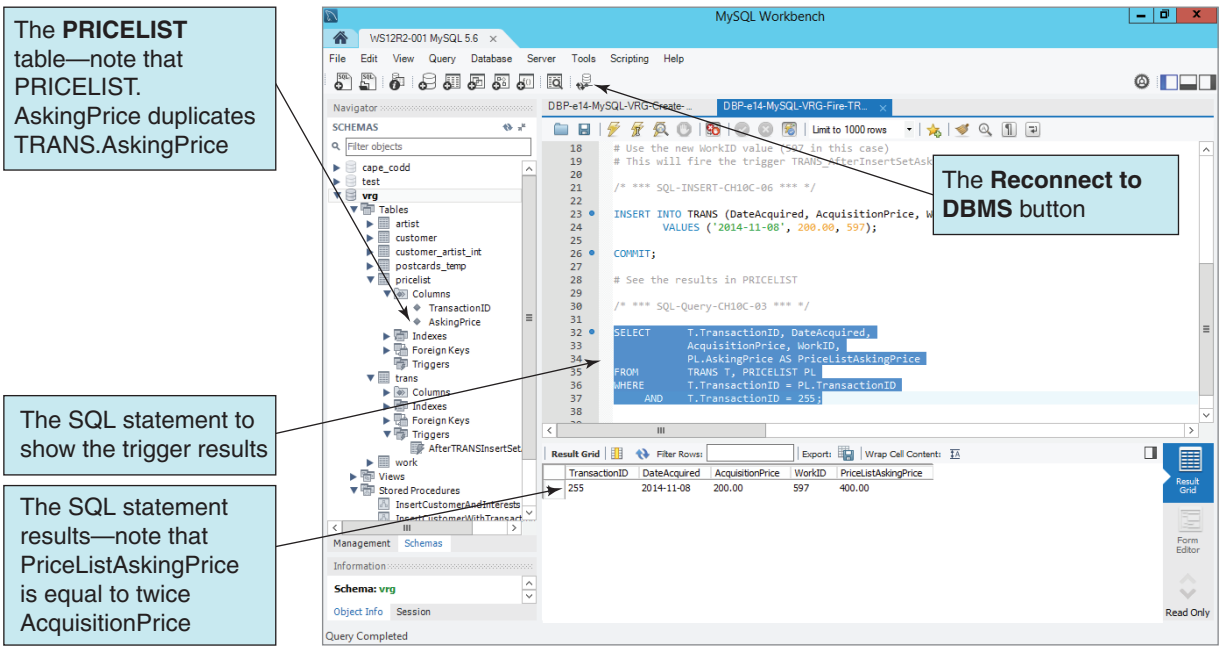
The View Ridge Gallery needs to track problem-customer accounts; these are customers either who have not paid promptly or who have presented other problems to the gallery. When a customer who is on the problem list attempts to make a purchase at the gallery, the gallery wants the transaction to be rolled back and a message displayed. Note that this feature requires an intertable CHECK constraint between the TRANS table and the CUSTOMER table.

Because this application logic uses a read against a table (CUSTOMER) other than the table firing the trigger (TRANS), we will not encounter the problem we discussed in the previous section. However, the application logic also calls for a transaction rollback if necessary and sending a message from the trigger. Neither of these is possible using MySQL triggers. MySQL stored procedures do allow these functions, and therefore we create a stored procedure to implement the necessary application logic.

To enforce this policy and the corresponding constraint, we need to add a column to the CUSTOMER table named isProblemAccount. This column will use the **MySQL BOOLEAN data type**, which can have the values NULL, 0, and 1. Zero will indicate a good account; 1 will indicate a problem account. And it looks like our new customer Melinda Gliddens had trouble with her previous payment, so we will set her isProblemAccount value to 1.

FIGURE 10C-74

Results for the TRANS_AfterInsertSetAskingPrice Trigger



But before we create the SQL statements to modify the CUSTOMER table, we need to discuss a default setting used in the MySQL Workbench that will affect what we are trying to do. As discussed earlier in this chapter, MySQL Workbench does not allow the execution of SQL UPDATE and DELETE statements unless statements have an SQL WHERE clause. As discussed earlier in this chapter, this setting is called *safe updates*. As we learned in our Chapter 7 discussion of SQL DML actions, an UPDATE statement without a WHERE clause will affect *every* row in the table, and a DELETE statement without a WHERE clause will remove *every* row in a table. Not a good idea!

To prevent this inadvertently happening, MySQL simply won't let you do it. However, when altering the structure of a table it can be necessary to UPDATE every row in the table. Therefore, before modifying the CUSTOMER table, we need to disable safe updates. Follow the steps discussed earlier and illustrated in Figures 10C-63 and 10C-64 to disable safe dates for the time being.

Now we are ready to modify the CUSTOMER table. The SQL statements to do this are:

```
/* *** SQL-ALTER-TABLE-CH10C-10 *** */
ALTER TABLE CUSTOMER
     ADD isProblemAccount   BOOLEAN NULL DEFAULT '0';
/* *** SQL-UPDATE-CH10C-03 *** */
UPDATE      CUSTOMER
     SET       isProblemAccount = 0;
/* *** SQL-UPDATE-CH10C-04 *** */
UPDATE      CUSTOMER
     SET       isProblemAccount = 1
     WHERE     LastName = 'Gliddens'
        AND    FirstName = 'Melinda';
/* *** SQL-Query-CH10C-10 *** */
SELECT      CustomerID, LastName, FirstName, isProblemAccount
FROM        CUSTOMER;
```

The results of the SELECT statement are:

CustomerID	LastName	FirstName	isProblemAccount
1000	Janes	Jeffrey	0
1001	Smith	David	0
1015	Twilight	Tiffany	0
1033	Smathers	Fred	0
1034	Frederickson	Mary Beth	0
1036	Warning	Selma	0
1037	Wu	Susan	0
1040	Gray	Donald	0
1041	Johnson	Lynda	0
1051	Wilkens	Chris	0
1052	Bench	Michael	0
1053	Gliddens	Melinda	1
NULL	NULL	NULL	NULL

At this point, we need to reset the safe updates setting back to its default–leaving it disabled is asking for trouble!

Now we will create a stored procedure named TransWithCheckIsProblemAccount. With this stored procedure, when a customer makes a purchase, the stored procedure determines whether the customer is flagged by the value of the isProblemAccount data in the CUSTOMER table. If so, the transaction is rolled back and a message is displayed. The stored procedure code in Figure10C-75 should enforce this policy.

We create the stored procedure by writing and executing a MySQL script. To test the trigger, we will have Melinda Gliddens attempt to buy another print from the View Ridge Gallery. Let's see what happens when we attempt to complete the transaction:

```
/* *** SQL-CALL-CH10C-03 *** */
CALL TransWithCheckIsProblemAccount (229, 1053, '2008-11-18', 475.00);
```

As shown in Figure 10C-76, the stored procedure successfully detected that there were problems with Melinda's account, stopped the transaction, and sent back a warning message.

FIGURE 10C-75

The SQL Statements for the TransWithCheckIsProblemAccount Stored Procedure

```
DELIMITER //

CREATE PROCEDURE TransWithCheckIsProblemAccount
          (IN    transTransactionID     Int,
           IN    transCustomerID        Int,
           IN    transDateSold          Date,
           IN    transSalesPrice        Numeric(8,2))

BEGIN

   DECLARE   varIsProblemAccount BOOLEAN;

   # Obtain value of varIsProblemAcocunt.

   SELECT    isProblemAccount INTO varIsProblemAccount
   FROM   CUSTOMER
   WHERE  CustomerID = transCustomerID;

   IF (varIsProblemAccount = 1) THEN
      # This is a problem account.
      # Rollback the transaction.
      ROLLBACK;
      # Print message
      SELECT 'Contact the manager immediately - Customer account problems.'
      AS TransWithCheckIsProblemAccountWarningMessage;
   ELSE
      # This is a good account.
      # Complete the transaction.
      # Update TRANS with the values of CustomerID, DateSold and SalesPrice
      UPDATE  TRANS
         SET    CustomerID = transCustomerID,
                DateSold = transDateSold,
                SalesPrice = transSalesPrice
         WHERE  TransactionID = transTransactionID;
      # Print message
      SELECT 'Transaction completed successfully.'
      AS TransWithCheckIsProblemAccountSuccessfullTransactionMessage;
   END IF;

END
//

DELIMITER ;
```

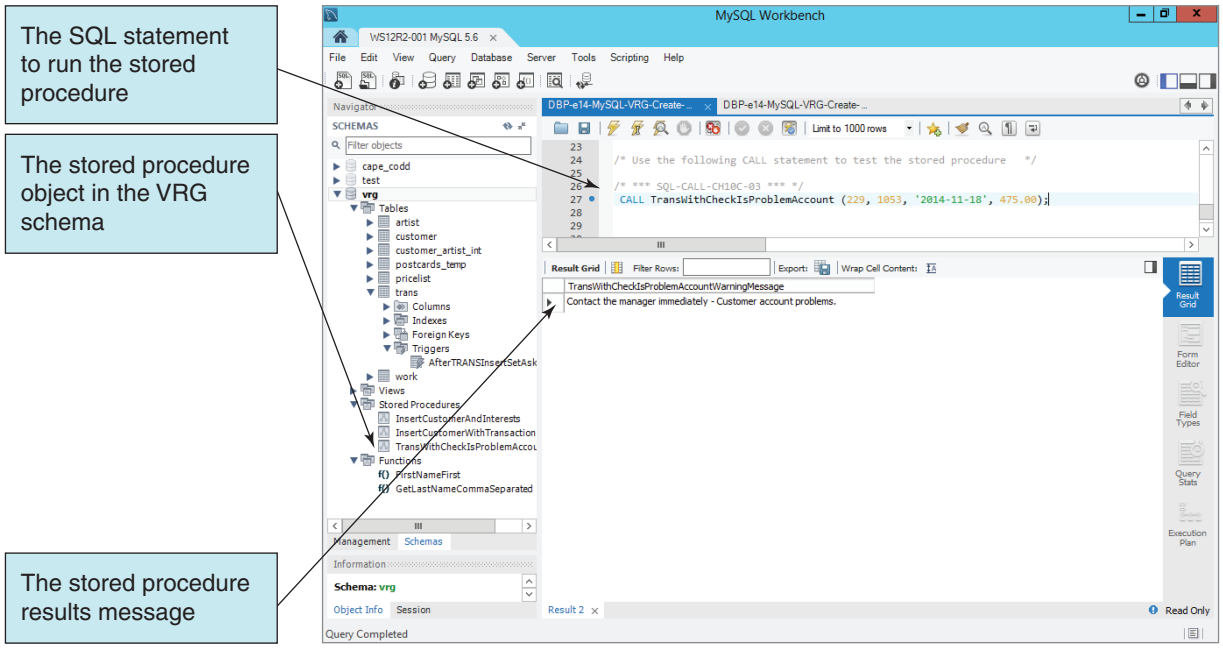


FIGURE 10C-76

Results for the TransWithCheckIsProblemAccount Stored Procedure

BY THE WAY

Using a table of valid or invalid values is more flexible and dynamic than placing such values in a CHECK constraint. For example, consider the CHECK constraint on Nationality values in the ARTIST table. If the gallery manager wants to expand the nationality of allowed artists, the manager will have to change the CHECK constraint using the ALTER TABLE statement. In reality, the gallery manager will have to hire a consultant to change this constraint.

A better approach is to place the allowed values of Nationality in a table, say ALLOWED_NATIONALITY. Then write a stored procedure like that shown in Figure 10C-73 to enforce the constraint that new values of Nationality exist in ALLOWED_NATIONALITY. When the gallery owner wants to change the allowed artists, the manager would simply add or remove values in the ALLOWED_NATIONALITY table.

A Trigger for Updating a View

In Chapter 7, we discussed the problem of updating views. One such problem concerns updating views created via joins; it is normally not possible for the DBMS to know how to update tables that underlie the join. However, sometimes application-specific knowledge can be used to determine how to interpret a request to update a joined view.

Consider the view CustomerInterestsView shown in Figure 10C-47. It contains rows of CUSTOMER and ARTIST joined over their intersection table. CUSTOMER.LastName is given the alias CustomerLastName, CUSTOMER.FirstLastName is given the alias CustomerFirstName, and ARTIST.LastName is given the alias ArtistName.

A request to change the last name of a customer in CustomerInterests can be interpreted as a request to change the last name of the underlying CUSTOMER table. Such a request, however, can be processed only if the value of (CUSTOMER.LastName, CUSTOMER.FirstName) is unique. If not, the request cannot be processed.

However, such a trigger requires an INSTEAD OF trigger, and MySQL does not support INSTEAD OF triggers. Thus, in MySQL updates will have to be written to the tables themselves in application logic, probably using a stored procedure, not to the CustomerInterestsView.

For example, suppose View Ridge Gallery's two newest customers Michael Bench and Melinda Gliddens just got married after meeting at a View Ridge Gallery opening, and Melinda wants us to change her last name. The UPDATE command will have to be issued

against the CUSTOMER table, not the CustomerInterestsView. The UpdateCustomerName stored procedure shown in Figure 10C-77 takes the customer's old and new names as input, checks to make sure the customer exists in the database, and then updates the name. Note that this will work only if the customer's name is unique in the database. If it is not, we will need the CustomerID number to make the change

BY THE WAY Although we can't see anything in the Stored Procedure code that would cause problems, the stored procedure would not run with the CALL statement we're about to run without disabling safe updates as described earlier in this chapter. Neither would the SQL-UPDATE-CH10C-05 and SQL-UPDATE-CH10C-06 in our final database update work below.

This may be a bug in MySQL Workbench. In any event, disable safe updates at this point so you can run the remaining work in this chapter.

FIGURE 10C-77

The SQL Statements for the UpdateCustomerName Stored Procedure

```
DELIMITER //

CREATE PROCEDURE UpdateCustomerName
          (IN    oldCustomerLastName     Char(25),
           IN    oldCustomerFirstName    Char(25),
           IN    newCustomerLastName     Char(25),
           IN    newCustomerFirstName    Char(25))

BEGIN

   DECLARE   varRowCount    Int;

   # Count number of synonyms in CUSTOMER.
   SELECT    COUNT(*) INTO varRowCount
   FROM      CUSTOMER AS C1
   WHERE     C1.LastName = oldCustomerLastName
      AND    C1.FirstName = oldCustomerFirstName
      AND    EXISTS
             (SELECT       *
              FROM         CUSTOMER AS C2
              WHERE        C1.LastName = C2.LastName
                    AND    C1.FirstName = C2.FirstName
                    AND    C1.CustomerID <> C2.CustomerID);

   IF (varRowCount >= 1) THEN
      # This is a problem account.
      # Rollback the transaction.
      ROLLBACK;
      # Print message
      SELECT 'Duplicate customer name - Contact the manager.'
      AS UpdateCustomerNameWarningMessage;
   ELSE
      # The Customer last name is unique.
      # Update the Customer record.
      UPDATE CUSTOMER
        SET         LastName = newCustomerLastName,
                    FirstName = newCustomerFirstName
        WHERE       LastName = oldCustomerLastName
              AND   FirstName = oldCustomerFirstName;
      # Print message
      SELECT 'Customer name updated successfully.'
      AS UpdateCustomerNameSuccessfulMessage;
      END IF;

END
//

DELIMITER ;
```

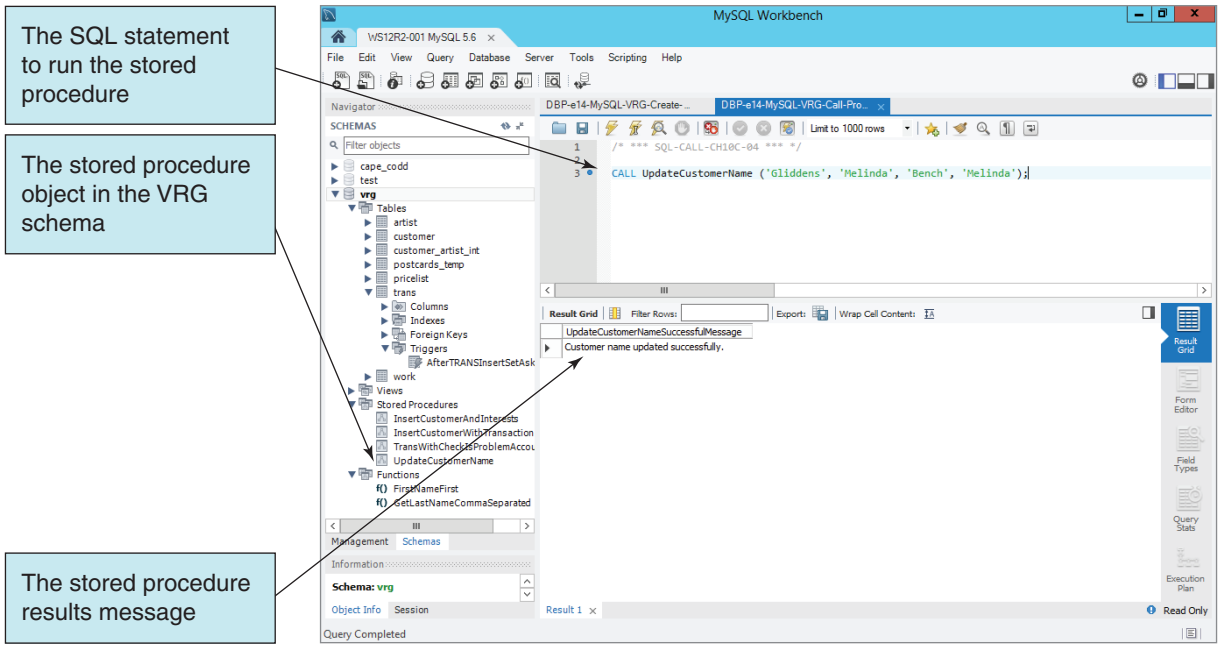


FIGURE 10C-78
Results for the UpdateCustomerName Stored Procedure

We will use the following SQL statement to call the stored procedure:

```
/* *** SQL-CALL-CH10C-04 *** */
CALL UpdateCustomerName ('Gliddens', 'Melinda', 'Bench', 'Melinda');
```

Figure 10C-78 shows the message indicating that the name change was successfully completed, and Figure 10C-79 shows Melinda's updated name in the CUSTOMER table. Melinda is now Melinda Bench.

A Trigger for Enforcing a Required Child Constraint

The VRG database design includes an M-M relationship between WORK and TRANS. Every WORK must have a TRANS to store the price of the work and the date the work was acquired, and every TRANS must relate to a WORK parent. Figure 10C-80 shows the tasks that must be accomplished to enforce this constraint; it is based on the boilerplate shown in Figure 6-29(b).

FIGURE 10C-79
The Updated Name in the CUSTOMER Table

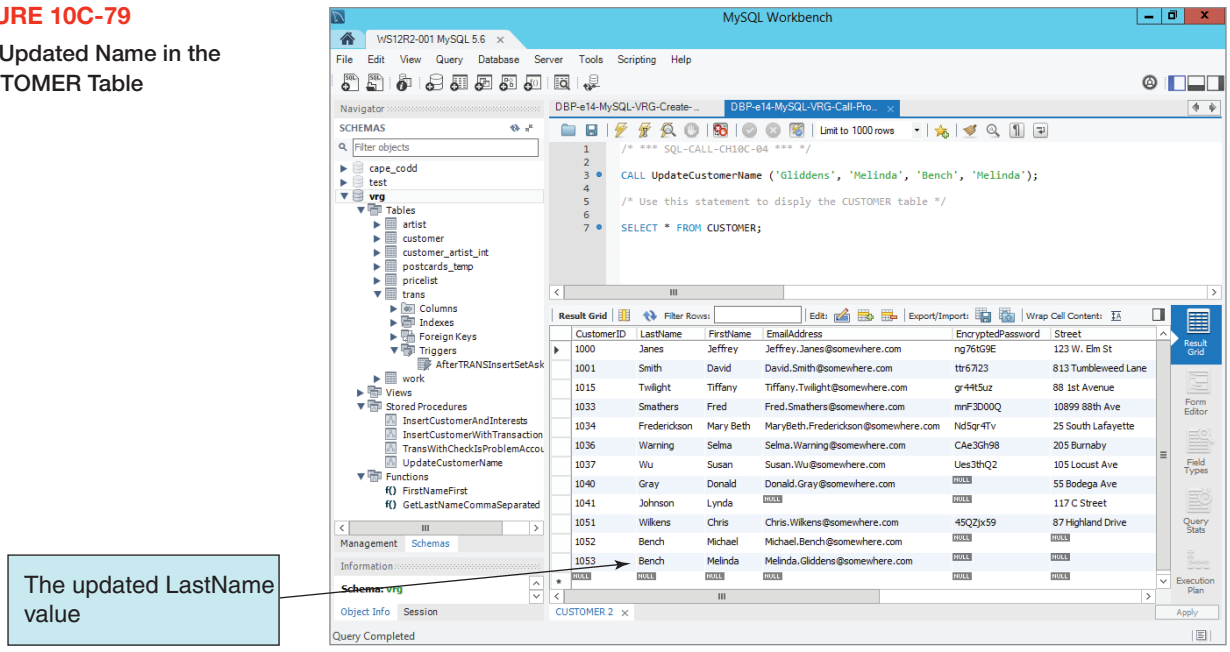


FIGURE 10C-80

Actions to Enforce Minimum Cardinality for the WORK-to-TRANS Relationship

WORK Is Required Parent TRANS Is Required Child	Action on WORK (Parent)	Action on TRANS (Child)
Insert	Create a TRANS row	New TRANS must have a valid WorkID (enforced by DBMS)
Modify key or foreign key	Prohibit—WORK uses a surrogate key	Prohibit—TRANS uses a surrogate key, and TRANS cannot change to a different WORK
Delete	Prohibit—Cannot delete a WORK with TRANS children (enforced by DBMS by lack of CASCADE DELETE)	Cannot delete the last child [Actually, data related to a transaction is never deleted (business rule)]

Because the CREATE TABLE statement for TRANS in Figure 10C-31 defines TRANS.WorkID as NOT NULL and the FOREIGN KEY constraint without cascading deletions, the DBMS will ensure that every TRANS has a WORK parent. So, we need not be concerned with enforcing the insert on TRANS or the deletion on WORK. As stated in Figure 10C-80, the DBMS will do that for us. Also, we need not be concerned with updates to WORK.WorkID because it is a surrogate key.

Three constraints remain that must be enforced by triggers or stored procedures: (1) ensuring that a TRANS row is created when a new WORK is created; (2) ensuring that TRANS.WorkID never changes; and (3) ensuring that the last TRANS child for a WORK is never deleted.

We can enforce the second constraint by writing a stored procedure for the update of TRANS that checks for a change in WorkID. If there is such a change, the stored procedure can roll back the change.

Concerning the third constraint, the gallery has a business policy that no TRANS data ever be deleted. Thus, we not only need to disallow the deletion of the last child, we also need to disallow the deletion of any child. We can do this by writing a stored procedure for deletions from TRANS that rolls back any attempted deletion.

However, the first constraint is a problem. We could write a trigger on the WORK INSERT to create a default TRANS row, but this trigger will be called before the application has a chance to create the TRANS row itself. The trigger would create a TRANS row and then the application may create a second one. To guard against the duplicate, we could then write a trigger on TRANS to remove the row the WORK trigger created in those cases when the application creates its own trigger. However, this solution is awkward at best.

A better design would be to require the applications to create the WORK and TRANS combination via a view. For example, consider the view WorkAndTransView:

```
/* *** SQL-CREATE-VIEW-CH10C-01 *** */
CREATE VIEW WorkAndTransView AS
     SELECT     Title, Copy, Medium, Description, ArtistID,
                DateAcquired, AcquisitionPrice
     FROM       WORK AS W JOIN TRANS AS T
          ON    W.WorkID = T.WorkID;
```

Unfortunately, the DBMS will not be able to process an insert on this view, and MySQL does not have an INSTEAD OF trigger to process the insert. We will have to resort to other application logic, and you are asked to design a stored procedure for this process in Project Question 10C.36.I.

As we end our discussion, we should note that Melinda, now Mrs. Michael Bench, has worked out her account problems with the View Ridge Gallery and has completed her purchase of the print of Horiuchi's *Color Floating in Time*.

```
/* *** SQL-UPDATE-CH10C-05 *** */
UPDATE      CUSTOMER
       SET        isProblemAccount = 0
       WHERE      LastName = 'Bench'
            AND   FirstName = 'Melinda';
/* *** SQL-UPDATE-CH10C-06 *** */
UPDATE      TRANS
       SET        DateSold = '2014-11-18',
                  SalesPrice = 475.00,
                  CustomerID = 1053
       WHERE      TransactionID = 229;
```

Therefore, we will restock a copy of this print into the gallery, but we cannot do this with a trigger on a view—note the work we have to do instead. The stored procedure you will create in Project Question 10C.36.H will be written to perform the equivalent of these steps.

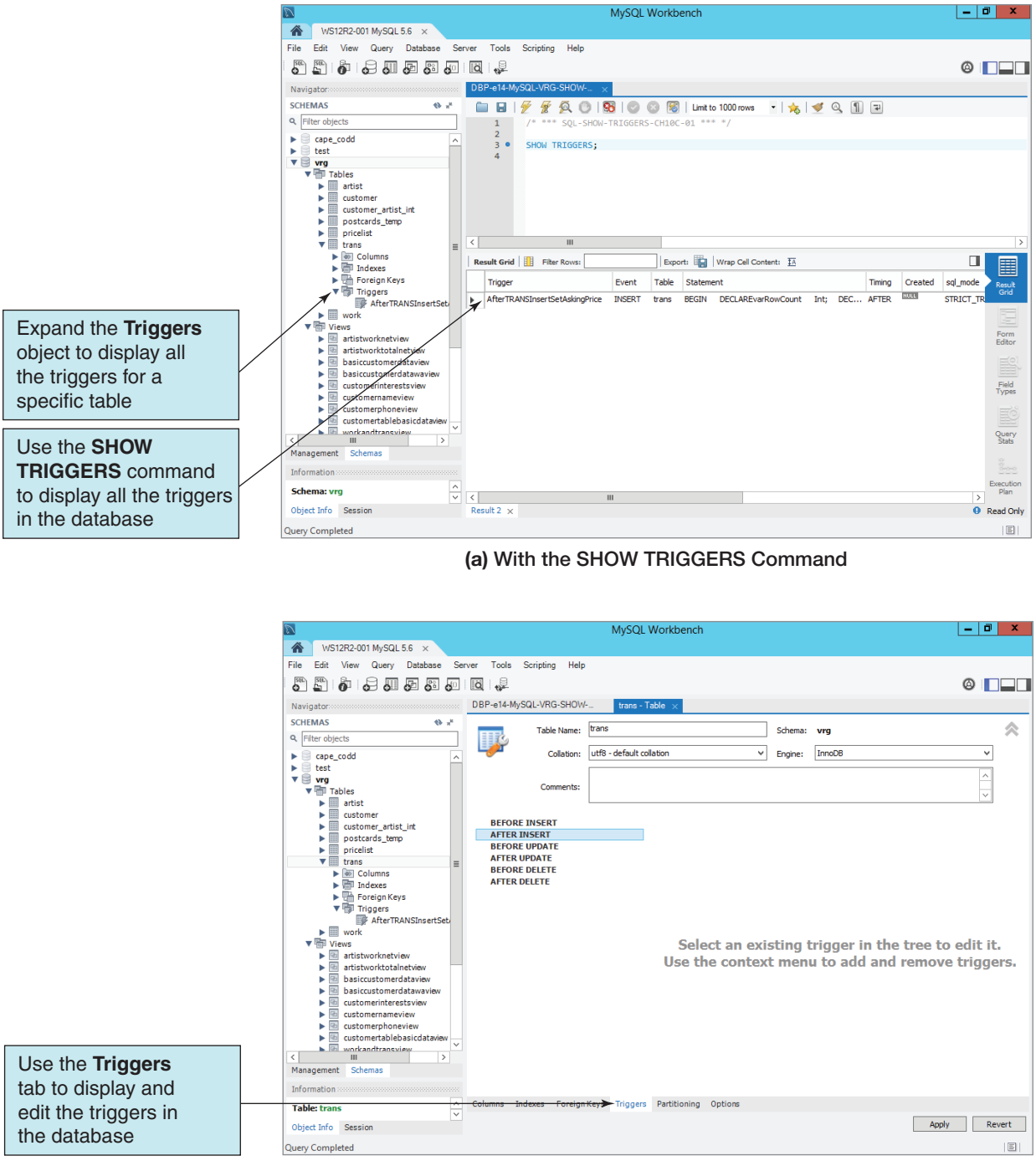
```
/* *** SQL-INSERT-CH10C-06 *** */
INSERT INTO WORK (Title, Copy, Medium, Description, ArtistID)
     VALUES(
     'Color Floating in Time', '493/500',
     'High Quality Limited Print',
     'Northwest School Abstract Expressionist style', 18);
# Obtain the new WorkID
/* *** SQL-Query-CH10C-11 *** */
SELECT      WorkID
FROM        WORK
WHERE       ArtistID = 18
      AND   Title = 'Color Floating in Time '
      AND   Copy = '493/500';
# Use the new WorkID value (598 in this case)
/* *** SQL-INSERT-CH10C-07 *** */
INSERT INTO TRANS (DateAcquired, AcquisitionPrice, WorkID)
      VALUES ('2015-02-05', 250.00, 598);
```

A Last Word on MySQL Stored Procedures and Triggers

Out of four possible uses for triggers, MySQL was able to support only one of them (and even that one required a work-around). MySQL triggers are useful in many circumstances, but they are not as useful for our current tasks as those in Microsoft SQL Server 2014 and Oracle Database.

We should also note that MySQL Workbench support for triggers is not as complete as that for views and stored procedures. View objects and stored procedure objects are displayed in the MySQL Workbench Object Browser, but there are no trigger objects displayed.

In that case, how do we see what triggers are in the database, and can we edit them after we have created them? As shown in Figure 10C-81(a), we can use the **SQL SHOW TRIGGERS statement** to display a list of triggers. Unfortunately, we cannot edit the triggers from the output of the SQL SHOW TRIGGERS statement. of the SQL SHOW



(a) With the SHOW TRIGGERS Command

(b) With the Triggers Tab in the MySQL Table Editor

FIGURE 10C-81

Displaying MySQL Triggers

TRIGGERS statement, which is another reason for using SQL scripts to save copies of our work. We can always drop the trigger using the **SQL DROP TRIGGER statement**, edit the trigger SQL script, and re-create the trigger. However, as shown in Figure 10C-81(b), the MySQL Table Editor *does* have a Triggers tab. Triggers that are created in this tabbed windows are available for editing via this tab, but, unfortunately, in the current version of MySQL Workbench triggers that are not created in this window are not available for editing. The triggers created in this tabbed window are stored directly in the database by this method, so it is still worth saving these triggers to an SQL script as backup.

FIGURE 10C-82
Concurrency Options for MySQL 5.6

Type	Scope	Options
Transaction isolation level	GLOBAL SESSION TRANSACTION	READ UNCOMMITTED READ COMMITTED REPEATABLE READ SERIALIZABLE

Concurrency Control

MySQL provides capabilities to control concurrent processing by using the transaction isolation level. Figure 10C-82 summarizes the concurrency control options. In this section, we will describe just the basics. For more information, see the MySQL 5.6 documentation on "SET TRANSACTION Syntax" at *http://dev.mysql.com/doc/refman/5.6/en/set-transaction.html*.

With MySQL, developers do not place explicit locks. Instead, developers declare the concurrency control behavior they want, and MySQL determines where to place the locks. Lock types include:

- **Record locks.** A lock on the index record.
- **Gap locks.** Generally, a lock on the unused index numbers between index records (see documentation).
- **Next-Key lock.** A combined record lock on the index record together with a gap lock on the unused index numbers before the index record itself.

For more information, see the MySQL 5.6 documentation "The InnoDB Transaction Model and Locking" at *http://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-model.html*.

As shown in Figure 10C-82, there are four **transaction isolation levels** listed in ascending level of restriction. Four of these options are the four you studied in Chapter 9, and they are options specified in the SQL standards. With MySQL InnoDB tables, REPEATABLE READ is the default isolation level. For READ COMMITTED, locking depends on the type of SQL statement being processed, and it is possible to allow dirty reads by setting the isolation level to READ UNCOMMITTED.

An SQL statement to set the isolation level can be issued anyplace SQL statements are allowed and can be set as GLOBAL (with appropriate permissions) or SESSION levels. An example SQL statement to set the isolation level of, say, REPEATABLE READ for the current session is:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

MySQL 5.6 Security

We discussed security in general terms in Chapter 9. Here we will summarize how those general ideas pertain to MySQL. The MySQL system consists of users who are granted permissions on various levels: global, database, table, column, or routine. MySQL does not have groups or roles. The MySQL 5.6 security model is shown in Figure 10C-83.

As usual, security consists of authentication and authorization. First, the user logs onto the DBMS, and then the user can do only what he or she has been granted permission to do. We create users and grant permissions with SQL statements and with the MySQL Workbench. We do this in the same WS12R2-001 MySQL 5.6 tabbed window we have been using—we simply need to expand the Navigator setting to display the MANAGEMENT, INSTANCE and PERFORMANCE tabbed window links that we contracted at the start of this chapter page as shown in Figure 10C-10(b). Click the Navigator objects expand/contract button to display the MANAGEMENT, INSTANCE, and PERFORMANCE tabbed window links as shown in Figure 10C-84.

Click the Server Status tabbed window link to display the Administration - Server Status tabbed window, as shown in Figure 10C-85, showing MySQL 5.6.configuration and performance data. After viewing this page, click the window **X [Close]** button to close the window.

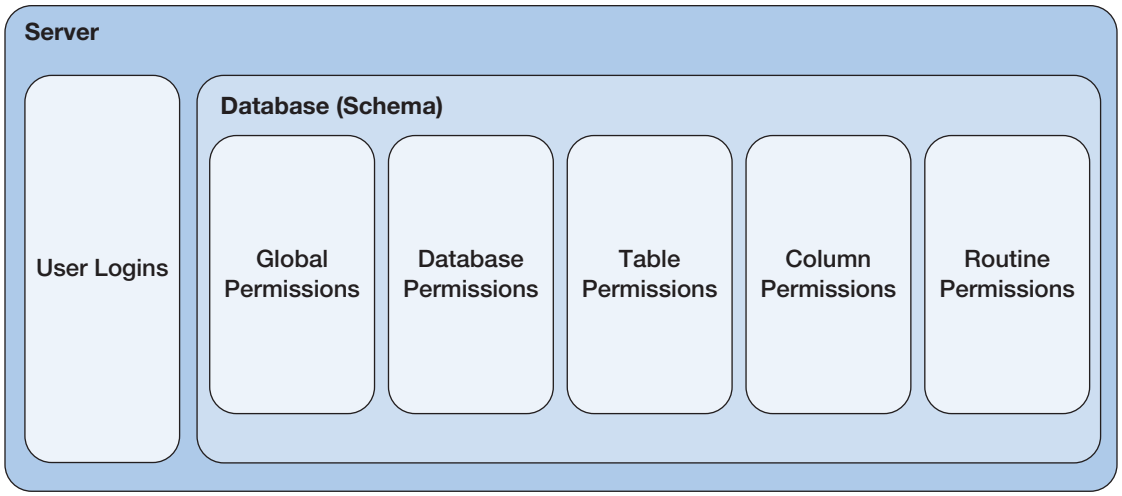


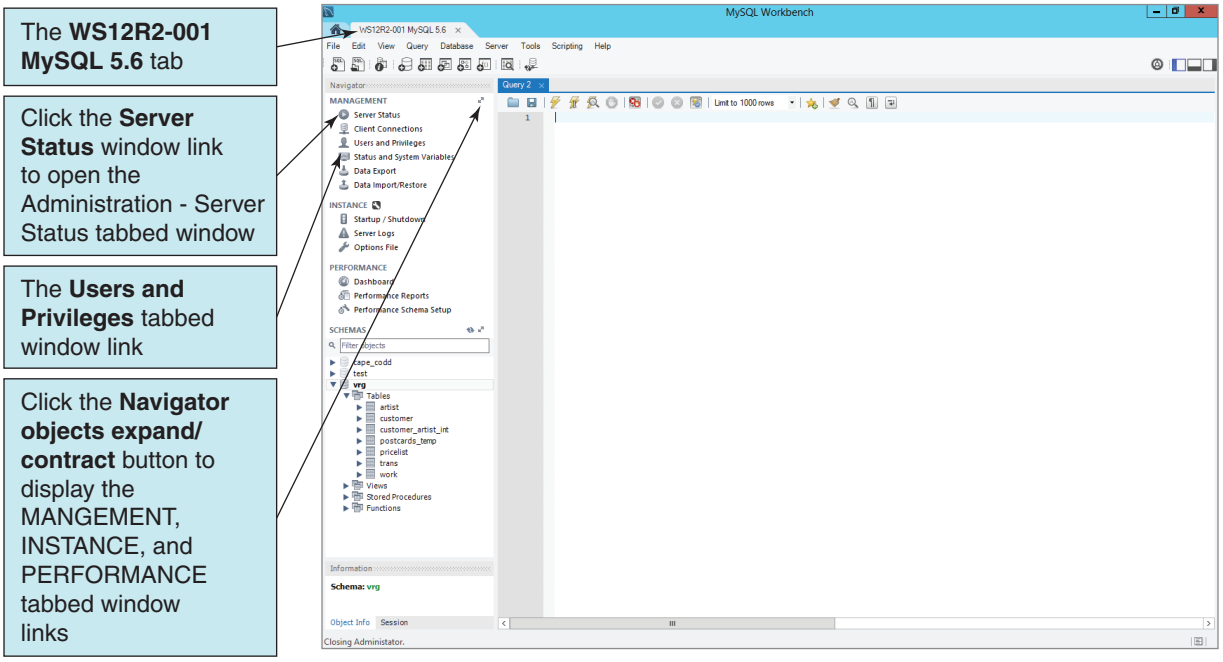
FIGURE 10C-83
MySQL 5.6 Security Model

Now, let's create a login for use by the View Ridge Gallery.

Creating a New User

1. Click the **Users and Privileges** tabbed window link to display the **Administration - Users and Privileges window**, as shown in Figure 10C-86.
2. Click the **Add Account** button. A new user with default login settings is created, as shown in Figure 10C-87.
3. The new login data for VRG-User is shown in 10C-88. Use that figure for reference in the following steps.
4. In the Login Name text box, type in the login name **VRG-User**.
5. In the Limit Connectivity to Hosts Matching: text box, type **localhost**. The User Information tab working area now appears, as shown in Figure 10C-88.
6. In the Password text box, type the password **VRG-User+password**.
7. In the Confirm Password text box, type the password **VRG-User+password**. The User Information tab working area now appears, as shown in Figure 10C-88.
8. Click the **Apply** button. The new user is created, as shown in Figure 10C-89.

FIGURE 10C-84
The MySQL Workbench MANGEMENT, INSTANCE and PERFORMANCE Page Links



The **WS12R2-001 MySQL 5.6** tab

The **Users and Privileges** tabbed window link

The **Administration – Server Status** window shows server status and performance data

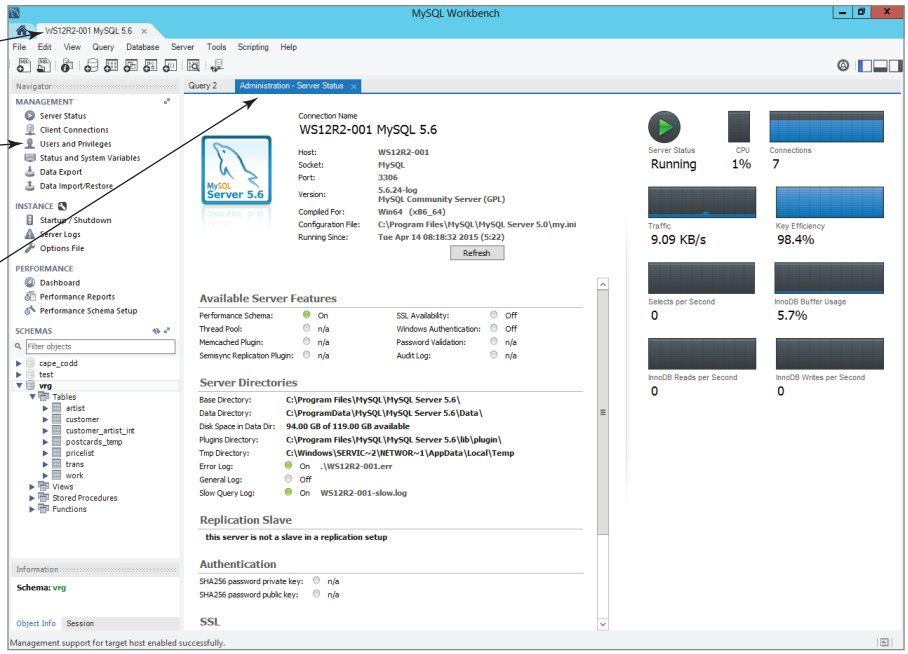


FIGURE 10C-85

The Administration—Server Status Window

MySQL Database Security Settings

Now we need to move to the database level of security. Here we grant users specific permissions to specific databases, and now we need to grant specific permission to our new user. We could use the **SQL GRANT statement** to do this, and a full discussion of how to do this is in the MySQL document "GRANT Syntax" at *http://dev.mysql.com/doc/refman/5.6/en/grant.html*. However, we will do this in the MySQL Workbench. There are two types of privileges we can grant to a user:

- **MySQL administrative roles**, which assign predefined sets of privileges to the DBMS itself and thus to every database on that server.
- **MySQL schema privileges**, which assign specific rights to a specific database.

We can, of course, assign a user a combination of server roles and schema privileges. First, we'll consider server administrative roles.

FIGURE 10C-86

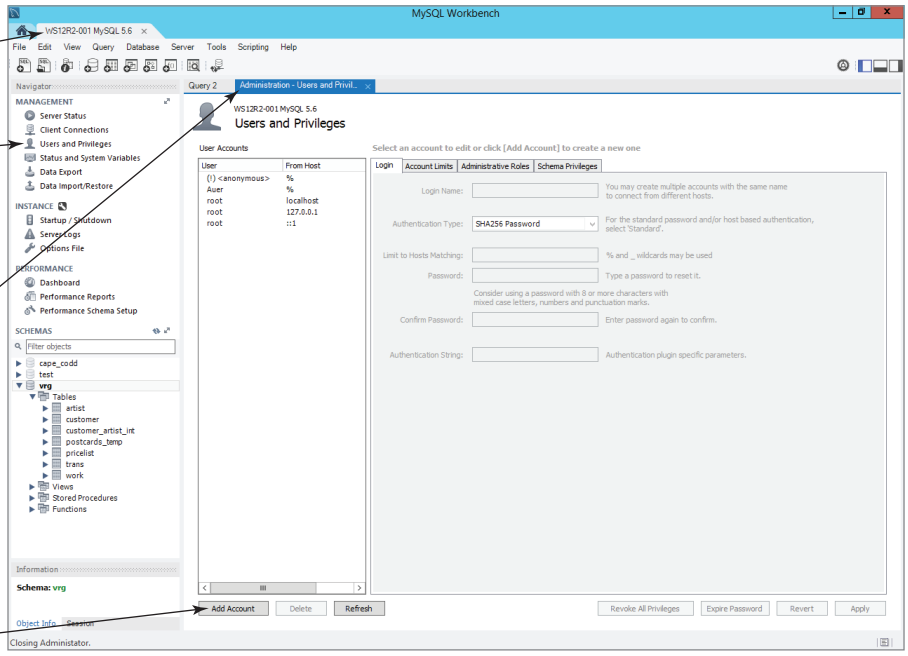
The Administration—User and Privileges Window

The **WS12R2-001 MySQL 5.6** tab

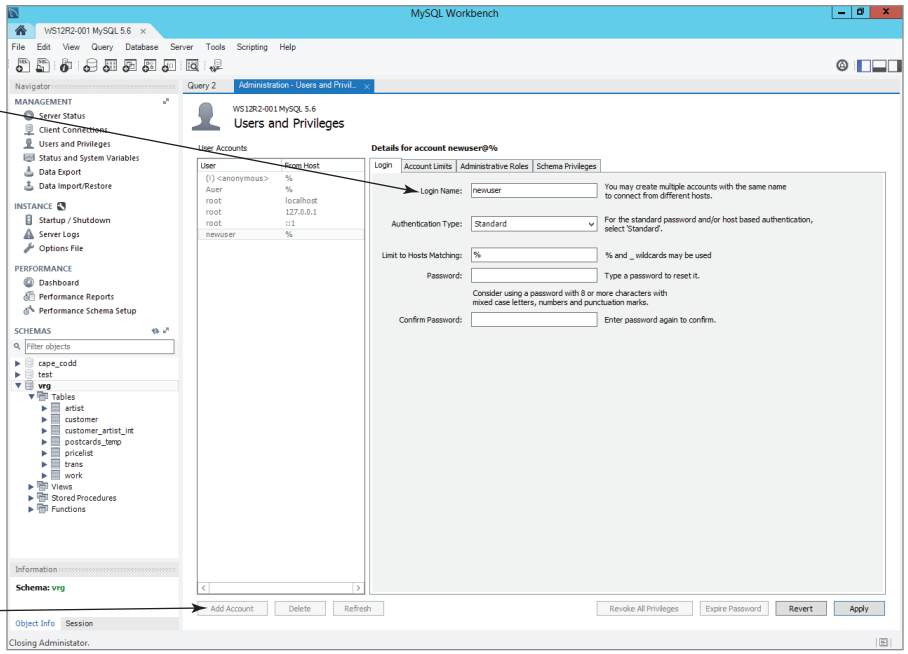
The **Users and Privileges** tabbed window link

The **Administration – User and Privileges** window is used to create and manage users, and to assign schema privileges to users

The **Add account** button



The default new user Login data—enter the Login Name, Password (and confirm it), and the host name of the MySQL instance for this login



The **Add account** button

FIGURE 10C-87

Default User Data

Granting MySQL Administrative Roles

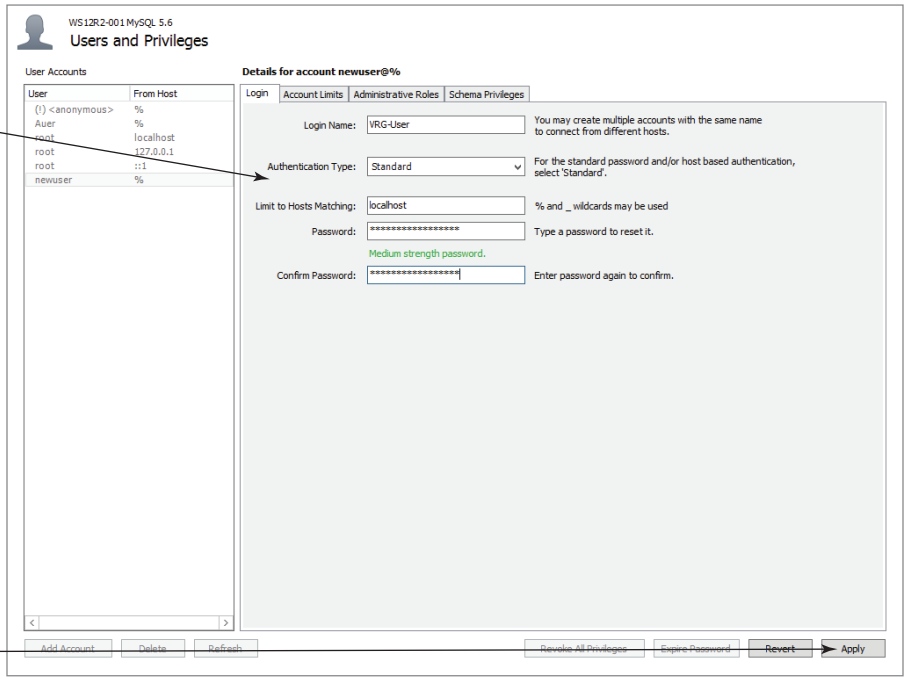
1. In **User Account,** click the **VRG-User** user object to select it.
2. Click the **Administrative Roles** tab to display the server administrative roles available to be assigned to VRG-User, as shown in Figure 10C-90.
3. At this point, we will decide that we do not want to grant VRG-User a server administrative role but would rather grant permissions only to the VRG database. Therefore, having considered these roles, we will not assign any of them to VRG-User.

Having decided that VRG-User needs only VRG database privileges, we will assign those to VRG-User.

FIGURE 10C-88

Creating the VRG-User Login

The new user Login data for **VRG-User**—*localhost* means the computer MySQL is running on, and is the host name of the MySQL instance for this login



The **Apply** button

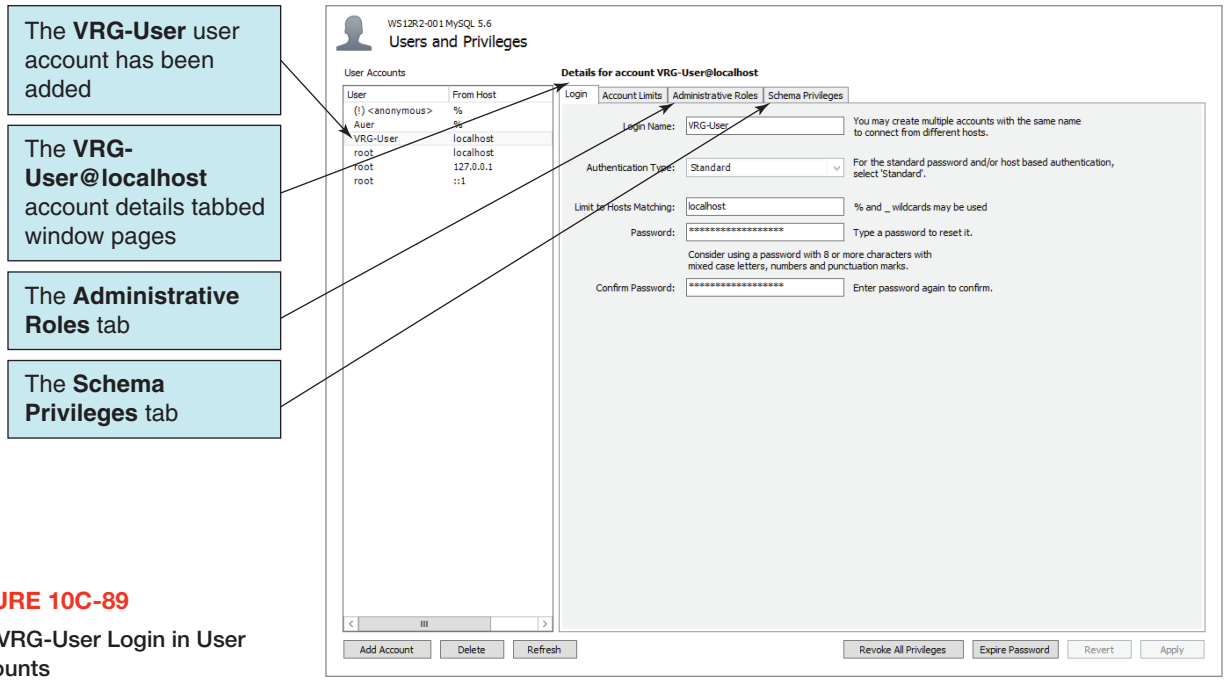


FIGURE 10C-89

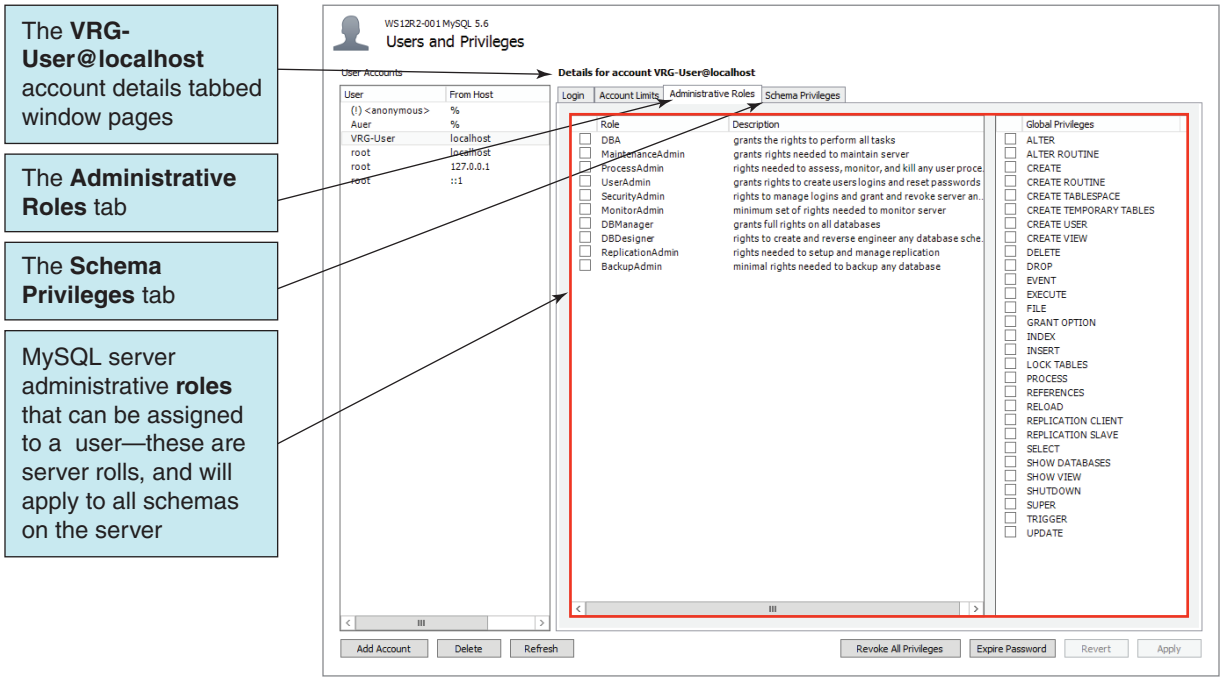
The VRG-User Login in User Accounts

Granting MySQL Database Privileges

1. The VRG-User account should still be selected. If not, click the **VRG-User** user object to select it.
2. Click the **Schema Privileges** tab to display the schema privileges for VRG-User. As shown in Figure 10C-91, VRG-User currently has no database-specific privileges.
3. Click the **Add Entry** button to display the New Schema Privilege Definition dialog box, as shown in Figure 10C-92.
4. In the New Schema Privilege Definition dialog box, click the **Selected schema** radio button, and then select **vrg** from the schema list.
5. In the New Schema Privilege Definition dialog box, click the **OK** button. The selected host and schema are assigned to VRG-User but without any permissions, as shown in Figure 10C-93.

FIGURE 10C-90

Assigning Administrative Roles



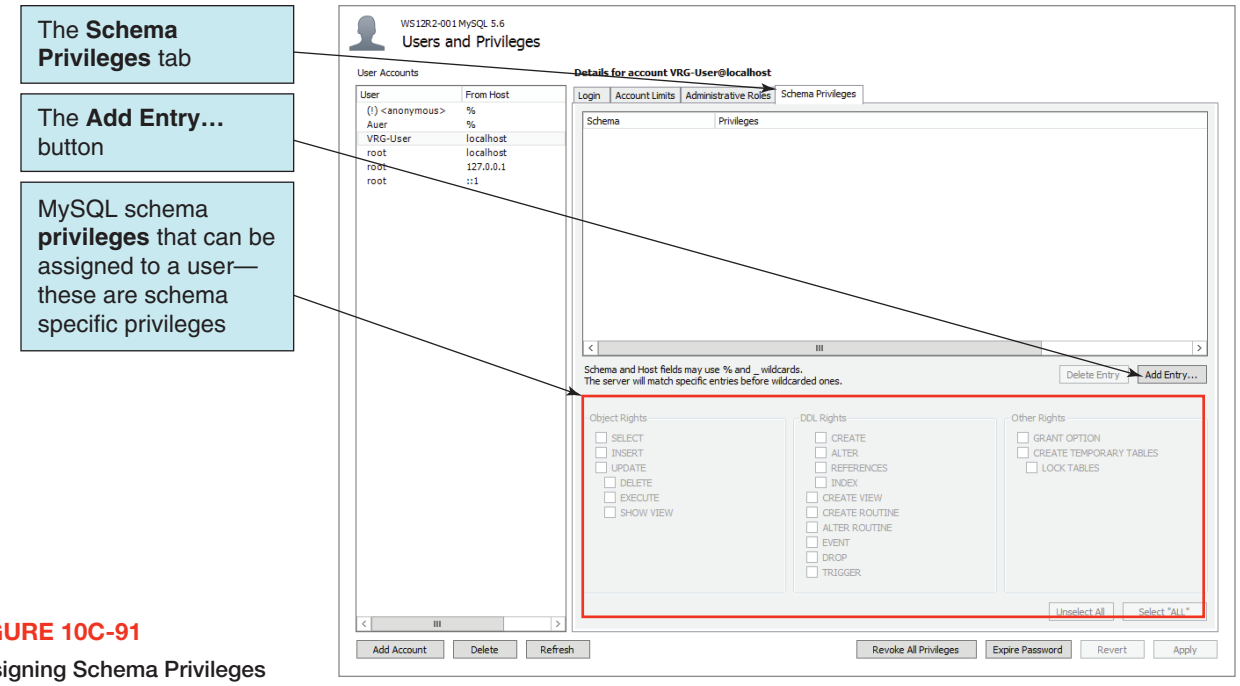
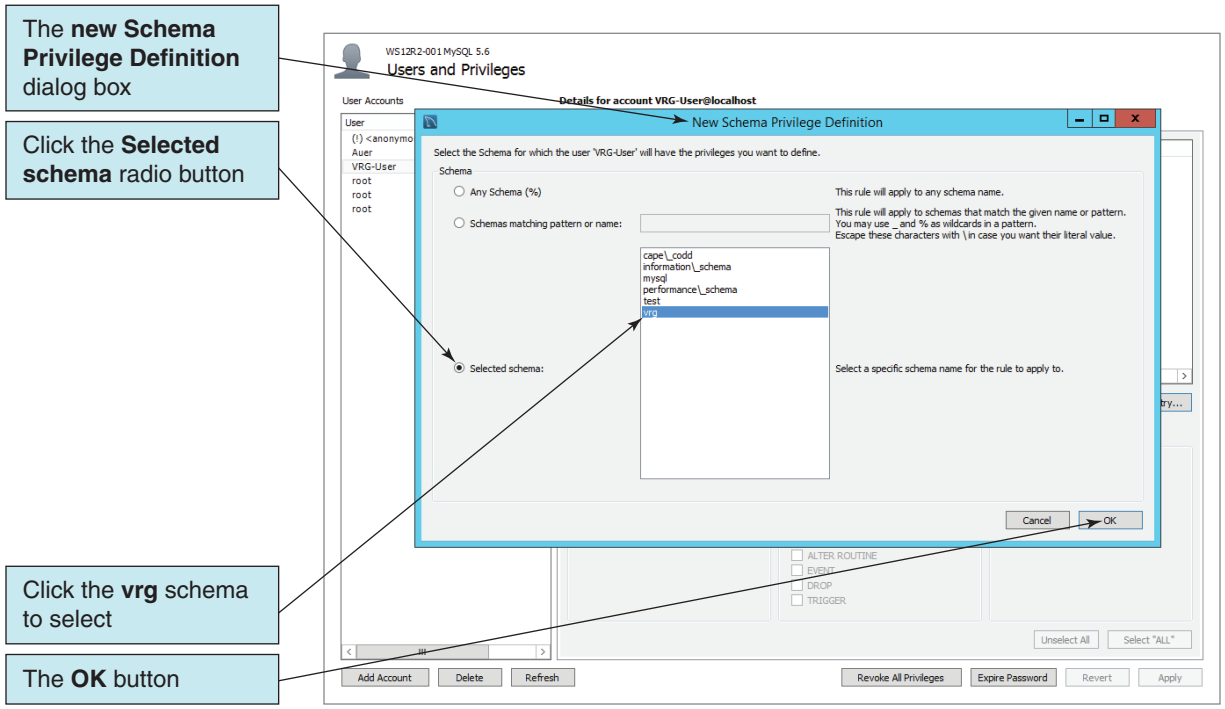


FIGURE 10C-91
Assigning Schema Privileges

6. Click the **Select "All"** button. All privileges except the GRANT OPTION are selected, as shown in Figure 10C-94. The GRANT OPTION would allow the user to give any assigned permission to another user, and this action should be reserved for database administrators.
7. Click the **Apply** button. The privileges are now granted, as shown in Figure 10C-95.

Now we have created the login and assigned permissions for VRG-User for the VRG database. We will use these permissions in Chapter 11 when we develop a Web database application based on this database.

FIGURE 10C-92
Selecting the Schema



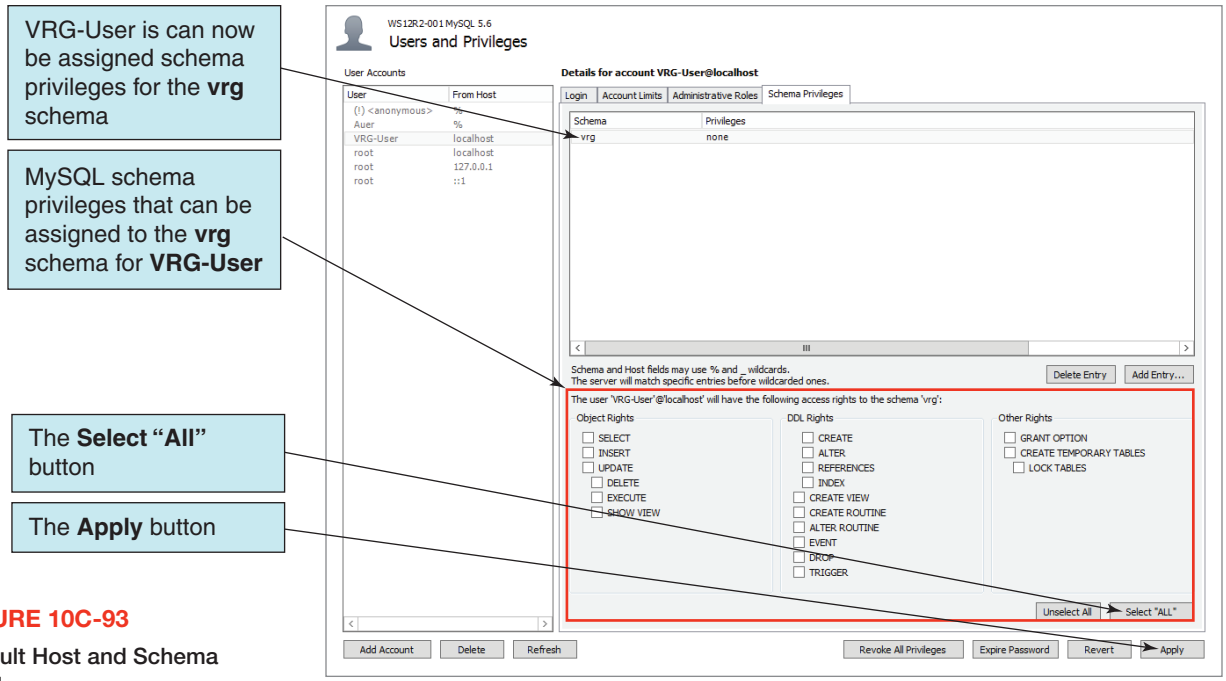


FIGURE 10C-93

Default Host and Schema Privileges

MySQL 5.6 DBMS Backup and Recovery

As explained in Chapter 9, database files should be backed up periodically. When this is done, it is possible to recover a failed database by restoring it from a prior saved database and applying changes from a log file. MySQL includes command-line utilities for making detailed backups and a backup facility in the MySQL Workbench. Here we will use the MySQL Workbench backup tools. For more information on the MySQL administration utilities, see the MySQL 5.6 Manual section "MySQL Administrative and Utility Programs" at *http://dev.mysql.com/doc/refman/5.6/en/programs-admin-utils.html*.

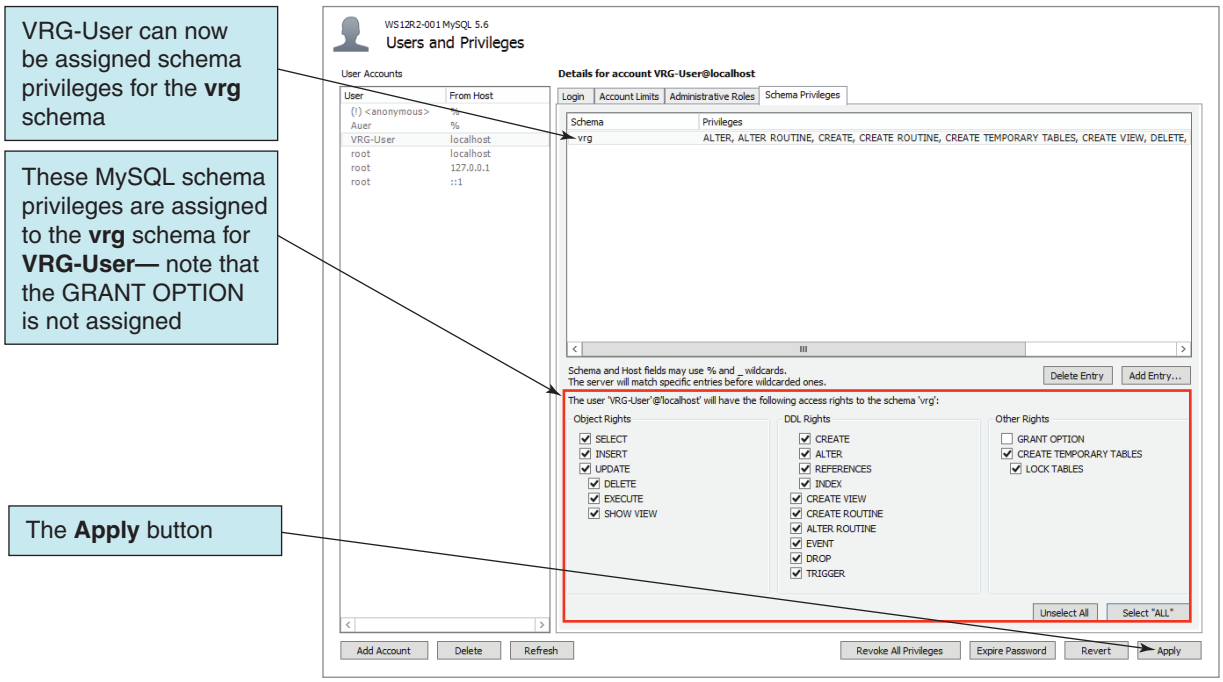


FIGURE 10C-94

Selected Host and Schema Privileges

These MySQL schema privileges are now permanently assigned to the **vrg** schema for **VRG-User**

To remove schema privileges, use the **Revoke All Privileges** button

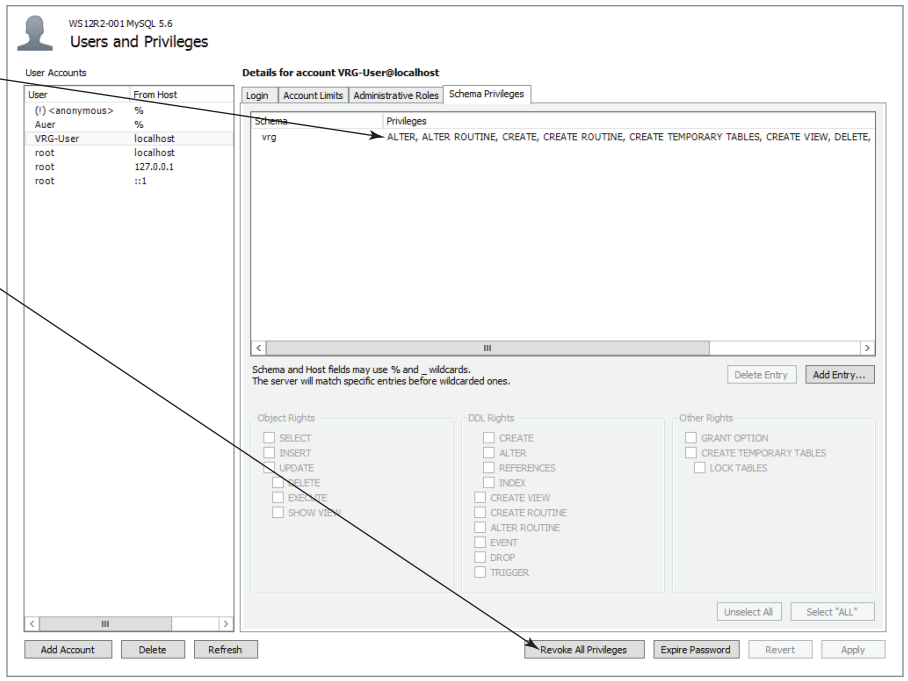


FIGURE 10C-95

Assigned Host and Schema Privileges

Note that the MySQL Enterprise Edition includes MySQL Enterprise Backup, which is a repackaging (under Oracle Corporation) of the former InnoDB Hot Backup backup software developed by Innobase, the same company that provided the InnoDB file system to MySQL. Innobase is also owned by Oracle Corporation. For more information on MySQL Enterprise Backup, see *http://www.mysql.com/products/enterprise/backup.html*.

Backing Up a MySQL Database

We will now make a complete backup of the VRG database. Note that MySQL Workbench uses the term **data export dump** as a synonym for backup. Clicking the **Data Export** tabbed window link (under the MANAGEMENT category) displays the **Data Export** page, as shown in Figure 10C-96. To create a backup, select the **vrg** schema, and then click the **Start Export** button.

FIGURE 10C-96

Data Export

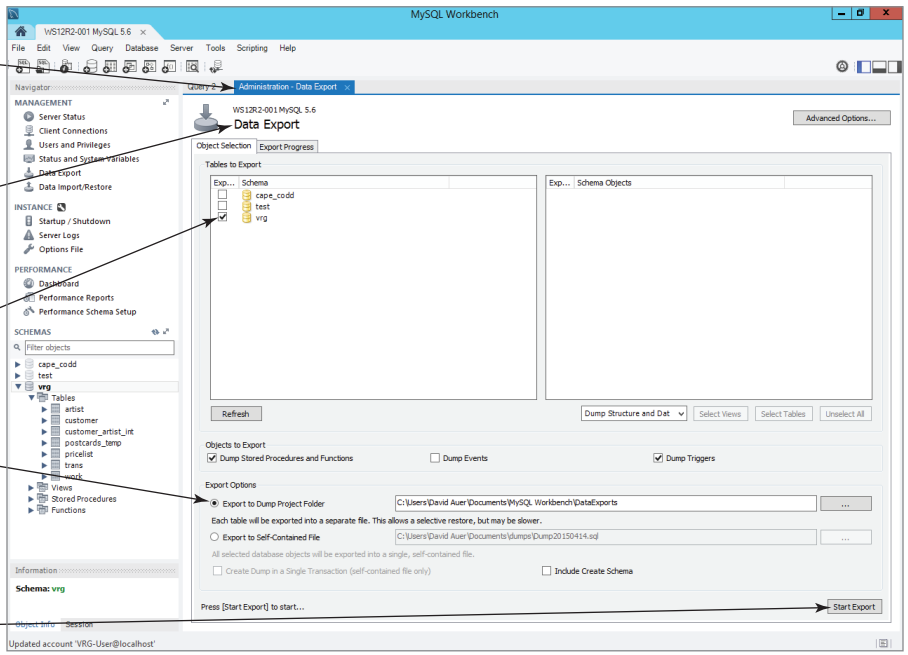
The **Administration - Data Export** tabbed window

The **Data Export** page is used to create and manage MySQL backups

The **vrg** schema has been selected for export

The **Export to Dump Project Folder** option has been selected—the file path shown is to a constructed **DataExports** folder

The **Start Export** button



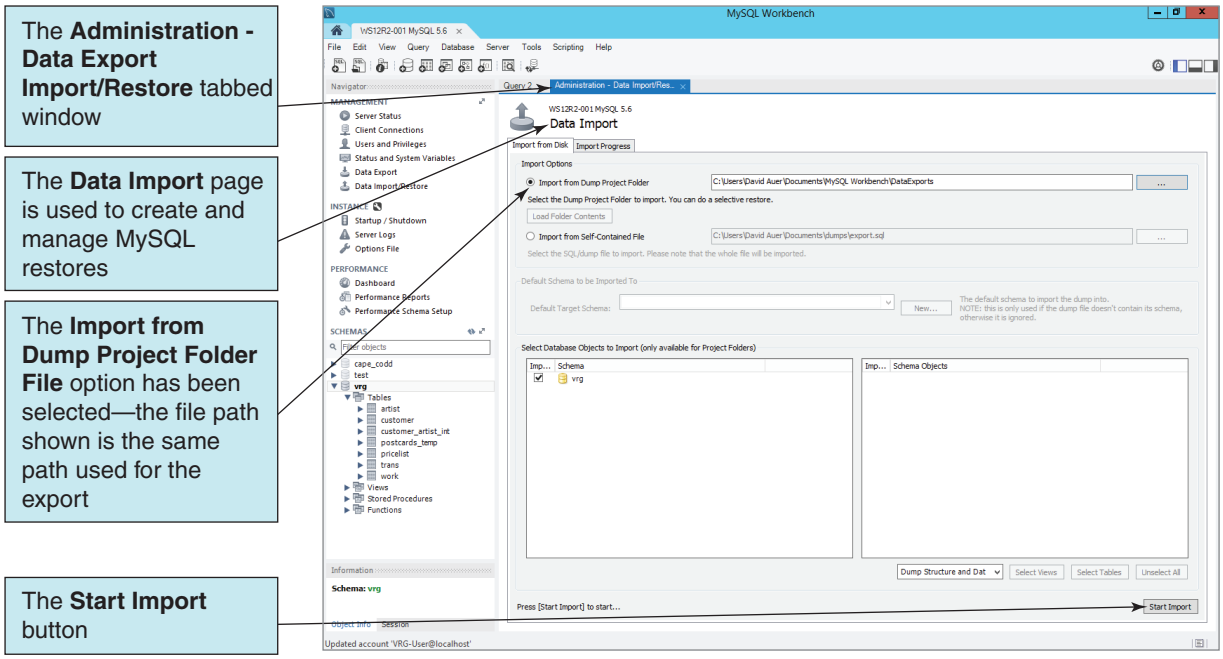


FIGURE 10C-97

Database Restore

Restoring a MySQL Database

For database recovery in MySQL, we will again use the MySQL Workbench. Having made a backup of the VRG database, we can recover it if needed. If the database and log files have been properly backed up, restoring the database is straightforward. However, because we do not need to actually restore the VRG database, we will simply note that restoring a database is a straightforward operation and uses the Import from Disk tab as shown in Figure 10C-97.

You can use backup and restore to transfer a database to another computer or user (such as your professor!). Just do a full backup of the database you want to share to a backup file as we have done, say, the file *MyBackup*. Then create a new database on another computer and name it whatever you want. Finally, restore the database using the MyBackup files as just described.

At this point we are done using MySQL Workbench, and we can close the program.

Topics Not Discussed in This Chapter

Several important MySQL topics are beyond the scope of this discussion. For one, MySQL 5.6 Enterprise provides utilities to measure database activity and performance. The DBA can use these utilities when tuning the database.

MySQL also supports database replication. Although very important in its own right, database replication is beyond the scope of this text. Finally, MySQL has facilities for working with XML.

Summary

MySQL 5.6 can be installed on a variety of operating systems, including Windows and Linux.

Tables, views, indexes, and other database structures can be created in two ways. One is to use the graphical design tools, similar to those in Microsoft Access. The other is to write SQL statements to create the structures and submit them to the DBMS via the MySQL Workbench. MySQL supports all of the SQL DDL features that you have learned in this text, but some of the SQL syntax is different. For example, MySQL uses

the AUTO_INCREMENT attribute for defining surrogate keys. There were no changes for the View Ridge schema itself, but the data values had to be reformatted for the INSERT statements.

Indexes are special data structures used to improve performance. MySQL automatically creates an index on all primary and most foreign keys. Additional indexes can be created using CREATE INDEX or the MySQL Table Editor.

MySQL's SQL syntax complements basic SQL statements with programming constructs such as parameters, variables, and logic structures, such as IF, WHILE, and so forth.

MySQL databases can be processed from application programs coded in programming languages, or application logic can be placed in stored procedures and triggers. Stored procedures can be invoked from standard languages or from PHP and other languages in Web pages. In this chapter, stored procedures were invoked from the MySQL Workbench. This technique should be used only during development and testing. For security reasons, no one should process a MySQL operational database in interactive mode. This chapter demonstrated MySQL triggers and their limitations because of the lack of an INSTEAD OF trigger for use with views.

MySQL concurrency control behavior is controlled primarily at the transaction isolation level. MySQL places locks on behalf of the developer.

MySQL supports different types of backups, but some are available only through third-party software. Backup and restore operations are easily done using the MySQL Workbench.

Key Terms

/* (slash asterisk) and */ (asterisk slash) signs
// (slash slash)
(pound sign)
Add Schema button
Administration - Users and Privileges Window
autocommit mode
AUTO_INCREMENT attribute
BEGIN…END block
BEGIN…END keyword
CLOSE keyword
command-line utility
Commit button
control-of-flow statements
data export dump
database storage engine
DECLARE CURSOR keywords
default schema
delimiter
delimiter //
Edit | Preferences command
FETCH keyword
IF…ELSE…END IF keywords
Forbid UPDATE and DELETE statements with no key in WHERE clause or no LIMIT clause checkbox
graphical user interface (GUI) utility
index
InnoDB storage engine
instance
local instance connection
LOCATE
MyISAM storage engine
MySQL
MySQL administrative roles
MySQL BOOLEAN data type
MySQL Command Line Client
MySQL CONCAT string function
MySQL connection
MySQL CREATE FUNCTION statement
MySQL CURRENT_DATE() function
MySQL function LAST_INSERT_ID()
MySQL Installer for Windows
MySQL Installer utility
MySQL NEW keyword
MySQL OLD keyword
MySQL schema privileges
MySQL Server Instance Configuration Wizard
MySQL Workbench
MySQL Workbench Home tab
Navigator window
OPEN keyword
Output window
parameter
Reconnect to DBMS button
REPEAT keyword
reserved word
Rollback button
ROLLBACK keyword
root user
safe updates
schema
Server Status window
spreadsheet
SQL ALTER TABLE statement
SQL/Persistent Stored Modules (SQL/PSM)
SQL/PSM
SQL Additions window
SQL ALTER PROCEDURE statement
SQL ALTER TABLE ADD INDEX statement
SQL COMMIT command
SQL COMMIT statement
SQL CREATE DATABASE statement
SQL CREATE PROCEDURE statement
SQL DROP PROCEDURE statement
SQL DROP TRIGGER statement
SQL Editor window
SQL Editor window tab
SQL GRANT statement
SQL Persistent Stored Modules (PSM
SQL ROLLBACK statement
SQL script comments
SQL scripts
SQL SHOW TRIGGERS statement
SQL START TRANSACTION statement
SUBSTRING
tabbed MySQL.com Web site window
Toggle autocommit mode button
transaction isolation levels
transition variables
user-defined function (stored function)
Users and Privileges window
variable
WHILE keyword
worksheet

Review Questions

If you have not already installed MySQL 5.6 (or do not otherwise have it available to you), you need to install a version of it at this point.

Review Questions 10C.1–10C.14 are based on a database named MEDIA that is used to record data about photographs that are stored in the database.

10C.1 Create a database named *MEDIA* in MySQL 5.6. Use the default settings for file sizes, names, and locations (the actual database name in MySQL will appear in lowercase letters only).

10C.2 In the MySQL Workbench folder structure in your *Documents* folder, create a folder named *DBP-e14-Media-Database* in the *Schemas* folder. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining Review Questions in this section.

10C.3 Using the MEDIA database, open a new tabbed SQL Query window, and save it as *MEDIA-CH10C-RQ-Solutions.sql* in the *DBP-e14-Media-Database* folder. Use this script record and save the SQL statements that you are asked to create in the remaining Review Questions in this section.

10C.4 Write an SQL CREATE TABLE statement to create a table named PICTURE using the column characteristics as shown in Figure 10C-98. Run the SQL statement to create the PICTURE table in the MEDIA database.

10C.5 Write an SQL CREATE TABLE statement to create the table SLIDE_SHOW using the column characteristics as shown in Figure 10C-99. Run the SQL statement to create the SLIDE_SHOW table in the MEDIA database.

10C.6 Write an SQL CREATE TABLE statement to create the table SLIDE_SHOW_PICTURE_INT using the column characteristics as shown in Figure 10C-100. SLIDE_SHOW_PICTURE_INT is an intersection table between PICTURE and SLIDE_SHOW, so create appropriate relationships between PICTURE and SLIDE_SHOW_PICTURE_INT and between SLIDE_SHOW and SLIDE_SHOW_PICTURE_INT. Set the referential integrity properties to disallow any deletion of a SLIDE_SHOW row that has any SLIDE_SHOW_PICTURE_INT rows related to it. Set the referential integrity properties to cascade deletions in the intersection table when a PICTURE is deleted. Cascade updates to PICTURE.PictureName.

10C.7 Write SQL INSERT statements to populate the PICTURE table using the data shown in Figure 10C-101. Run the SQL statements to populate the PICTURE table.

FIGURE 10C-98

Column Characteristics for the MEDIA Database PICTURE Table

Column Name	Type	Key	Required	Remarks
PictureID	Integer	Primary Key	Yes	Surrogate Key: Initial value=1 Increment=1
PictureName	Character (35)	No	Yes	
PictureDescription	Varchar (255)	No	No	Default “None”
DateTaken	Date	No	Yes	
PictureFileName	Varchar (45)	No	Yes	

FIGURE 10C-99
Column Characteristics for the MEDIA Database SLIDE_SHOW Table

Column Name	Type	Key	Required	Remarks
ShowID	Integer	Primary Key	Yes	Surrogate Key: Initial value=1000 Increment=1
ShowName	Character (35)	No	Yes	
ShowDescription	Varchar (255)	No	No	Default "None"
Purpose	Character (15)	No	Yes	Data value must be one of the following: Home Office Family Recreation Sports Pets

FIGURE 10C-100
Column Characteristics for the MEDIA Database SLIDE_SHOW_PICTURE_INT Table

Column Name	Type	Key	Required	Remarks
ShowID	Integer	Primary Key, Foreign Key	Yes	REF: SLIDE_SHOW
PictureID	Integer	Primary Key, Foreign Key	Yes	REF: PICTURE

10C.8 Write SQL INSERT statements to populate the SLIDE_SHOW table using the data shown in Figure 10C-102. Run the SQL statements to populate the SLIDE_SHOW table.

10C.9 Write SQL INSERT statements to populate the SLIDE_SHOW_PICTURE_INT table using the data shown in Figure 10C-103. Run the SQL statements to populate the SLIDE_SHOW_PICTURE_INT table.

10C.10 Write an SQL statement to create a view named *PopularShowsView* that has SLIDE_SHOW.ShowName and PICTURE.PictureName for all slide shows that have a Purpose of either 'Home' or 'Pets'. Execute this statement to create the view in the MEDIA database.

10C.11 Run an SQL SELECT query to demonstrate that the view PopularShowsView was constructed correctly.

FIGURE 10C-101
Sample Data for the MEDIA Database PICTURE Table

PictureID	PictureName	PictureDescription	Date Taken	PictureFileName
1	SpotAndBall	My dog Spot chasing a ball	2015-09-07	spot00001.jpg
2	SpotAndCat	My dog Spot chasing a cat	2015-09-08	spot00002.jpg
3	SpotAndCar	My dog Spot chasing a car	2015-10-11	spot00003.jpg
4	SpotAndMailman	My dog Spot chasing a Mailman - BAD DOG!	2015-11-22	spot00004.jpg
5	TheJudgeAndI	I explain that Spot is really a good dog. and did not mean to chase the mailman	2015-12-13	me00001.jpg

FIGURE 10C-102
Sample Data for the MEDIA Database SLIDE_SHOW Table

ShowID	ShowName	ShowDescription	Purpose
1000	My Dog Spot	My dog Spot likes to chase things	Pets
1001	My Day In Court	I explain that Spot is really a good dog	Home

10C.12 Use the MySQL Workbench GUI tools to determine that *PopularShowsViews* view was constructed correctly. Modify this view to include PICTURE.PictureDescription and PICTURE.PictureFileName. *Hint:* Right-click the MEDIA **View** object and then **Refresh All** before opening the view.

10C.13 Can the SQL DELETE statement be used with the *PopularShowsView* view? Why or why not?

10C.14 Under what circumstances can the *PopularShowsView* view be used for inserts and modifications?

10C.15 In Figure 10C-66, what is the purpose of the varRowCount variable?

10C.16 In Figure 10C-66, why is the SELECT statement that begins SELECT varRowCount = COUNT(*) necessary?

10C.17 Explain how you would change the stored procedure in Figure 10C-66 to connect the customer to all artists who either (a) were born before 1900 or (b) had a null value for DateOfBirth.

10C.18 Explain the purpose of the transaction shown in Figure 10C-69.

10C.19 What happens if an incorrect value of Copy is input to the stored procedure in Figure 10C-69?

10C.20 In Figure 10C-69, what happens if the ROLLBACK statement is executed?

10C.21 In Figure 10C-73, why is SUM used instead of AVG?

10C.22 What is the primary factor that influences MySQL locking behavior?

10C.23 Explain why the strategy for storing CHECK constraint values in a separate table is better than implementing them in a table-based constraint. How can this strategy be used to implement the constraint on ARTIST.Nationality?

10C.24 Explain why the lack of an INSTEAD OF trigger disallows updating views in MySQL.

10C.25 Explain the meaning of each of the transaction isolation levels under Options shown in Figure 10C-82.

10C.26 How are backups performed using MySQL Administrator?

10C.27 How are recoveries performed using MySQL Administrator?

FIGURE 10C-103
Sample Data for the MEDIA Database SLIDE_SHOW_PICTURE_INT Table

ShowID	PictureID
1000	1
1000	2
1000	3
1000	4
1001	4
1001	5

Project Questions

In the Chapter 7 Review Questions, we introduced the Wedgewood Pacific Corporation (WPC) and developed the WPC database. Two of the tables that are used in the WPC database are:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)

EMPLOYEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)

Assume that the relationship between these tables is M-M, and use them as the basis for your answers to Project Questions 10C.28 through 10C.34.

10C.28 In the MySQL Workbench folder structure in your *MyDocuments* folder, create a folder named *DBP-e14-WPC-CH10C-PQ-Database* in the *Schemas* folder. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

10C.29 Create a MySQL database (schema) named WPC-CH10C-PQ.

10C.30 Using the information about the WPC database in the Chapter 7 Review Questions and the referenced figures in Chapter 1, create the EMPLOYEE and DEPARTMENT tables and the relationship between these tables.

10C.31 Using the information about the WPC database in the Chapter 7 Review Questions and the referenced figures in Chapter 1, populate the EMPLOYEE and DEPARTMENT tables.

10C.32 If possible, code a MySQL trigger to enforce the constraint that an employee can never change his or her department. If it is not possible to use a MySQL trigger, explain why and code a stored procedure in its place.

10C.33 If possible, code a MySQL trigger to allow the deletion of a department if it has only one employee. Before deleting the department, assign the last remaining employee to the Human Resources department. If it is not possible to use a MySQL trigger, explain why and code a stored procedure in its place.

10C.34 If possible, design a system of triggers to enforce the M-M relationship. Use Figure 10C-80 as an example, but assume that departments with only one employee can be deleted. Assign the last employee in a department to Human Resources. If it is not possible to use a MySQL trigger, explain why and code a stored procedure in its place.

10C.35 Create a user named WPC-10C-User with a password of WPC-10C-User+password. Assign WPC-10C-User all schema privileges except GRANT to the WPC-10C-PQ database.

Project Question 10C.36 and 10C.37 are is based on the View Ridge Gallery database discussed in this chapter.

10C.36 Write SQL statements to accomplish the following tasks, and save your work as *.sql scripts as appropriate. Run your *.sql scripts in MySQL by using the MySQL Workbench. Save your work in an SQL script named *VRG-CH10C-PQ-10C-36.sql.*

A. In the MySQL Workbench folder structure in your *MyDocuments* folder, create a folder named *DBP-e14-VRG-CH10C-PQ-Database* in the *Schemas* folder. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

B. Create a MySQL database (schema) named VRG-CH10C-PQ.

C. In the VRG-CH10C-PQ database, create the tables shown in Figure 10C-31, except do not create the NationalityValues constraint.

D. Create, save, and run an SQL script to populate your database with the sample data from Figure 10C-44.

E. Create all the VRG views discussed in the Chapter 7 section on SQL views.

F. Write a stored procedure to add a new artist into the ARTIST table. Research data for two actual artists, and add those data to the ARTIST table using your stored procedure in the MySQL Workbench.

G. Write a stored procedure that adds a new artist to the ARTIST table and a work by that artist to the WORK table. Research data for two actual artists, and add those data to the ARTIST and WORK tables using your stored procedure in the MySQL Workbench.

H. Write a stored procedure to update customer phone data. Assume that your stored procedure receives LastName, FirstName, PriorAreaCode, NewAreaCode, PriorPhoneNumber, and NewPhoneNumber. Your procedure should first ensure that there is only one customer with the values of (LastName, FirstName, PriorAreaCode, PriorPhoneNumber). If not, produce an error message and quit. Otherwise, update the customer data with the new phone number data, and print a results message.

I. Create a table named ALLOWED_NATIONALITY with one column called Nation. Place the values of all nationalities currently in the View Ridge database into the table. If possible, write a trigger that will check to determine whether a new or updated value of Nationality resides in this table, and, if not, write an error message and roll back the insert or change. Use the MySQL Workbench to demonstrate that your trigger works. If it is not possible to write a MySQL trigger, explain why and write an equivalent stored procedure.

J. Create the view named WorkAndTransView as described in this chapter, which has all of the data from the WORK and TRANS tables except for the surrogate keys. Write a stored procedure in lieu of an INSTEAD OF INSERT trigger on this view that will create a new row in both WORK and TRANS. Use the MySQL Workbench to demonstrate that your stored procedure works.

K. Create a user named VRG-10C-User with a password of VRG-10C-User+password. Assign VRG-10C-User all schema privileges except GRANT to the VRG-10C-PQ database.

10C.37 Write SQL statements to accomplish the following tasks, and submit them to MySQL 5.6 via MySQL Workbench. Save your work in an SQL script named *VRG-CH10C-PQ-10C-37sql*. This Project Question shows the steps necessary to integrate the *postcards_temp* table data into the VRG database. A database diagram showing how the VRG database will appear after these steps are completed (drawn in MySQL Workbench) is shown in Figure 10C-104.

A. If you haven't done so, work through Project Question 10C-36 to create the MySQL 5.6 database named *VRG-CH10C-PQ* as described in that Project Question.

B. Use the steps described in this chapter to:

- Create a Microsoft Excel 2013 workbook containing the POSTCARDS worksheet shown in Figure 10C-49.
- Import the data in the POSTCARDS worksheet into a table in the VRG database named *postcards_temp* using the *MySQL for Excel Add-In* as illustrated in the chapter.
- Create the *GetLastNameCommaSeparated* user-defined function shown in Figure 10C-60.
- Alter the postcards_temp table to include the ArtistLastName and ArtistID columns as discussed in the text and as shown in Figure 10C-62.
- Populate the postcards_temp table ArtistLastName and ArtistID columns as discussed in the text and as shown in Figure 10C-65.

C. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, and with the names separated by a comma and one space. Write an SQL SELECT statement using the postcards_temp table to test your function.

D. Alter the postcards_temp table to include an ArtistFirstName column (Char (25) data, allow NULL values). Use the *GetFirstNameCommaSeparated* function that you created in part C to populate this column.

E. Alter the postcards_temp table to include a WorkID column (Integer data, allow NULL values). By using and comparing the data in the postcards_temp.WorkTitle and the WORK.Title columns, populate this column. (*Hint:* In the WORK table, the WorkTitle may appear more than once. For these cases, use the lowest numbered WorkID. This will be the WorkID of the first occurrence of the WorkTitle, and can be found using the MySQL LIMIT clause in an SQL SELECT statement. See the MySQL 5.6 documentation at *http://dev.mysql.com/doc/refman/5.6/en/select.html*).

F. Create a new table named postcard_size_price. Use the column characteristics shown in Figure 10C-104, where PostcardSizePriceID is a surrogate key starting at 1 and incrementing by 1.

G. Populate the postcard_size_price table using the data stored in the postcards_temp table. *Hint:* You should insert distinct data into the table, and your final table will only have 3 records.

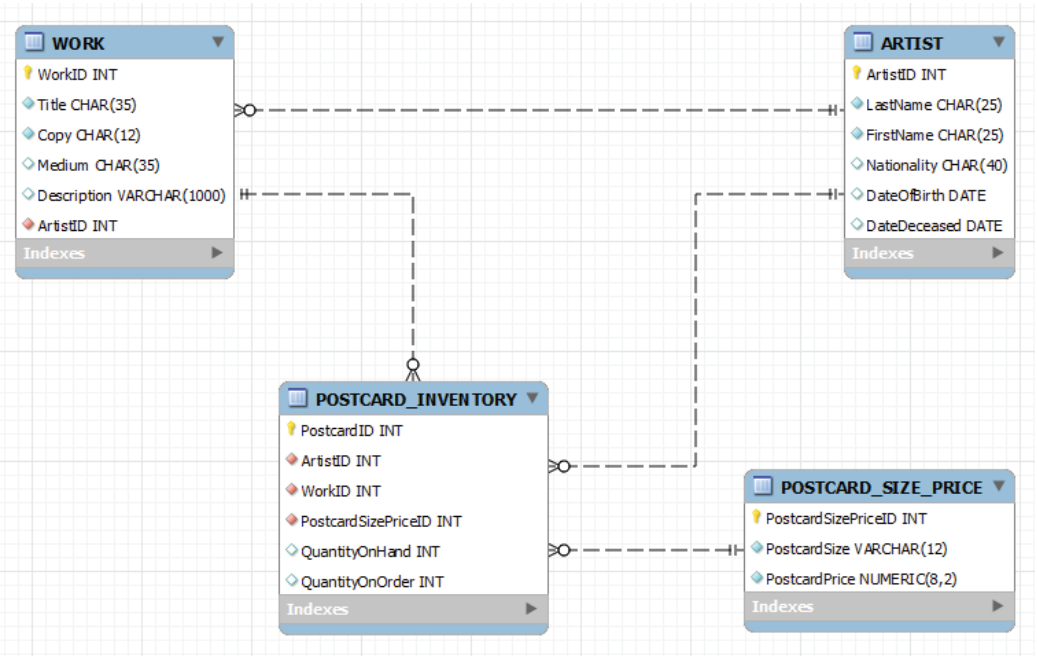
H. Alter the postcards_temp table to include a PostcardSizePriceID column (Integer data, allow NULL values). By using and comparing the data in the postcards_temp.PostCardSize and the postcard_size_price.PostCardSize columns, populate this column.

I. Create a new table named postcard_inventory. Use the column characteristics shown in Figure 10C-104, where PostcardID is a surrogate key starting at 1 and incrementing by 1.

J. Populate the postcard_inventory table using the data stored in the postcards_temp table. *Hint:* You will have a one record in this table for every record in the postcards_temp table, and your final table will only have 26 records.

K. We have completed our modifications of the VRG database, and we are done with the temporary postcards_temp table. We *could* delete if we wanted to, but we will *keep the postcards_temp table* in the database.

FIGURE 10C-104
Partial Database Design for the Revised VRG Database



Case Questions

Marcia's Dry Cleaning Case Questions

Marcia Wilson owns and operates *Marcia's Dry Cleaning*, which is an upscale dry cleaner in a well-to-do suburban neighborhood. Marcia makes her business stand out from the competition by providing superior customer service. She wants to keep track of each of her customers and their orders. Ultimately, she wants to notify them that their clothes are ready via email. Suppose that you have designed a database for Marcia's Dry Cleaning that has the following tables:

CUSTOMER (CustomerID, FirstName, LastName, Phone, Email)

INVOICE (InvoiceNumber, *CustomerID*, DateIn, DateOut, Subtotal, Tax, TotalAmount)

INVOICE_ITEM (*InvoiceNumber*, ItemNumber, *ServiceID*, Quantity, UnitPrice, ExtendedPrice)

SERVICE (ServiceID, ServiceDescription, UnitPrice)

The referential integrity constraints are:

CustomerID in INVOICE must exist in CustomerID in CUSTOMER

InvoiceNumber in INVOICE_ITEM must exist in InvoiceNumber in INVOICE

ServiceID in INVOICE_ITEM must exist in ServiceID in SERVICE

Assume that CustomerID of CUSTOMER and InvoiceNumber of INVOICE are surrogate keys with values as follows:

CustomerID	**Start at 100**	**Increment by 1**
InvoiceNumber	**Start at 2015001**	**Increment by 1**

Further, assume that ServiceID is a surrogate key, but not one that automatically increments—the values of ServiceID are assigned by Marcia's Dry Cleaning management when new services are added at Marcia's Dry Cleaning.

A. Specify NULL/NOT NULL constraints for each table column.

B. Specify alternate keys, if any.

C. State relationships as implied by foreign keys, and specify the maximum and minimum cardinality of each relationship. Justify your choices.

D. Explain how you will enforce the minimum cardinalities in your answer to part C. Use referential integrity actions for required parents, if any. Use Figure 6-29(b) as a boilerplate for required children, if any.

E. Using MySQL 5.6 and the MySQL Workbench, create a database named *MDC*.

F. In the MySQL Workbench folder structure in your *My Documents* folder, create a folder named *DBP-e14-MDC-Database* in the *Schemas* folder. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

Using the MDC database, create an SQL script named *MDC-Create-Tables.sql* to answer parts G and H.

G. Write CREATE TABLE statements for each of the tables using your answers to parts A–D, as necessary. Set the first value of CustomerID to 100 and increment

CustomerID	FirstName	LastName	Phone	Email
100	Nikki	Kaccaton	723-543-1233	Nikki.Kaccaton@somewhere.com
101	Brenda	Catnazaro	723-543-2344	Brenda.Catnazaro@somewhere.com
102	Bruce	LeCat	723-543-3455	Bruce.LeCat@somewhere.com
103	Betsy	Miller	723-654-3211	Betsy.Miller@somewhere.com
104	George	Miller	723-654-4322	George.Miller@somewhere.com
105	Kathy	Miller	723-514-9877	Kathy.Miller@somewhere.com
106	Betsy	Miller	723-514-8766	Betsy.Miller@elsewhere.com

FIGURE 10C-105
Sample Data for the MDC Database CUSTOMER Table

it by 1. Use FOREIGN KEY constraints to create appropriate referential integrity constraints. Set UPDATE and DELETE behavior in accordance with your referential integrity action design. Set the default value of Quantity to 1. Write a constraint that SERVICE.UnitPrice be between 1.50 and 10.00.

H. Explain how you would enforce the data constraint that INVOICE_ITEM.UnitPrice be equal to SERVICE.UnitPrice, where INVOICE_ITEM.ServiceID = SERVICE .ServiceID.

Using the MDC database, create an SQL script named *MDC-Insert-Data.sql* to answer part I.

I. Write INSERT statements to insert the data shown in Figures 10C-105, 10C-106, 10C-107, and 10C-108.

Using the MDC database, create an SQL script named *MDC-DML-CH10C.sql* to answer parts J and K.

J. Write an UPDATE statement to change values of SERVICE.Description from Mens Shirt to Mens' Shirts.

K. Write a DELETE statement(s) to delete an INVOICE and all of the items on that INVOICE.

FIGURE 10C-106
Sample Data for the MDC Database SERVICE Table

ServiceID	ServiceDescription	UnitPrice
10	Men's Shirt	$1.50
11	Dress Shirt	$2.50
15	Women's Shirt	$1.50
16	Blouse	$3.50
20	Slacks—Men's	$5.00
25	Slacks—Women's	$6.00
30	Skirt	$5.00
31	Dress Skirt	$6.00
40	Suit—Men's	$9.00
45	Suit—Women's	$8.50
50	Tuxedo	$10.00
60	Formal Gown	$10.00

FIGURE 10C-107
Sample Data for the MDC Database INVOICE Table

InvoiceNumber	CustomerID	DateIn	DateOut	SubTotal	Tax	TotalAmount
2015001	100	04-Oct-15	06-Oct-15	$158.50	$12.52	$171.02
2015002	101	04-Oct-15	06-Oct-15	$25.00	$1.98	$26.98
2015003	100	06-Oct-15	08-Oct-15	$49.00	$3.87	$52.87
2015004	103	06-Oct-15	08-Oct-15	$17.50	$1.38	$18.88
2015005	105	07-Oct-15	11-Oct-15	$12.00	$0.95	$12.95
2015006	102	11-Oct-15	13-Oct-15	$152.50	$12.05	$164.55
2015007	102	11-Oct-15	13-Oct-15	$7.00	$0.55	$7.55
2015008	106	12-Oct-15	14-Oct-15	$140.50	$11.10	$151.60
2015009	104	12-Oct-15	14-Oct-15	$27.00	$2.13	$29.13

FIGURE 10C-108
Sample Data for the MDC Database INVOICE_ITEM Table

InvoiceNumber	ItemNumber	ServiceID	Quantity	UnitPrice	ExtendedPrice
2015001	1	16	2	$3.50	$7.00
2015001	2	11	5	$2.50	$12.50
2015001	3	50	2	$10.00	$20.00
2015001	4	20	10	$5.00	$50.00
2015001	5	25	10	$6.00	$60.00
2015001	6	40	1	$9.00	$9.00
2015002	1	11	10	$2.50	$25.00
2015003	1	20	5	$5.00	$25.00
2015003	2	25	4	$6.00	$24.00
2015004	1	11	7	$2.50	$17.50
2015005	1	16	2	$3.50	$7.00
2015005	2	11	2	$2.50	$5.00
2015006	1	16	5	$3.50	$17.50
2015006	2	11	10	$2.50	$25.00
2015006	3	20	10	$5.00	$50.00
2015006	4	25	10	$6.00	$60.00
2015007	1	16	2	$3.50	$7.00
2015008	1	16	3	$3.50	$10.50
2015008	2	11	12	$2.50	$30.00
2015008	3	20	8	$5.00	$40.00
2015008	4	25	10	$6.00	$60.00
2015009	1	40	3	$9.00	$27.00

Using the MDC database, create an SQL script named *MDC-Create-Views-and-Functions.sql* to answer parts L through T.

L. Create a view called OrderSummaryView that contains INVOICE.InvoiceNumber, INVOICE.DateIn, INVOICE.DateOut, INVOICE_ITEM.ItemNumber, INVOICE_ITEM.ServiceID, and INVOICE_ITEM.ExtendedPrice.

M. Create a view called CustomerOrderSummaryView that contains INVOICE.InvoiceNumber, CUSTOMER.FirstName, CUSTOMER.LastName, CUSTOMER.Phone, INVOICE.DateIn, INVOICE.DateOut, INVOICE.SubTotal, INVOICE_ITEM.ItemNumber, INVOICE_ITEM.ServiceID, and INVOICE_ITEM.ExtendedPrice.

N. Create a view called CustomerOrderHistoryView that (1) includes all columns of CustomerOrderSummaryView except INVOICE_ITEM.ItemNumber and INVOICE_ITEM.Service; (2) groups orders by CUSTOMER.LastName, CUSTOMER.FirstName, and INVOICE.InvoiceNumber, in that order; and (3) sums and averages INVOICE_ITEM.ExtendedPrice for each order for each customer.

O. Create a view called CustomerOrderCheckView that uses CustomerOrderHistoryView and that shows any customers for whom the sum of INVOICE_ITEM.ExtendedPrice is not equal to INVOICE.SubTotal.

P. Explain, in general terms, how you will use triggers to enforce minimum cardinality actions as required by your design. You need not write the triggers, just specify which triggers you need and describe, in general terms, their logic.

Q. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

R. Create and test a view called CustomerOrderSummaryView that contains the customer name concatenated and formatted as *LastName, FirstName* in a field named CustomerName, INVOICE.InvoiceNumber, INVOICE.DateIn, INVOICE.DateOut, and INVOICE.TotalAmount.

S. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the comma and space).

T. Create and test a view called CustomerDataView that contains the customer name concatenated and formatted as *FirstName LastName* in a field named CustomerName, Phone, Email.

Using the MDC database, create an SQL script named *MDC-Create-Triggers.sql* to answer parts U and V.

U. Assume that the relationship between INVOICE and INVOICE_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10C-80 and the discussion of that figure as an example, but assume that Marcia does allow INVOICEs and their related INVOICE_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

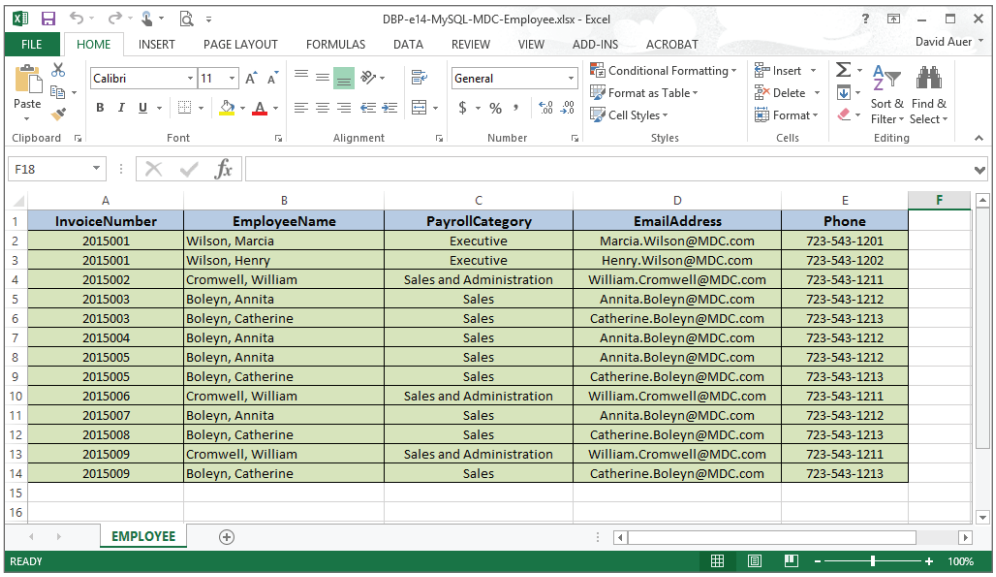
V. Write and test the triggers you designed in part U.

W. Create a user named MDC-10C-User with a password of MDC-10C-User+password. Assign MDC-10C-User all schema privileges except GRANT to the MDC database.

Marcia's Dry Cleaning tracks which employees have worked on specific dry cleaning jobs. This is somewhat complicated by the fact that more than one employee may have helped a customer with a particular order. So far, the company has kept their insurance records in a Microsoft Excel 2013 worksheet, as shown in Figure 10C-109. They have decided to integrate this data into the MDC database. The modifications

FIGURE 10C-109

The Marcia's Dry Cleaning Employee Worksheet



	A	B	C	D	E	F
1	InvoiceNumber	EmployeeName	PayrollCategory	EmailAddress	Phone	
2	2015001	Wilson, Marcia	Executive	Marcia.Wilson@MDC.com	723-543-1201	
3	2015001	Wilson, Henry	Executive	Henry.Wilson@MDC.com	723-543-1202	
4	2015002	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
5	2015003	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
6	2015003	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
7	2015004	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
8	2015005	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
9	2015005	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
10	2015006	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
11	2015007	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
12	2015008	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
13	2015009	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
14	2015009	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
15						
16						

to the MDC database needed to accomplish this are shown in Figure 10C-110 (as a MySQL Workbench EER diagram). Using the MDC database, create an SQL script named *MDC-Import-Excel-Data.sql* to answer parts X through AJ.

X. Duplicate the EMPLOYEE worksheet in Figure 10C-109 in a worksheet (or spreadsheet) in Microsoft Excel 2013 (or another tool such as Apache OpenOffice Calc).

Y. Import the data in the EMPLOYEE worksheet into a table in the MDC database named EMPLOYEE_TEMP. *Hint:* Use the *MySQL for Excel Add-In*, which *exports* the data from Microsoft Excel to MySQL.

Z. Create the *GetLastNameCommaSeparated* user-defined function shown in Figure 10C-60.

AA. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

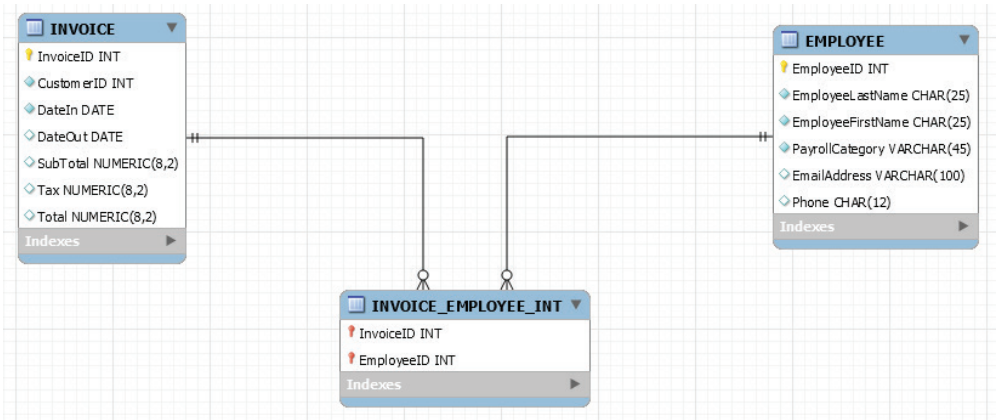
AB. Alter the EMPLOYEE_TEMP table to include EmployeeLastName and EmployeeFirstName columns (Char(25), allow NULL values).

AC. Use the *GetLastNameCommaSeparated* user-defined function you created in step Z to populate the EmployeeLastName column.

AD. Use the *GetFirstNameCommaSeparated* user-defined function you created in step AA to populate the EmployeeFirstName column.

FIGURE 10C-110

Partial Database Design for the Modified MDC Database



AE. Create a new table named EMPLOYEE, as shown in Figure 10C-110. Use the column characteristics shown in Figure 10C-110, where EmployeeID is a surrogate key starting at 1 and incrementing by 1.

AF. Populate the EMPLOYEE table using the data stored in the EMPLOYEE_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 5 records.

AG. Alter the EMPLOYEE_TEMP table to include an EmployeeID column (Integer data, allow nulls). By using and comparing the EMPLOYEE_TEMP.EmployeeLastName and EMPLOYEE_TEMP.EmployeeFirstName columns with the EMPLOYEE .EmployeeLastName and EMPLOYEE.EmployeeFirstName columns, populate this column. *Hint:* Assume for this question that no two employees have the same first and last names.

AH. Create a new table named INVOICE_EMPLOYEE_INT, as shown in Figure 10C-110. Use the column characteristics shown in Figure 10C-110.

AI. Populate the INVOICE_EMPLOYEE_INT table using the data stored in the EMPLOYEE_TEMP table. *Hint:* You will have one record in the INVOICE _EMPLOYEE_INT table for every record in the EMPLOYEE_TEMP table, and your final table will have 13 records.

AJ. We have completed the modifications of the MDC database, and are done with the temporary EMPLOYEE_TEMP table. We *could* delete it if we wanted to, but we will *keep the EMPLOYEE_TEMP table* in the database.

The Queen Anne Curiosity Shop

If you have not completed the discussion of the Queen Anne Curiosity Shop database at the end of Chapter 7 on pages 383–389, work through the Chapter 7 QACS Project Questions now. Use the QACS database that you created in the Chapter 7 QACS Project Questions as the basis for your answers to the following questions:

Using the QACS database, create an SQL script named *QACS-Create-Views-and-Functions.sql* to answer parts A through E.

A. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

B. Create and test a view called CustomerSaleSummaryView that contains the customer name concatenated and formatted as *LastName, FirstName* in a field named CustomerName, SALE.SaleID, SALE.SaleDate, and SALE.Total.

C. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the space).

D. Create and test a user-defined function named *CityStateZIP* that combines three parameters named *City, State,* and *ZIP* into a concatenated name field formatted *City, State ZIP* (including the comma and the spaces).

E. Create and test a view called CustomerMailingAddressView that contains the customer name concatenated and formatted as *FirstName LastName* in a field named CustomerName, the customer's street address in a field named CustomerStreetAddress, and the customer's City, State, ZIP concatenated and formatted as *City, State ZIP* in a field named CustomerCityStateZIP.

Using the QACS database, create an SQL script named *QACS-Create-Triggers.sql* to answer parts F and G.

F. Assume that the relationship between SALE and SALE_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10C-80 and the discussion of that figure

as an example, but assume that the Queen Anne Curiosity Shop does allow SALEs and their related SALE_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

G. Write and test the triggers you designed in part F.

H. Create a user named QACS-10C-User with a password of QACS-10C-User+password. Assign QACS-10C-User all schema privileges except GRANT to the QACS database.

The Queen Anne Curiosity Shop payroll is paid twice monthly, once on the 10th of the month and once on the 25th of the month. Pay is determined by the type of job (the payroll category) and the number of hours worked (rounded to a whole number). Of course, the QACS owners keep detailed payroll records. So far, they have kept their records for these items in a Microsoft Excel worksheet, as shown in Figure 10C-111. They have decided to integrate this data into the QACS database. Using the QACS database, create an SQL script named *QACS-Import-Excel-Data.sql* to answer parts I through T.

I. Duplicate the PAYROLL worksheet in Figure 10C-111 in a worksheet (or spreadsheet) in an appropriate tool (such as Microsoft Excel or Apache OpenOffice Calc).

J. Import the data in the PAYROLL worksheet into a table in the QACS database named PAYROLL_TEMP. *Hint:* Use the *MySQL for Excel Add-In*, which *exports* the data from Microsoft Excel to MySQL.

K. Create the *GetLastNameCommaSeparated* user-defined function shown in Figure 10C-60.

L. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

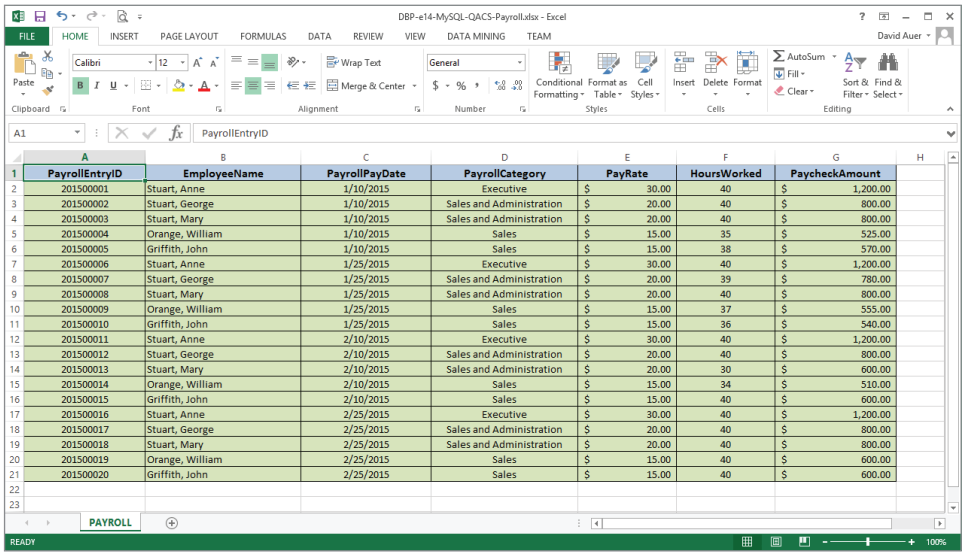
M. Alter the PAYROLL_TEMP table to include EmployeeLastName and EmployeeFirstName columns (Char(25), allow NULL values).

N. Use the *GetLastNameCommaSeparated* user-defined function you created in step K to populate the EmployeeLastName column.

O. Use the *GetFirstNameCommaSeparated* user-defined function you created in step L to populate the EmployeeFirstName column.

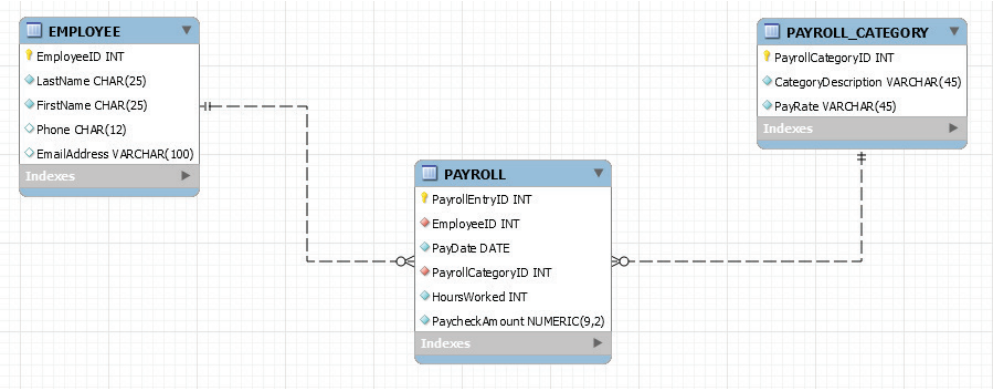
P. Create a new table named PAYROLL_CATEGORY, as shown in Figure 10C-112. Use the column characteristics shown in Figure 10C-112, where PayrollCategoryID is a surrogate key starting at 1 and incrementing by 1.

FIGURE 10C-111
The Queen Anne Curiosity Shop Payroll Worksheet



PayrollEntryID	EmployeeName	PayrollPayDate	PayrollCategory	PayRate	HoursWorked	PaycheckAmount
201500001	Stuart, Anne	1/10/2015	Executive	$ 30.00	40	$ 1,200.00
201500002	Stuart, George	1/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500003	Stuart, Mary	1/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500004	Orange, William	1/10/2015	Sales	$ 15.00	35	$ 525.00
201500005	Griffith, John	1/10/2015	Sales	$ 15.00	38	$ 570.00
201500006	Stuart, Anne	1/25/2015	Executive	$ 30.00	40	$ 1,200.00
201500007	Stuart, George	1/25/2015	Sales and Administration	$ 20.00	39	$ 780.00
201500008	Stuart, Mary	1/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500009	Orange, William	1/25/2015	Sales	$ 15.00	37	$ 555.00
201500010	Griffith, John	1/25/2015	Sales	$ 15.00	36	$ 540.00
201500011	Stuart, Anne	2/10/2015	Executive	$ 30.00	40	$ 1,200.00
201500012	Stuart, George	2/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500013	Stuart, Mary	2/10/2015	Sales and Administration	$ 20.00	30	$ 600.00
201500014	Orange, William	2/10/2015	Sales	$ 15.00	34	$ 510.00
201500015	Griffith, John	2/10/2015	Sales	$ 15.00	40	$ 600.00
201500016	Stuart, Anne	2/25/2015	Executive	$ 30.00	40	$ 1,200.00
201500017	Stuart, George	2/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500018	Stuart, Mary	2/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500019	Orange, William	2/25/2015	Sales	$ 15.00	40	$ 600.00
201500020	Griffith, John	2/25/2015	Sales	$ 15.00	40	$ 600.00

FIGURE 10C-112
Partial Database Design for the Modified QACS Database



Q. Populate the PAYROLL_CATEGORY table using the data stored in the PAYROLL_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 3 records.

R. Alter the PAYROLL_TEMP table to include an EmployeeID column (Integer data, allow nulls). By using and comparing the PAYROLL_TEMP.EmployeeLastName and PAYROLL_TEMP.EmployeeFirstName columns with the EMPLOYEE.LastName and EMPLOYEE.FirstName columns, populate this column. *Hint:* Assume for this question that no two employees have the same first and last names.

S. Alter the PAYROLL_TEMP table to include a PayrollCategoryID column (Integer data, allow nulls). By using and comparing the PAYROLL_TEMP.PayrollCategory column with the PAYROLL_CATEGORY.CategoryDescription column, populate this column.

T. Create a new table named PAYROLL, as shown in Figure 10C-112. Use the column characteristics shown in Figure 10C-112. Note that PayrollEntryID is a surrogate key, with initial value 20150001 and incrementing by 1.

U. Populate the PAYROLL table using the data stored in the PAYROLL_TEMP table. *Hint:* You will have one record in the PAYROLL table for every record in the PAYROLL_TEMP table, and your final table will have 20 records.

V. We have completed the modifications of the QACS database, and are done with the temporary PAYROLL_TEMP table. We *could* delete it if we wanted to, but we will *keep the PAYROLL_TEMP table* in the database.

Morgan Importing

If you have not completed the discussion of Morgan Importing database at the end of Chapter 7 on pages 390–395, work through the Chapter 7 Morgan Importing Project Questions now. Use the MI database that you created in the Chapter 7 Morgan Importing Project Questions as the basis for your answers to the following questions:

Using the MI database, create an SQL script named *MI-Create-Views-and-Functions.sql* to answer parts A through D.

A. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

B. Create and test a view called *PurchasingAgentSummaryView* that contains the employee name of any MI employees who purchase items for the company, concatenated and formatted as *LastName, FirstName* in a field named PurchasingAgentName, ITEM.ItemDescription, ITEM.PurchaseDate, STORE.StoreName, STORE.City, Store. Country.

C. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the space).

D. Create and test a view called ReceivingAgentSummaryView that contains the employee name of any MI employees who received items for the company, concatenated and formatted as *FirstName LastName* in a field named ReceivingAgentName, SHIPMENT_RECEIPT.ReceiptNumber, SHIPMENT.ShipmentID, SHIPPER.ShipperName, SHIPMENT.EstimatedArrivalDate, SHIPMENT_RECEIPT.ReceiptDate, SHIPMENT_RECEIPT.ReceiptTime.

Using the MI database, create an SQL script named *MI-Create-Triggers.sql* to answer parts E and F.

E. Assume that the relationship between SHIPMENT and SHIPMENT_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10C-80 and the discussion of that figure as an example, but assume that Morgan does allow SHIPMENTs and their related SHIPMENT_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

F. Write and test the triggers you designed in part E.

G. Create a user named MI-10C-User with a password of MI-10C-User+password. Assign MI-10C-User all schema privileges except GRANT to the MI database.

Morgan Importing purchases marine insurance to protect the company from monetary loss during shipping. So far, the company has keep their insurance records in a Microsoft Excel 2013 worksheet, as shown in Figure 10C-113. They have decided to integrate this data into the MI database. The modifications to the MI database needed to accomplish this are shown in Figure 10C-114 (as a MySQL Workbench EER diagram). Using the MI database, create an SQL script named *MI-Import-Excel-Data.sql* to answer parts H through T.

H. Duplicate the INSURANCE worksheet Figure Figure 10C-113 in a worksheet (or spreadsheet) in an Mircosoft Excel 2013 (or another tool such as Apache OpenOffice Calc).

I. Import the data in the INSURANCE worksheet into a table in the MI database named INSURANCE_TEMP. *Hint:* Use the *MySQL for Excel Add-In*, which *exports* the data from Microsoft Excel to MySQL.

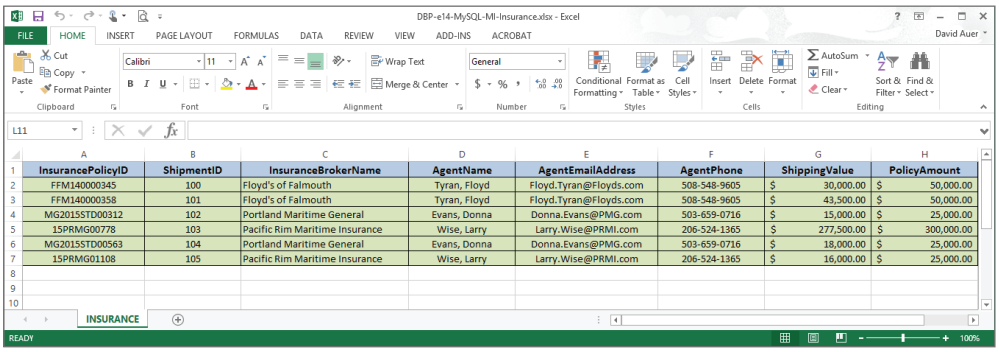
J. Create the *GetLastNameCommaSeparated* user-defined function shown in Figure 10C-60.

K. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

L. Alter the INSURANCE_TEMP table to include AgentLastName and AgentFirstName columns (Varchar(45), allow NULL values).

M. Use the *GetLastNameCommaSeparated* user-defined function you created in step J to populate the AgentLastName column.

FIGURE 10C-113
The Morgan Importing Maritime Insurance Worksheet



InsurancePolicyID	ShipmentID	InsuranceBrokerName	AgentName	AgentEmailAddress	AgentPhone	ShippingValue	PolicyAmount
FFM140000345	100	Floyd's of Falmouth	Tyran, Floyd	Floyd.Tyran@Floyds.com	508-548-9605	$ 30,000.00	$ 50,000.00
FFM140000358	101	Floyd's of Falmouth	Tyran, Floyd	Floyd.Tyran@Floyds.com	508-548-9605	$ 43,500.00	$ 50,000.00
MG2015STD00312	102	Portland Maritime General	Evans, Donna	Donna.Evans@PMG.com	503-659-0716	$ 15,000.00	$ 25,000.00
15PRMG00778	103	Pacific Rim Maritime Insurance	Wise, Larry	Larry.Wise@PRMI.com	206-524-1365	$ 277,500.00	$ 300,000.00
MG2015STD00563	104	Portland Maritime General	Evans, Donna	Donna.Evans@PMG.com	503-659-0716	$ 18,000.00	$ 25,000.00
15PRMG01108	105	Pacific Rim Maritime Insurance	Wise, Larry	Larry.Wise@PRMI.com	206-524-1365	$ 16,000.00	$ 25,000.00

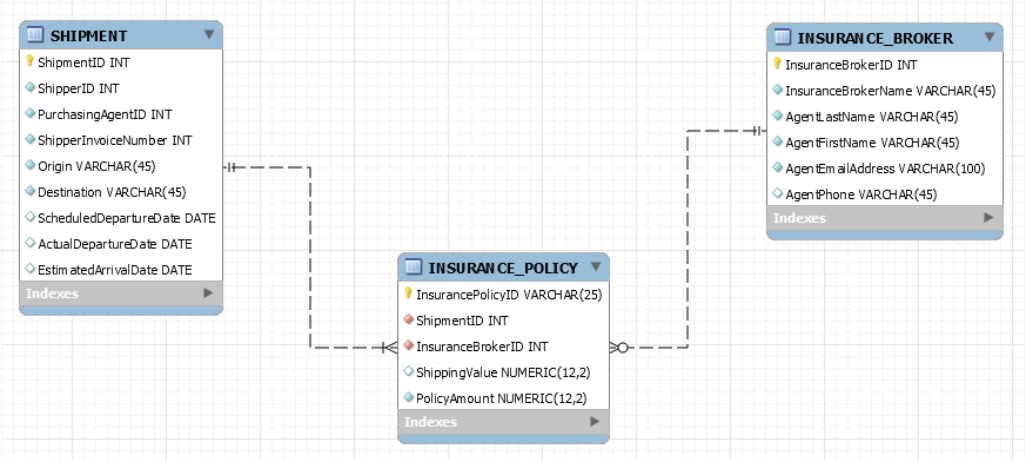


FIGURE 10C-114

Partial Database Design for the Modified MI Database

N. Use the *GetFirstNameCommaSeparated* user-defined function you created in step K to populate the AgentFirstName column.

O. Create a new table named INSURANCE_BROKER. Use the column characteristics shown in Figure 10C-114, where InsuranceBrokerID is a surrogate key starting at 1 and incrementing by 1.

P. Populate the INSURANCE_BROKER table using the data stored in the INSURANCE_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 3 records.

Q. Create a new table named INSURANCE_POLICY. Use the column characteristics shown in Figure 10C-114. Note that InsurancePolicyID is not a surrogate key, but rather uses a Varchar (25) character string.

R. Alter the INSURANCE_TEMP table to include an InsuranceBrokerID column (Integer data, allow nulls). By using and comparing the INSURANCE_TEMP .InsuranceBrokerName and INSURANCE_BROKER.InsuranceBrokerName columns, populate this column. *Hint:* Assume for this question that no two insurance broker names are the same.

S. Populate the INSURANCE_POLICY table using the data stored in the INSURANCE_TEMP table. *Hint:* You will have one record in the INSURANCE_POLICY table for every record in the INSURANCE_TEMP table, and your final table will have 6 records.

T. We have completed the modifications of the MI database, and are done with the temporary INSURANCE_TEMP table. We *could* delete it if we wanted to, but we will *keep the INSURANCE_TEMP table* in the database.