



Gerald V. Post

**Database
Management
SYSTEMS**

Oracle 11g

**Designing & Building Business Applications
Fifth Edition**

Database Management Systems

Designing and Building
Business Applications
With
Oracle 11g

Version 5.0.1

Gerald V. Post

University of the Pacific

Database Management Systems
Designing and Building Business Applications
With Oracle

Copyright © 2010 by Gerald V. Post

All rights reserved. No part of this publication may be reproduced or distributed in any form or stored in any database or retrieval system without the prior written consent of Gerald V. Post.

Students:

Your honesty is critical to your reputation. No company wants to hire a thief—particularly for jobs as critical as application development and database administration. If someone is willing to steal something as inexpensive as an e-book, how can that person be trusted with billions of dollars in corporate accounts?

You are not allowed to “share” this book in any form with anyone else. You cannot give or sell any information from this publication in any form to anyone else.

To purchase this book or other books: <http://JerryPost.com/books>

Brief Contents

1 Introduction

Part One: Systems Design

2 Database Design

3 Data Normalization

PART TWO: QUERIES

4 Data Queries

5 Advanced Queries and Subqueries

PART THREE: APPLICATIONS

6 Forms and Reports

7 Database Integrity and Transactions

8 Applications

9 Data Warehouses and Data Mining

PART FOUR: DATABASE ADMINISTRATION

10 Database Administration

11 Distributed Databases

12 Physical Data Storage

Introduction, 1

- Case: All Powder Board and Ski Shop, 2
 - Inventory*, 3
 - Bindings and Boots*, 4
 - Sales*, 5
 - Rentals*, 6
- Lab Exercise, 6
 - Project Outline*, 7
 - Project Plan*, 7
 - Feasibility*, 9
 - The Database Management System*, 9
 - Create a Table*, 12
 - Create a Form*, 15
- Exercises, 22
- Final Project, 24

Database Design, 25

- Database Design, 26
- Oracle Data Types, 26
- Case: All Powder Board and Ski Shop, 28
 - Business Objects: First Guess*, 28
 - Relationships*, 29
- Lab Exercise, 30
 - Database Design System*, 30
 - All Powder Design*, 31
- Exercises, 38
- Final Project, 40

Data Normalization, 41

- Database Design, 42
- Generated Keys: Sequences, 43
- Case: All Powder Board and Ski Shop, 43
- Lab Exercise, 44
 - All Powder Board and Ski Database Creation*, 44
- Exercises, 56
- Final Project, 57

Database Queries and SQL, 58

- Database Queries, 59
- Case: All Powder Board and Ski Shop, 59
- Lab Exercise, 60
 - All Powder Board and Ski Data*, 60
 - Computations and Subtotals*, 69
- Exercises, 75
- Final Project, 76

Advanced Queries, 77

- Advanced Database Queries, 78
- Case: All Powder Board and Ski Shop, 79
- Lab Exercise, 79
 - All Powder Board and Ski Data*, 79
 - SQL Data Definition and Data Manipulation*, 90
- Exercises, 96
- Final Project, 98

Forms and Reports, 99

- Forms and Reports, 100
 - Model-View-Controller*, 101
- Case: All Powder Board and Ski Shop, 102
- Lab Exercise, 103
 - All Powder Board and Ski Shop Forms*, 103
 - All Powder Basic Reports*, 131
- Exercises, 136
- Final Project, 137

Database Integrity and Transactions, 138

- Program Code in Oracle, 139
- Case: All Powder Board and Ski Shop, 144
- Lab Exercise, 144
 - All Powder Board and Ski Data*, 144
 - Database Cursors, Keys, and Locks*, 168
- Exercises, 172
- Final Project, 174

Applications, 175

- Applications, 176
- Case: All Powder Board and Ski Shop, 176
- Lab Exercise, 177
 - All Powder Board and Skip Shop Application*, 177
 - Connecting Pages with Task Flows*, 185
 - Testing Login Credentials*, 190
 - A Report for One Customer Using the Login Data*, 194
 - Connect Table Row to Detail Report*, 198
- Exercises, 215
- Final Project, 216

Data Warehouses and Data Mining, 217

- Data Warehouse, 218
 - Tools and Downloads*, 219

Case: All Powder Board and Ski Shop, 220
Lab Exercise, 220
 All Powder Board and Ski Shop, 220
 Introductory Data Analysis, 236
Exercises, 246
Final Project, 248

Database Administration, 250

Database Administration Tasks, 251
Case: All Powder Board and Ski Shop, 252
Lab Exercise, 253
 All Powder Board and Ski Shop, 253
 Security and Privacy, 261
Exercises, 268
Final Project, 269

Distributed Databases, 270

Location, Location, Location, 271
Case: All Powder Board and Ski Shop, 272
Lab Exercise, 272
 All Powder Board and Ski Shop, 272
 The Internet, 276
Exercises, 279
Final Project, 280

Physical Database Design, 282

Storing Data, 283
Case: All Powder Board and Ski Shop, 284
Lab Exercise, 285
 All Powder Board and Ski Shop, 285
 Data Clusters, 288
Exercises, 291
Final Project, 292

Introduction

Chapter Outline

Case: All Powder Board and Ski Shop, 2

Inventory, 3

Bindings and Boots, 4

Sales, 5

Rentals, 6

Lab Exercise, 6

Project Outline, 7

Project Plan, 7

Feasibility, 9

The Database Management System, 9

Create a Table, 12

Create a Form, 15

Exercises, 22

Final Project, 24

Objectives

- Identify the main elements of the case.
- Structure the work needed for the case.
- Create a feasibility analysis of the case.
- Create a new database.

Oracle is a complex product with many options. Databases reflect complex interactions among data. These complexities make it challenging to learn to create databases and to use the DBMS software. This lab workbook explains most of the basic steps needed to design a database and build an initial database application. However, it is difficult to remember all of the tasks that you will perform in an activity. You should seriously consider keeping a lab notebook or journal. Whenever you have to make a decision or solve a problem, write a note in the journal with the basic information and explain your decision. Later, if you encounter a similar problem, you can look at your notes to see which techniques worked and which did not. This journal could be electronic—which will make it easier to organize and search later. But, sometimes it is faster to keep a paper journal.

Case: All Powder Board and Ski Shop

The ski industry has been through many changes in the 50 years since Bill Shimek founded the ski shop that is now run by his grandson. One of the biggest changes is reflected in the prominence of “Board” in the shop name. Snowboards have revolutionized the industry in several respects. They revived youth interest in the sport, brought new designs to equipment and resorts, and increased sales dramatically. On the other hand, the increased changes in ski and snowboard equipment make it more difficult for shops to stock the hundreds of options and combinations that enthusiasts might want. Shops have become larger, forcing small firms out of business. Even large ski shops have had to identify their customers and forecast customer demands carefully to make sure the high-demand equipment is in stock. Tracking sales, trends, and buyer needs has become critical to survival.

Another factor in the industry is that the firms increasingly rely on rentals. Partly because of the rapid changes in the industry, many people prefer to rent equipment so they can avoid having to buy new boards and skis every year. Consequently, the shop buys several relatively standard boards and skis every year and rents them out. At the end of the year, the used equipment is sold at a discount to make room for next year’s models.

Figure 1.1

Inventory							
Snowboards							
	Manufacturer	Mfg ID	Size	Description	Graphics	List Price	QOH
Freestyle							
Pipe							
Standard							
Extreme							
Skis							
	Manufacturer	Mfg ID	Size	Description	Graphics	List Price	QOH
Cross country-skate							
Cross country-trad.							
Telemark							
Jumping							
Freestyle							
Downhill/race							

Inventory

Monitoring inventory is a first critical step in the process of providing the selection demanded by customers. Figure 1.1 shows some of the detailed information needed, as well as the diversity of equipment available. Note that because of the variety of styles, there are many different types of snowboards and skis. Figure 1.1 shows the importance of the skill categories. Manufacturers produce special boards and skis for each of these categories. Of course, it would be impossible to stock all of the required sizes for rental purposes. Rental boards and skis tend to be as generic as possible. Even for sales, some sizes of the high-end skis and boards have to be special ordered.

Within a category, manufacturers tend to sell boards and skis targeted for different levels of skiers—from beginner to intermediate to expert (Type I, Type II, and Type III skier). Even within the type classifications, All Powder salespeople evaluate customers on the basis of their aggressiveness on the slope. Because of the size of snowboards, along with the youthful image of the sport, manufacturers place a high value on the graphics (images and colors) displayed on both sides of the boards. Customers have often been known to choose a board because of the graphics. Some of this emphasis on graphics has filtered down to skis as well.

Sizes of boards and skis are somewhat tricky and definitely present a challenge to keeping adequate inventory. The length of the ski or board is a critical number, but the customer's choice is also based on several other ski measurements. Snowboards revolutionized board and ski design by adding a narrower waist to aid in turning. This concept migrated to most varieties of skis as well. So customers often want to know the waist width, sideout depth, and effective edge length as well. Generally, boards and skis with narrower waists are targeted for more advanced skiers. Additionally, the construction of the board or ski, in terms of materials and thickness, significantly affects its flexibility and handling. Customers generally want to feel the ski to evaluate and compare its flexibility, but measures of stance location (for boards) and the rider weight range provide some estimate of the handling characteristics. Most skis and boards are also designed for a particular riding weight. For cross-country skis it is particularly important to get the proper length

Figure 1.2

Boot-Binding Compatibility						
Manuf.	Mfg. ID	Board/Ski	Binding/Style	Color	Price	Cost

for the weight of the skier. With cross-country skis, the central part of the ski, under the skier's foot needs to float above the snow when weight is balanced on both skis. When the skier steps down, it pushes the kick area into the snow to provide traction.

Bindings and Boots

Bindings and boots represent another common problem for All Powder and other ski shops. Each ski and each board can technically be fitted with several types of bindings. Each binding type generally requires a matching style of boot; and some of the boots can work only with some bindings. For example, snowboards can use clincher, strap, or plate bindings. Cross-country skis can use pin, strap, or rod bindings. Most modern skis use the rod binding, but customers sometimes want boots that fit the older pin bindings. Downhill, freestyle, and slalom skis use similar bindings, and they are the most popular so the store usually stocks several models focusing on skill levels.

Figure 1.2 shows an example of the card system that All Powder uses to help salespeople select bindings and boots. Currently, the salespeople are supposed to change the quantity on hand whenever a boot or binding is sold. Of course, the cards are rarely kept up-to-date and the salespeople often have to go search the physical inventory to see if a size needed by a customer is in stock. Note that boots and bindings are specifically matched and a boot for one purpose can rarely be used for a different application. For example, it would not be possible to use a cross-country boot in a downhill binding. The binding is usually listed as a type (rod, step-in, telemark/cable, and so on). On the other hand, it is possible to mount bindings on different types of skis. For instance, you could mount a telemark binding to a downhill ski. Some of the strange combinations should be avoided, but this knowledge will not be needed in the database.

Figure 1.3

Sales						
Customer					Sale Date	
First Name	Last Name				Salesperson	
Phone	E-Mail				Department	
Address			Shipping Address			
City, State	ZIP	City, State			ZIP	
Male/Female		Ski/Board		Skill Level		
Age/Date of Birth		Style				
Item	Description	New/Used	Size	Quantity	Price	Subtotal
Item Total						
Tax						
Total Due						Method of Payment

Sales

The sales form shown in Figure 1.3 is fairly standard. All of the hard work in terms of configuration was done by the salesperson. In some cases, the salesperson might ask the customer to initial some items that might present compatibility issues to make sure the customer is aware of the potential problems.

Returns are usually accepted on most items as long as they have not been used outside (e.g., scratched or worn boots cannot be returned). The description generally includes the manufacturer's name and style. The SKU (stock keeping unit) is a special number created within the store to code each item. It is important for salespeople to identify the type of boarding/skiing and the skill level. This information is used to send customers mailings about special sales. The owner also has started thinking about keeping customer sizes in a database. This information would be particularly helpful in clearing out last year's inventory of special sizes (very small or very large), because it would help pinpoint customers who could use those special sizes. The catch is that the store owner is concerned about consumer privacy and fears that customers may not want to have their sizes on file at the store. If a customer has already purchased items in a specific category and size, that data will be available. The difficulty lies in having salespeople ask customers their sizes when they are not purchasing these products. For instance, it might appear rude to ask a customer who only came in to buy ski wax for his or her jacket size.

The store evaluates salespeople by the amount of sales they make, so it is important to track sales by each employee. Of course, the database should contain additional information about each employee, such as his or her phone number, address, and primary department assignment. Since clerks rarely write down the department names properly, it makes sense to have a separate lookup table for the department names.

Figure 1.4

Rentals						
Customer First Name		Last Name		Rental Date		
Phone		E-Mail		Expected Return		
Address City, State ZIP			Shipping Address City, State ZIP			
Male/Female		Ski/Board		Skill Level		
Age/Date of Birth		Style				
Item	Description	Size	Fee	Return Date	Condition	Charges
Item Total						
Tax						
Total Due		Added Charges				
Method of Payment		Signature				

Also, note that some of the best customers participate in several styles, even crossing between skis and boards. A customer who is an expert at downhill skiing, however, might be a beginner with snowboards.

Rentals

The form to handle rentals is similar to the sales form. But notice in Figure 1.4 that columns have been added for return date, condition, and additional charges. The additional charges are imposed if an item is returned late or if it is damaged. Additionally, customers are required to sign the form to indicate their agreement with the skill level, rental conditions, and the release printed on the back of the form. Katy, the current manager, has talked about capturing the customer signatures digitally and storing them online, but it is not a high priority.

Observe that the current form requires that each rented item be checked off separately when it is returned. Although store clerks often complain about having to mark each row separately, about 20 percent of the time, a customer forgets to return an item and has to bring it back later.

Renting ski equipment also raises the issue of reservations. On some holidays, all of the equipment is rented out before 10:00 A.M. Some long-term customers have suggested they would like to be able to reserve equipment. Currently, the rental managers sometimes set aside equipment if a valuable repeat customer calls in advance. This process works reasonably well, but the managers have talked about creating a system that is available to everyone. One of the drawbacks is that they are concerned that the general public might reserve items and then never show up, leaving equipment unused that could be rented to someone else.

Lab Exercise

The first step in any project is to identify some basic elements of the system. What are the goals? What is the scope? What tools will be needed? What are the benefits? What are the expected costs? How much development time

will be needed? All of these questions are difficult to answer and rarely have a single value. Instead, you need to create a project plan. The plan will include a feasibility statement that describes the basic costs and potential benefits. As a real-world project, the plan would also include a list of developers and a statement of expected fees, so the owners can evaluate the decision to hire you.

Action

Find information about skis and snowboards on the Internet.

If necessary, install and upgrade the DBMS.

Figure 1.5

Project Title: *Sales System for Boards and Skis*
Customer: *All Powder Board and Ski Shop*
Primary Contact: **Katy**
Goals:
Project Description:
Primary Forms:
Primary Reports:
Lead Developer:
Estimated Development Time:
Estimated Development Cost:
Date Prepared:

Project Outline

As a first step in developing the project plan, you need to summarize the overall project. This summary should contain a brief description of the project, its goals, and initial lists of primary forms and reports. Ultimately, this summary would also include the scope and anticipated budget for the project.



Activity: Review the Case and Research the Industry

For the purposes of this lab, you will prepare a project proposal for developing the sales system needed by the All Powder Board and Ski Shop. The rental component will be left for another exercise.

You should begin by reviewing the description of the company. You should also use the Internet to check out some of the manufacturers and some of the competitors. You need to be sure that you understand the key factors in the industry. Figure 1.5 provides a possible structure for your summary. You should review the case and enter the basic information requested.

Action

Fill in the project milestone dates based on your school calendar.

Project Plan

The project plan consists of a detailed breakdown of the steps needed to create the final system. A common approach is to follow the steps of the systems development life-cycle methodology: Initiation, Analysis, Design, Implementation, and Review. Some organizations have rigid descriptions of each of the steps involved in this process. Some organizations adopt a more flexible approach. Either way, this plan should outline the basic steps that need to be completed and an estimated schedule.

In the initial phase, it is also helpful to identify any potential risks to the project development. At various stages, what might go wrong? If you are aware of the potential problems, managers can monitor for them and can prepare solutions more quickly.

Figure 1.6

1. Define the project and obtain approval.
2. Analyze the user needs and identify all forms and reports.
3. System Design
 - a. Determine the tables and relationships needed.
 - b. Create the tables and load basic data.
 - c. Create queries needed for forms and reports.
 - d. Build forms and reports.
 - e. Create transaction elements.
 - f. Define security and access controls.
4. Additional Features
 - a. Create data warehouse to analyze data as needed.
 - b. Handle distributed database elements as needed.
5. System Implementation
 - a. Convert and load data.
 - b. Train users.
 - c. Load testing.
6. System review



Activity: Create the Initial Project Plan

Project plans and schedules are often shown with Gantt charts to illustrate how the various steps depend on each other. If you have access to software such as Microsoft Project, it is relatively easy to create the project plan. Figure 1.6 shows the basic steps that the labs will follow in building the application. Ultimately, you would estimate the times required for each step. However, until you have read the rest of the book and worked with the databases, it is difficult to estimate the times needed for each step. For now, evaluate the steps and try to identify any dependencies between the tasks. For example, is it possible to create the forms without having the database tables and relationships? Assuming you

Action
Create the feasibility plan for the project.

Figure 1.7

Assumptions			
Annual discount rate	0.03		
Project life/years	5		
Costs		Present Value	Subtotal
One time			
DBMS software			
Hardware			
Development			
Data entry			
Training			
Ongoing			
Personnel			
Upgrades/annual			
Supplies			
Support			
Maintenance			
Benefits			
Cost Savings			
Better inventory control			
Fewer clerks			
Strategic			
Increased sales			
Other?			
Net Present Value			

have several people to help, reorganize the tasks so that as many tasks as possible can be done at the same time.

Feasibility

Feasibility studies are notoriously difficult. The concept is certainly simple: Identify the potential costs and potential benefits of a system and compare them. The problem is that benefits might not be quantifiable, so it is difficult to attach meaningful numbers. Nonetheless, it is useful to at least write down the anticipated costs and expected benefits. Even if numbers are not available, at least managers can see a concise statement of the analysis.



Activity: Create the Feasibility Analysis

Figure 1.7 shows the basic elements of a feasibility study. You need to create a spreadsheet with these main categories. Use research to identify approximate costs of the various components. For example, assume that the shop will need to purchase a server to host the main database and two client computers for the sales staff. With Oracle, several configurations are possible. Examine the software license to determine the number of copies you will need and the approximate cost. Other numbers, including benefits can be estimated. Remember that annual costs and benefits should be discounted to compensate for the time value of money. Use the present value (PV) function in Excel. Although the benefits are relatively well-defined, they can still be difficult to estimate. For example, how will the system reduce the need for sales clerks? How many or how many hours? How much do clerks earn? Likewise, in terms of inventory control, how much money will be saved by not having to slash prices at the end of the season to clear the unsold inventory? You need to know or estimate the number and value of items typically left at the end of the season. In practice, the managers might have answers to some of these questions, but you will still have to do additional research. In this example, be sure that you spell out your assumptions.

The Database Management System



Activity: Explore the DBMS

Although Oracle is one of the most popular database systems, it can be somewhat difficult to install and maintain. Generally, the Oracle DBMS is installed on a server and the developer software is installed on the individual workstations. In a production environment, separate servers are often used for the DBMS and the Web or Application server. However, it is possible to install all of the current Oracle components onto a single computer. If you are working in a classroom lab, an Oracle server should already be configured, and your machines should have the Oracle client software installed and tested. If you are working on your own

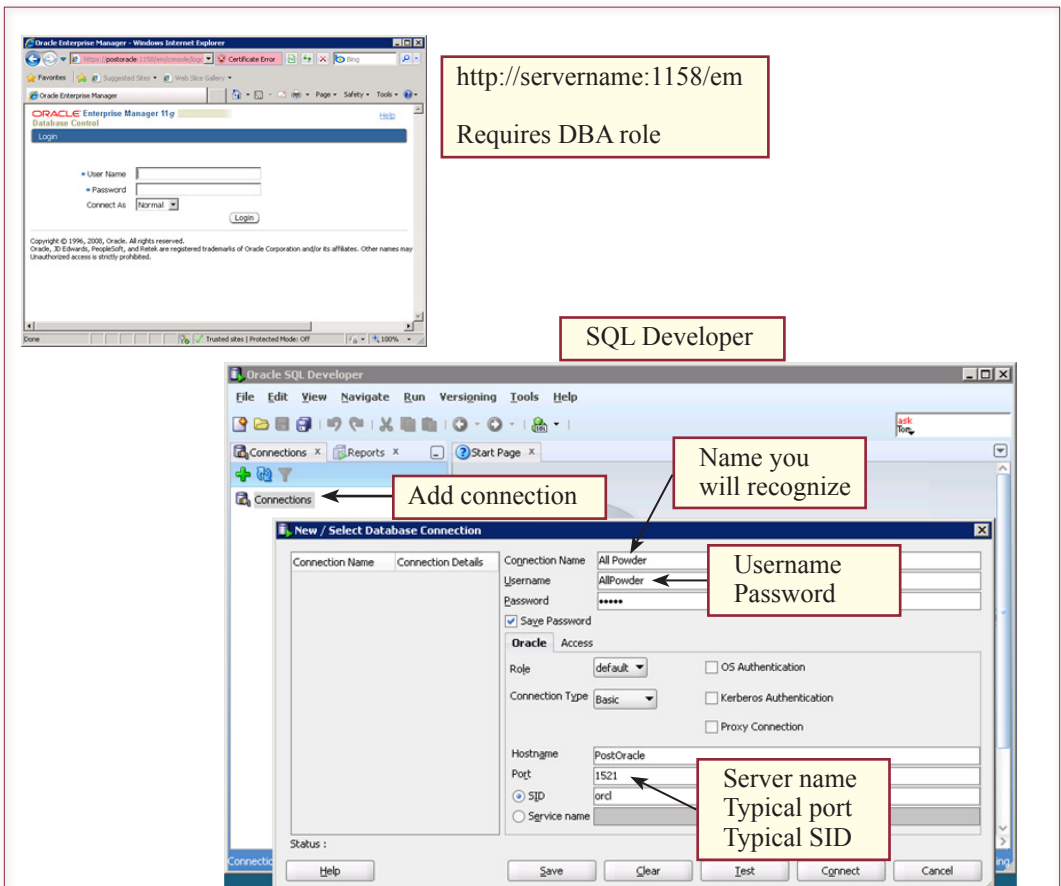
Figure 1.8

```
https://academy.oracle.com
  Oracle academic initiative--software
http://www.oracle.com/technology
  Oracle technology network--downloads
http://www.oracle.com/technology/documentation
  Oracle documentation—requires OTN
```

computer, Figure 1.8 shows where you can obtain the Oracle software through two main programs: the academic initiative and the technology network. If your school does not participate in the OAI program, the technology network enables software developers to download the most recent software and the documentation, but it requires a relatively fast Internet connection. Most of the labs in this book can be completed using the enterprise 11g database, the SQL Developer, and the JDeveloper tools. If you install Oracle yourself, be sure to follow the installation instructions from Oracle completely. In particular, if you use DHCP to obtain your network address (just about everyone), you need to install the Microsoft Loopback Adapter. The process is spelled out in the Oracle Windows Installation Guide. The Loopback Adapter can be installed from the Control Panel/Device Manager. Right-click the computer name and choose the option to add legacy hardware. Do a manual search, choose Network adapters, then Microsoft and pick the Microsoft Loopback Adapter. It should not disrupt your normal network connection. Follow the Oracle installation process and accept most of the default values unless you need to store data in different locations on your computer. Write down any changes you make and write down any messages you see.

If you install your own copy of Oracle, be sure to write down the SID, and port numbers used—particularly the http URLs created for the administration tools. You should also write down the password created for the sys account. Log into

Figure 1.9



that account as sysdba and create a new user account for yourself. Assign the DBA role to that account and write down the username and password. Use the new DBA account for the exercises in this book. Do not use the sys or sysdba accounts for development.

To access an Oracle database, the machine you are using must have an Oracle network configuration that describes how to reach the database. This configuration file defines a name for each Oracle database that it knows how to reach. To connect to an Oracle database, you need the database name, a user name, and a password. This chapter assumes that you have already installed Oracle and have an account to use.

Data in Oracle is stored in tables, and these tables are collected into a schema. A schema is simply an organization structure assigned to each user. With a schema, your work stays separate from tables created by other users.

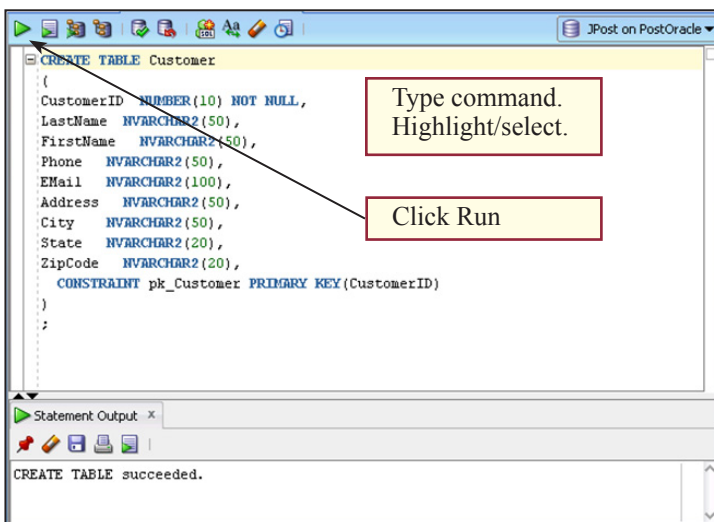
Not counting forms and reports and a couple of other tricks, you have three basic ways to connect to an Oracle database: (1) SQL Plus, (2) Enterprise Manager (admin), and (3) SQL Developer. SQL Plus runs as a command-line program on the server and you are not likely to use it anymore. The Enterprise Manager requires at least DBA permissions and is used to monitor and configure the database. It provides a more graphically-oriented approach to several administrative tasks, and provides assistance in creating common SQL commands. If you are running your own copy of Oracle, you can play with the Enterprise Manager. If you are using a shared copy of Oracle, you probably do not have DBA privileges, so you will not be able to use it. It is examined in more detail in Chapter 10.

That leaves SQL Developer—a relatively new tool. It creates SQL Worksheets to enter and run SQL commands on the server. This tool can be installed on client computers that can connect across the network—it will not connect across a

Action

Start SQL Developer.
Connect to the database.
Enter the CREATE TABLE commands.
You can use the designer but need the Advanced options to set NVARCHAR2.
Click the Run button.
Type DESCRIBE Customer to ensure that the table was created correctly.

Figure 1.10



standard Internet connection. (In a production environment, you might set up a virtual private network or VPN.) But Oracle does not provide a Web-based development or SQL tool.

The process of connecting to an Oracle server depends on which of the tools you will use. Figure 1.9 shows that the startup screens for the EM and SQL Developer. The EM requires a specific URL including the port number. The value used here is the standard default value (1158). It is possible to change these values when the products are installed, but you should keep these values to reduce confusion.

Create a Table

SQL Developer can be started from your computer's main menu. It does not ask for a username or password when it starts because it can be connected to multiple data sources at the same time—including different versions of Oracle, or even Microsoft SQL Server and Access databases. When you add a Connection, you will be asked to specify the name of the database server and its SID along with your username and password.

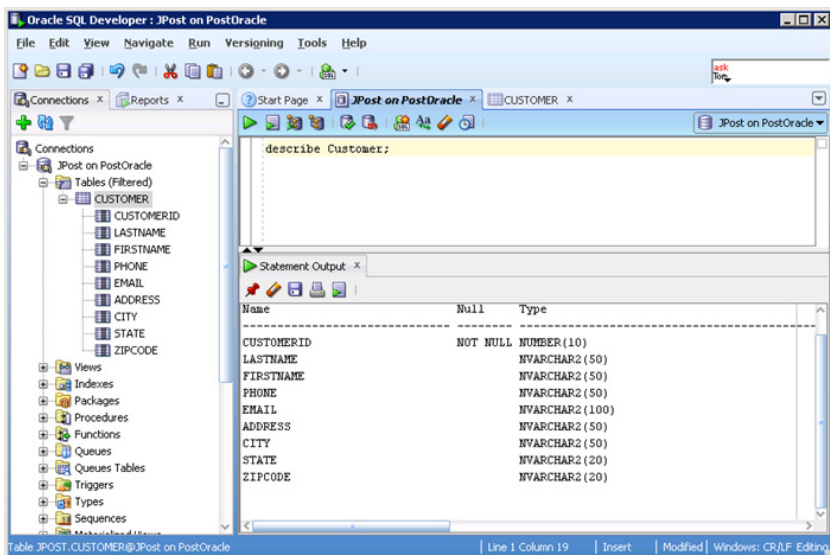
To get a quick perspective of the various components of the DBMS, you need to build a simple database. One of the first things you have to understand with Oracle is that it is heavily reliant on SQL—which will be covered in greater detail in later chapters. Most administrators perform all tasks by writing SQL statements. If you have DBA privileges, you can use the Enterprise Manager to perform many tasks. However, most real Oracle DBAs use SQL, so you really should work with SQL for now. For the most part, this book will use plain SQL statements.

To illustrate the process of creating a database, you need to start SQL Developer. The first step is to create a small table. This table will hold basic customer data, so it needs columns for CustomerID, LastName, FirstName, Phone, Email,

Action

Insert three lines of data into the table using SQL Worksheet.
Copy the resulting INSERT commands and save them in your lab notebook.

Figure 1.11



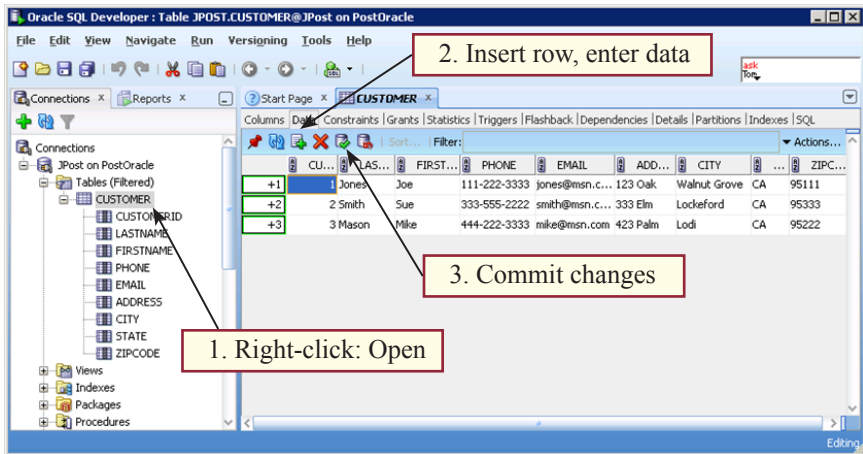


Figure 1.12

Address, City, State, and ZipCode. Figure 1.10 shows the CREATE TABLE command you should enter to build the table. The spacing and letter case are not important, but are used to make the command easier to read. Unless you put quotes (“) around the table and column names, Oracle will convert all names to upper case. The semi-colon at the end is critical—it tells Oracle you are finished typing the command. If you forget the semicolon and just press the Enter key, you will simply get new blank lines. That is OK, just put the semi-colon in the current line.

SQL Developer does have a visual method for creating tables, but the current version has some limitations. You can right-click the Tables entry in the navigator and choose New Table option. The tool enables you to enter column names and choose data length in a grid. However, the base grid supports only a limited set of data choices; that do not include the NVARCHAR2 options. But, NVARCHAR2 is used to store Unicode characters, so the default data type is too limited. Setting the Advanced checkbox opens a more detailed editor, where additional options including NVARCHAR2 can be specified for each column. It is also possible to see the SQL statement. Once you learn the main data types, it is usually easier to just type the CREATE TABLE command.

Figure 1.13

```
INSERT INTO CUSTOMER(CUSTOMERID, LASTNAME, FIRSTNAME, PHONE, EMAIL,
ADDRESS, CITY, STATE, ZIPCODE)
VALUES (1, 'Jones', 'Joe', '111-222-3333', 'jones@msn.com', '123 Oak', 'Walnut Grove', 'CA',
'95111');

INSERT INTO CUSTOMER(CUSTOMERID, LASTNAME, FIRSTNAME, PHONE, EMAIL,
ADDRESS, CITY, STATE, ZIPCODE)
VALUES (2, 'Smith', 'Sue', '333-555-2222', 'smith@msn.com', '333 Elm', 'Lockeford', 'CA',
'95333')

INSERT INTO CUSTOMER(CUSTOMERID, LASTNAME, FIRSTNAME, PHONE, EMAIL,
ADDRESS, CITY, STATE, ZIPCODE)
VALUES (3, 'Mason', 'Mike', '444-222-3333', 'mike@msn.com', '423 Palm', 'Lodi', 'CA', '95222')
commit;
```

In any case, you should copy the SQL command and save it in your notes. You will encounter many circumstances where you need to recreate a table or alter it slightly. It is much easier to do when you can edit the SQL statement and rerun it.

Figure 1.11 shows a useful trick with Oracle SQL. The DESCRIBE command displays basic information about tables and other objects. Simply type: DESCRIBE Customer (you can use all lower case letters). With SQL Developer, you can also expand the list of tables and see the column details of the specific table in the navigator list.

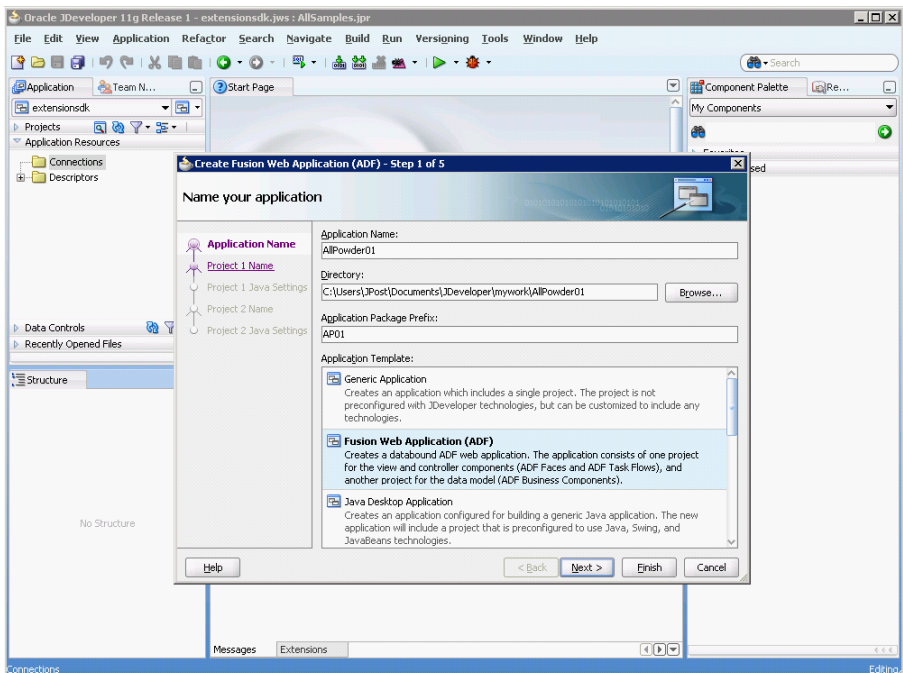
The next step is to enter some data for fake customers. You can use the SQL Developer Worksheet. As shown in Figure 1.12, right-click the table and choose Open. Click the icon to Insert Row. Enter the data in the worksheet. After entering two or three rows, click the Commit changes icon. This action will save the data and also show you the SQL statements used to insert the rows. Keep in mind that everything you do in the designer is converted to SQL so that it can be submitted to the DBMS for processing. It is often useful to copy and save the SQL statements in an electronic notebook in case you have to run them again in the future. Figure 1.13 shows the SQL INSERT commands. These commands are explored in more detail in Chapters 4 and 5.

Go back and look at the CREATE TABLE command and you will see that each column except CustomerID was defined as NVARCHAR2, which means it is a text column. Text data must be surrounded by quotation marks. If you are curious, the “N” in NVARCHAR2 stands for “National,” and means that you can enter

Action

In JDeveloper, select Application/New. Name it AllPowder01. Select Template: Fusion Web Application (ADF). Add the ADF Faces components.

Figure 1.14



Unicode data. For example, you could enter special language characters for various languages, including Chinese and Japanese ideograms. To enter national language characters, you might have to install the appropriate editor (IME). When using INSERT commands, you should also use a special notation to indicate the special characters. For example, N'María' where the quotation mark is preceded by the letter N.

Finally, note the use of the COMMIT command at the end. Whenever you make changes to data tables in Oracle, you eventually need to write a COMMIT statement. It tells the DBMS that you are finished making change and they should all be written to the tables. In this example, you could have typed COMMIT after each INSERT command, but it is time-consuming to type it that often when you are entering data by hand.

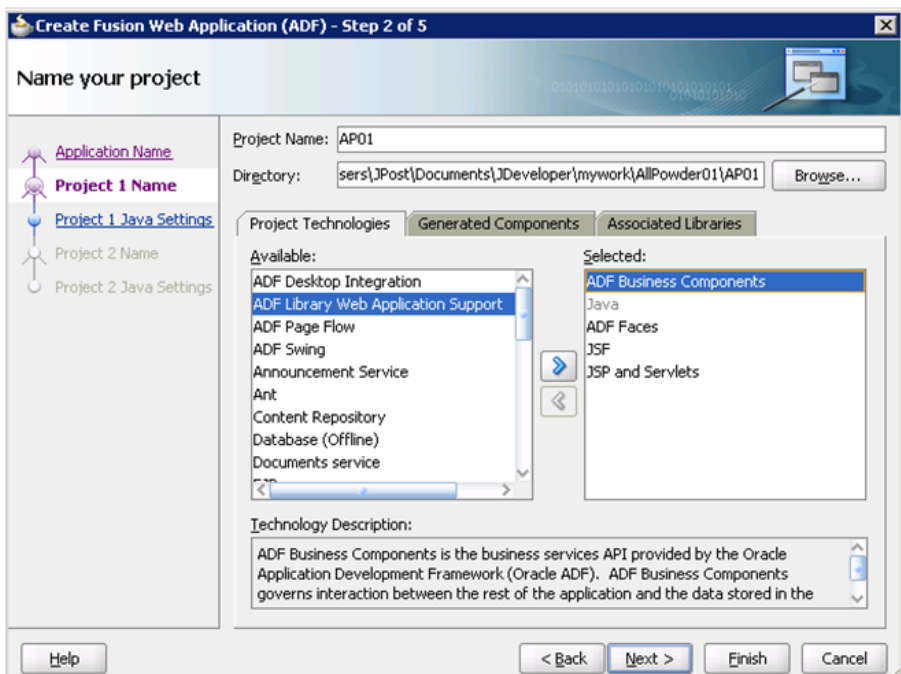
Action

Open Step 2: Connect to a database.
 Create a Database connection (button).
 Enter the database name and password.
 Verify the server name and SID.
 Click the Test Connection button.

Create a Form

You probably noticed that the INSERT command is a relatively painful method of entering data into tables. You certainly cannot expect clerks to enter data this way. Even entering data through the table editor is risky. In practice, you will rarely enter data directly into tables. Instead, you will build forms that users can run to enter and edit data. Oracle provides several tools to help build forms and reports. This workbook focuses on the J2EE (Java) Web forms using the application development framework (ADF).

Figure 1.15



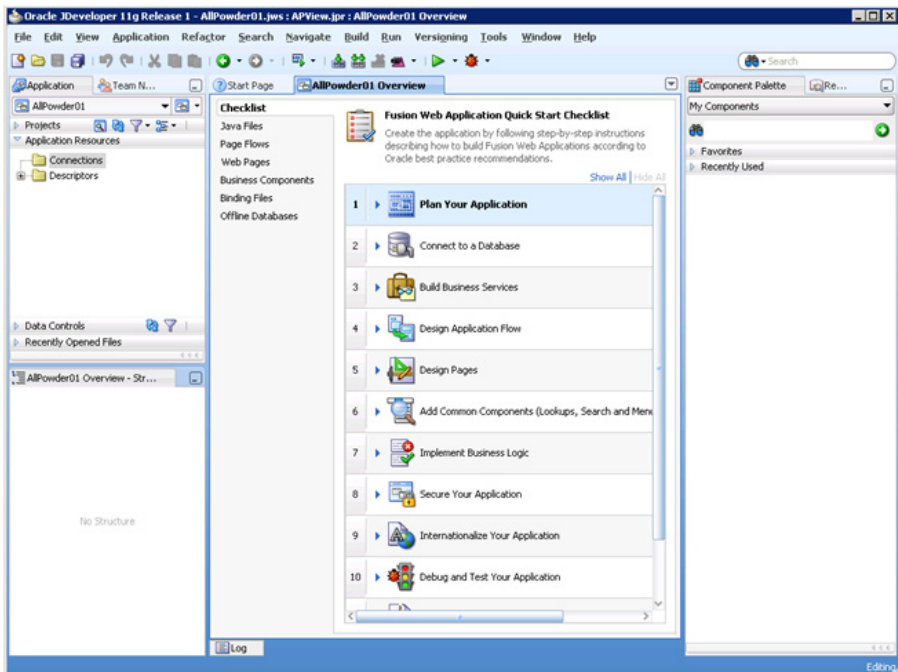
You need JDeveloper installed on your computer to use it to create new forms and reports. JDeveloper is an integrated development environment (IDE) that contains wizards and other tools that help create forms and generate code to build applications. It can be downloaded from the Oracle Technology network. Installation consists of unzipping the files into a folder on your computer. Remember the folder location. Find the main file (folder/jdev/bin/jdeveloper.exe), right-click it and choose the option to create a shortcut on your desktop to make it easier to start. Also, the first time you run JDeveloper, you will be asked to locate the main Java.exe program from the JDK (Java Development Kit) folder. JDeveloper installs a version of the JDK into the Oracle folder, for example: C:\Oracle\Middleware\jdk160_14_R27.6.5-32\bin\java.exe. Alternatively, you can download a newer version of the JDK from Sun (www.sun.com/java).

Once JDeveloper is installed, the built-in wizards help you create forms and reports. When the system starts, accept the default option to include all wizards. The process begins by creating an application. Ultimately, applications contain many forms and reports, as well as business logic to connect the forms and take various actions based on user input. For now, you will create a simple form to edit data in the Customer table.

Action

Open Step 3: Build Business Services.
 Open substeps.
 Step 3.1: Add Entity
 Choose the Model object.
 At Entity Objects: Click Query button.
 Select the Customer table.
 Add it to the selected Updatable Views.
 Skip the Read-only View objects.
 Stick with default choices.

Figure 1.16



In JDeveloper, select Application/New and name it AllPowder01 or something similar that you will recognize later. For the most part, you can use the default values for the directory and package prefix. If you are running on a shared lab computer, you might have to select a different directory to hold your files. The most important step is to choose the Fusion Web Application (ADF) template—which loads some fundamental objects to help create and run forms. Figure 1.14 shows the selection of the Fusion ADF.

Action

- Return to the Checklist.
- Open Step 5 and the substeps.
- Open Step 5.2: Create Pages.
- Create a JSF Page.
- Project: ViewController.
- Name: Customer.jspx
- Quick Start Layout: One Column Header.
- Use the defaults to finish.

Oracle has created several tools that add more features to Web pages. These components are grouped into the ADF Faces collection. They might not be required for this simple form, but you will want to add them for most applications. Figure 1.15 shows the setup where you find the ADF Faces and transfer them to the right side.

Once the application is created, you will see a checklist of steps. Only a couple of the steps are needed for this simple project. If you look through the list and expand a couple of steps, you will see that many have multiple substeps. You should also note that the steps have links to help and additional details. Figure 1.16 shows the start of the list.

One of the most important steps is to connect to the database—Step number two on the list of tasks. Select Step 2 on the checklist and click the button to Create a Database Connection. Figure 1.17 shows the basic form. Assign a name

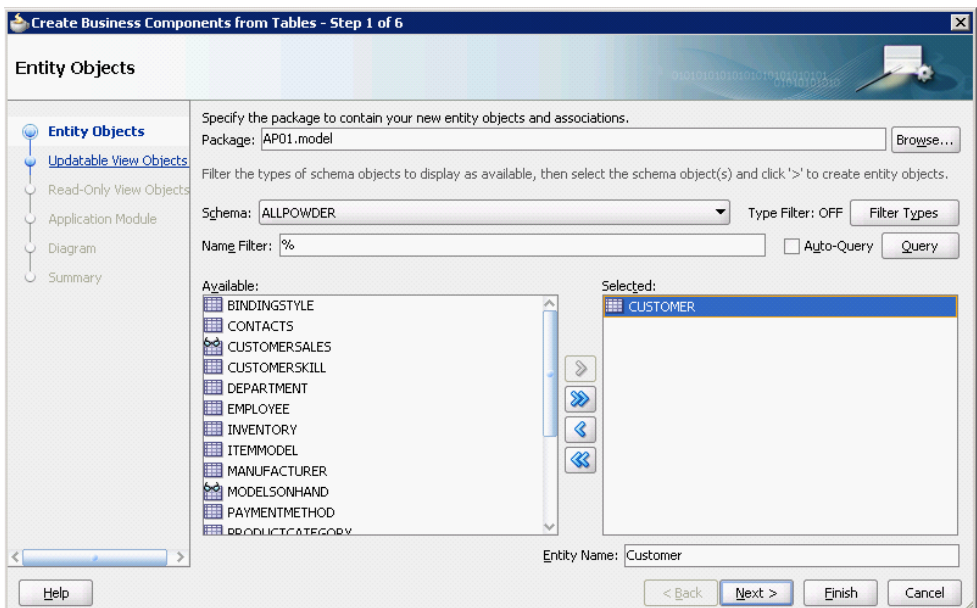
Figure 1.17

to the connection (such as AllPowder or Chapter01). Enter the username you were provided or one that you created to use as a developer. Enter the associated password. For now, ensure that the Save Password box is checked so you can avoid having to log in constantly. Check the server hostname. If JDeveloper is running on the same computer as the Oracle database, you can use the computer's name or just localhost. Enter the SID—which needs to match the SID value entered when the database was created. Always run the Test Connection button to ensure you entered the data correctly. If the connection fails, double check all of the data. When finished, click the OK button to close the editor.

The next main step is to define an entity in the Model that will hold the data. Oracle ADF uses an architecture known as Model-View-Controller (MVC). The Model holds the data and interrelationships. It consists of objects that know how to retrieve and store data in the database tables. The View holds objects to display pages. The pages hold objects that exchange data with the objects in the Model. The Controller (or Services) holds objects that contain logic to evaluate the data and transfer from one page to another. One Model object (Customer) and one View object (display page) are needed for this simple test project.

Open Step 3 (Build Business Services) and click the link to open the substeps. Choose Step 3.1 (Add Entity), then select the Model project to hold the entities. Figure 1.18 shows the form used to select the tables. Click the Query button to get a list of all tables in your database schema. Your database probably holds only the Customer table you created, unless you are using the full All Powder database.

Figure 1.18



Action

- At the top of the page, type Customer.
- Select/highlight it and choose the Heading 1 style from the dropdown list.
- Expand Data Controls navigator (on the left-side).
- Drag CustomerView1 onto the lower page section.
- Choose Form/ADF Form.
- Check boxes for Navigation and Submit buttons.

Select the Customer table in the list on the left side. Click the single arrow to move it to the right side list of selected tables.

Click the Next button to move to the next screen. The goal is to create a page that edits the Customer data, and this page enables you to create updatable views. Select the Customer table and move it to the right-side list. Click the Next button to move to the next set of choices. This set of choices is similar, but it refers to Read-only views. These tables are typically used for lookup lists. None are needed at this time, so click the Next button. Continue through the wizard, leaving the default values, until the wizard finishes. Click the link near the top to return to the main checklist.

The next major step is to create a page to display the data. Open Step 5 and the substeps. Open Step 5.2 (Create Pages). Choose the option to create a JSF page, and store it in the ViewController project. Name it Customer.jspx, which creates the page in XML format and supports additional options later.

Every page can be assigned a layout—in fact, you can create a special template that can be used as the foundation for all pages to ensure a consistent look across a project. For now, as shown in Figure 1.19, simply choose the Quick Start Layout and pick the One Column which is a single-column page with a simple header.

The main objective is to put form controls on the lower section of the page that display the columns (name, phone, and so on) so they can be edited by the users. Look at the JDeveloper layout and find the Data Controls window—probably on the middle-left side by default. Expand the section and the entries within it. Find the CustomerView1 object and drag it onto the form. Figure 1.20 shows the

Action

Right-click the form, choose Run.

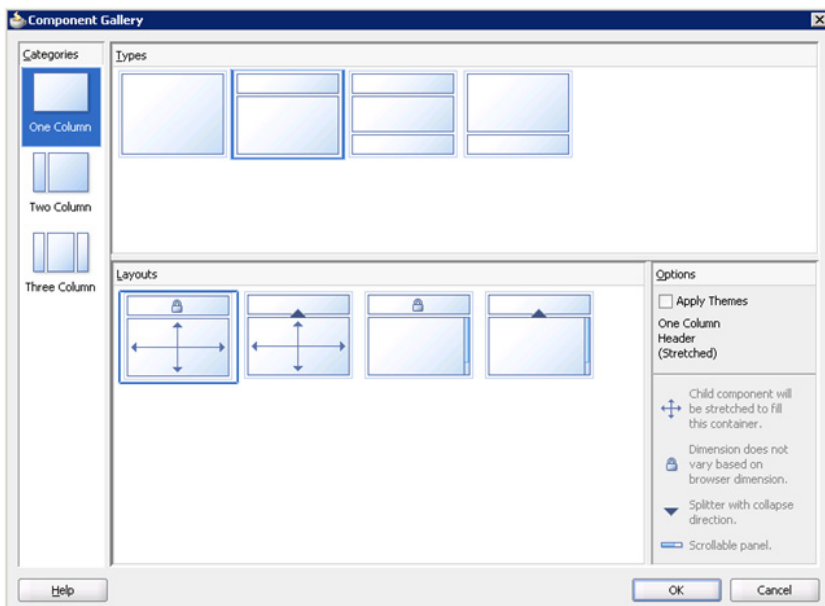
Wait for the server to start.

If on a server, best to add the server to the list of Trusted Sites in Options/Security.

Probably want to resize the E-mail input box in Properties: Appearance: Columns.

Use bindings.Lastname... instead of bindings.Email...

Figure 1.19



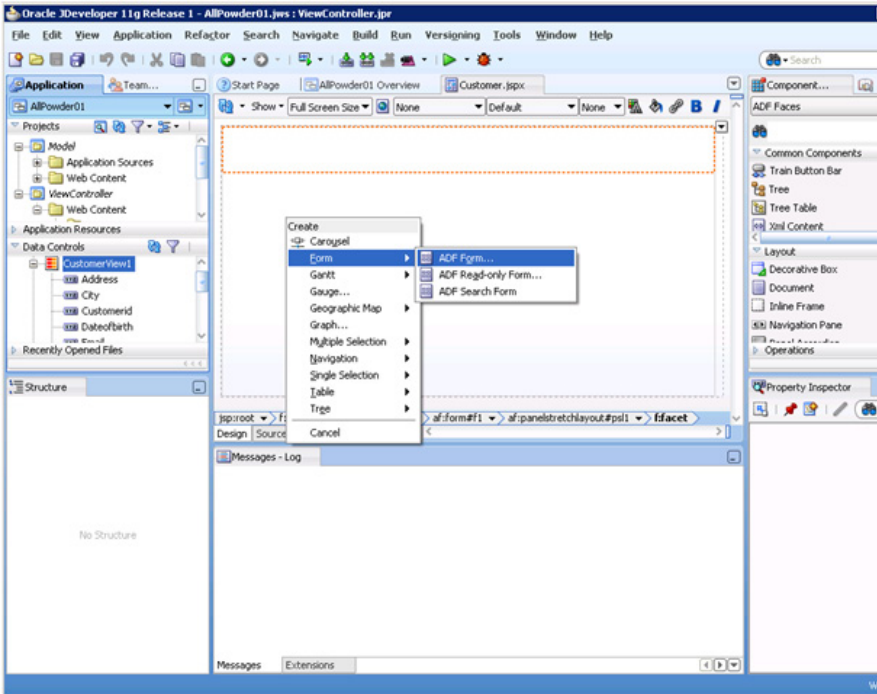
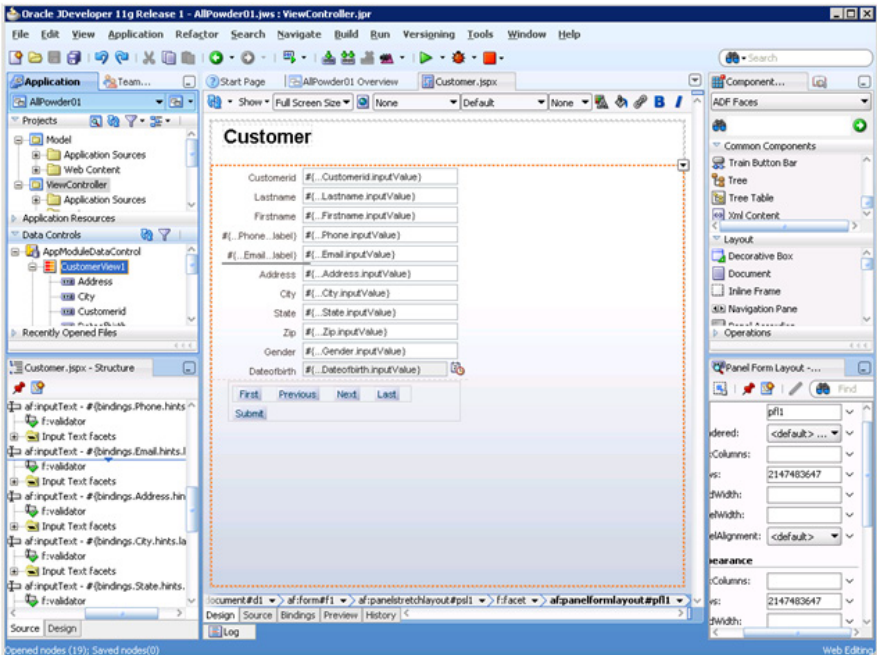


Figure 1.20

resulting pop-up menu. Choose the Form/ADF Form menu options, which will automatically create controls for all of the columns on the form—including the labels. Before closing the fields page, click the two check boxes at the bottom left: Include Navigation Controls and Include Submit button.

Figure 1.21



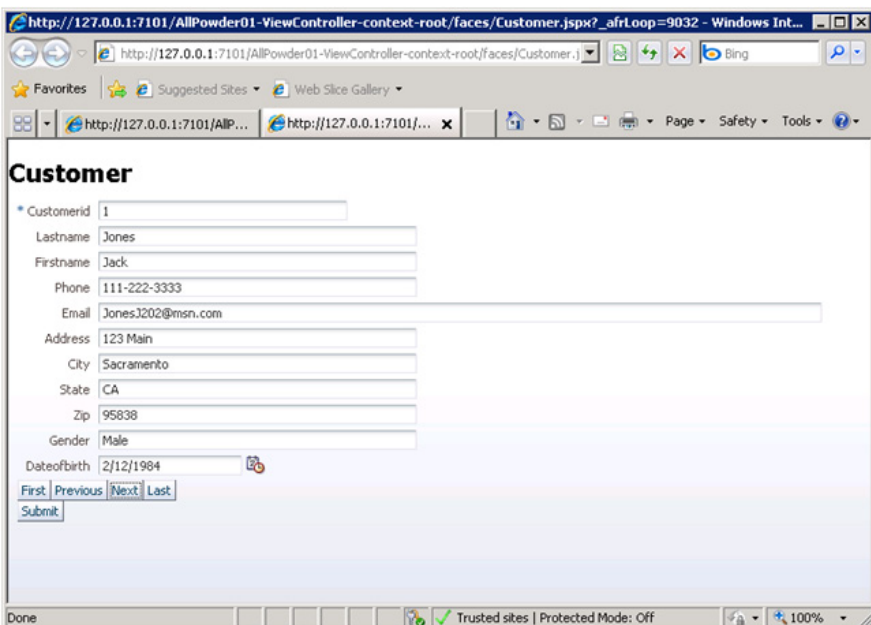
Add a title to the top of the form by clicking the top/header section. Type Customer as the header. Because the page is largely based on HTML, you can add styles. Select the word Customer and choose Heading 1 from the style drop-down list. You can also add a cascading style sheet (CSS) and define styles for any controls on the form. Figure 1.21 shows the design layout of the page.

You are almost finished. Right-click the form and choose the option to Run it. You will probably have to wait for the application to compile and the application server to run. Note that if you are running the application on a server, you will probably have to add the server to the browser's list of trusted sites. In Microsoft Internet Explorer, use the menu Internet Options/Security/Trusted Sites.

Figure 1.22 shows the actual form. Use the navigation buttons to check out some of the data rows. You can edit some of the data and click the Submit button to save the changes to the database.

Your initial version of the form probably has a relatively large text box for the e-mail entry because that column was defined with more characters than the other columns. An easy way to reduce the width is to change its properties. You can use the red square on the JDeveloper menu to stop the form browser. Right-click the E-mail box on the form and choose the Properties option. The properties browser is typically displayed at the bottom right corner of JDeveloper. Expand the properties for Appearance until you find the Columns entry. Change the entry to use bindings.Lastname... instead of bindings.Email... This change will set the width of the text box to match that used for the LastName column instead of the Email column. You could also replace the entire entry with a fixed number. Re-run the form. When you are finished, save everything. You can close JDeveloper and accept the options to stop the Web server. If you look at the source for the form in the browser, you will see that Oracle created relatively standard HTML and Javascript code to build the page. Once an application is built and deployed to a production server, almost any browser can access and display the page. Of course, securing the application and building an application process require additional work.

Figure 1.22



The screenshot shows a web browser window with the following details:

- Address bar: `http://127.0.0.1:7101/AllPowder01-ViewController-context-root/faces/Customer.jspx?_afLoop=9032 - Windows Int...`
- Page Title: **Customer**
- Form Fields:
 - Customerid: 1
 - Lastname: Jones
 - Firstname: Jack
 - Phone: 111-222-3333
 - Email: JonesJ202@msn.com
 - Address: 123 Main
 - City: Sacramento
 - State: CA
 - Zip: 95838
 - Gender: Male
 - Dateofbirth: 2/12/1984
- Navigation: [First](#) | [Previous](#) | [Next](#) | [Last](#)
- Submit:
- Status Bar: Done, Trusted sites | Protected Mode: Off, 100%

Because of the large number of options, it takes many steps to create a form. In the end, the process is not too complex, but it is somewhat confusing the first few times through. This example is only a small step, but it begins to illustrate the power of building Web forms.

Exercises



Many Charms

Madison and Samantha, friends of yours, have a small business selling charms for bracelets and necklaces. They buy some of the charms they sell; others they make. So far, they have run the business as a hobby, selling primarily to friends and relatives. But they have recently established a website to display pictures and prices of some of the charms. You have agreed to build a database for them to track their inventory, customers, and sales. Any orders they receive through the website will be e-mailed, so the website does not have to be connected live to the database. The database is a relatively traditional sales system, but it is slightly complicated by the nature of the charms. Charms come in a variety of shapes, sizes, and materials. For example, customers who want a quarter-moon charm have a choice of 4 mm or 8 mm; and of silver, gold, gold plate, bronze, or painted ceramic. Charms are also offered in categories such as animals, hearts, birthdays, and so on. Additionally, the duo offers a variety of chains and pins to hold the charms. Eventually, they want to track the sales by all of these categories, so they will know which items are selling the best and which make the most profit. Costs and prices tend to fluctuate. If they purchase items in large bulk, the per-piece cost is lower, but they need to know they can sell the entire shipment. If an item sits around too long, they find that they have to significantly cut the price just to clear out the stock. Of course, gold items are more expensive, making them more difficult to sell, and they are reluctant to tie up their money in high-priced merchandise.

1. Research similar sites on the Internet. Describe or sketch the major forms and reports that the company might use.
2. Create the initial proposal and feasibility study.



Standup Foods

Laura runs a catering company that focuses on Hollywood movie studios. Her chefs prepare hors d'oeuvres, sandwiches, and other food items that are served to the cast and crew of various movies and studios. To be fresh, the food is prepared each day in the main kitchens, and meals are then assembled and displayed on-site. For some clients, the company vans deliver fresh food every few hours. To hold costs down, many of Laura's employees are part time—only a few chefs and managers are full-time employees. Some of Laura's clients call at the last minute, so she maintains a large list of potential workers who can perform a variety of tasks, from driving to food preparation and display, as well as cleanup. The chefs and managers evaluate workers after each job in terms of timeliness, appearance, friendliness, and the ability to take orders and accomplish tasks. Workers often perform many tasks at a given event. For instance, a driver might also be a server. But some tasks require specific certifications. Not all workers are licensed

to drive, and only a few have been trained to perform some tasks such as cutting meats. Most of the employee ratings are somewhat informal at the moment, but she would like to computerize them to help her select the best workers for future jobs. At some point, she would like to offer bonuses or higher pay to workers who routinely perform well. Another challenge Laura faces is that some clients are finicky about certain types of food. In particular, some movie clients have special preferences as well as some items that cause allergic reactions. The chefs currently keep these two lists in paper folders for some major performers and actors. But to be safe, Laura wants to computerize the lists and, ultimately, the recipe ingredients. Then when a chef plans the meals, the computer could check the list of main guests and their allergies against the recipe list to identify potential problems.

1. Research similar sites on the Internet. Describe or sketch the major forms and reports that the company might use.
2. Create the initial proposal and feasibility study.



EnviroSpeed

Brennan and Tyler are owner/managers of a consulting firm that specializes in environmental issues. In particular, the company's scientists are experts in clean-ups for chemical spills. For example, if a tanker crashes and spills chemicals on a highway, the company can quickly evaluate the potential problems and identify the best method to clean up the spill and prevent problems. The company itself does not clean up the spill, but it has contacts with several crews around the globe that it can call if local emergency workers need additional help. The primary focus of the company is to provide expert knowledge in the time of a crisis. This task requires specialized scientists, good communication systems, and in-depth training and practice. Brennan wants to improve the existing information system to maintain a database of case histories. Then, if a similar problem arises in the future, the scientists can quickly search the database and identify secondary problems to examine which solutions and ideas were successful and which ones caused more problems. Tyler has explained that at a minimum, the database has to hold the contact information for all of the scientists and emergency crews. It must also list the specialties, training, and skill levels of each person in a variety of areas. In terms of actual situations, the database should track the identities and roles of the various people and the key time frames (when reported, response time, and so on). Scientists also need the ability to list all of the chemicals involved and details about the terrain (hills, water, soil composition). More subjective data must also be captured, including comments by the onsite team and a description of the problem and secondary factors. All proposed solutions should be entered into the database, along with comments regarding their strengths and weaknesses as well as the final selections and an evaluation of the result. It is important to track potential solutions that were discarded. Even if they did not apply to the original problem, they might be useful for a future event with different circumstances.

1. Research similar sites on the Internet. Describe or sketch the major forms and reports that the company might use.
2. Create the initial proposal and feasibility study.

Final Project

The main textbook has an appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, do the following.

1. Research similar sites on the Internet. List the major forms and reports that the company might use.
2. Create the initial proposal and feasibility study.

Database Design

Chapter Outline

Database Design, 26

Oracle Data Types, 26

Case: All Powder Board and Ski Shop, 28

Business Objects: First Guess, 28

Relationships, 29

Lab Exercise, 30

Database Design System, 30

All Powder Design, 31

Exercises, 38

Final Project, 40

Objectives

- Design the initial tables for the case.
- Create the design in the database design system.
- Determine the initial relationships for the case.
- Identify the data types needed for the attributes.

Database Design

You can design a database using paper and pencil. As you gain experience and become more skilled at the task, using pencil and paper will be relatively easy. However, when you are learning, using pencil and paper is tedious because you often need to remove items from potential classes or even alter the entire diagram. As an alternative, you might consider going directly to the DBMS and defining the tables or classes off the top of your head. This approach will not work with Oracle because Oracle limits the changes you can make to tables—particularly after relationships have been built and data has been added.

A few computer-assisted software engineering (CASE) tools remain that can help you define classes in a graphical environment. They are relatively powerful, and many have the ability to generate the final tables based on the class diagram. However, they are also expensive, hard to install, and cumbersome to learn. But if you work for a company that has invested in these tools, they are an excellent way to define the database classes. Oracle 11g does have a designer to build entity-relationship diagrams (Oracle SQL Developer Data Modeler). This system is useful because it will generate the tables from the diagrams. But it has limited advice and design checking facilities. It requires a separate download and installation from the technology network. Still, it can be useful for creating diagrams from existing databases to help you visualize the relationships.

There is a better tool to learn database design. The database design system is an online expert system that enables students to create class diagrams graphically in a Java-enabled Web browser. The system makes it easy for you to create classes (entities) and build associations (relationships). More importantly, it provides immediate feedback on the design, which is the expert system part. The system runs on a custom Web server and diagrams are stored in a central database. This approach means that you can access your diagrams from almost any computer. Changes you make in class or in your instructor's office are saved and available when you return to a lab or your own computer. From an instructional perspective, the best part is that the system contains some complex rules to provide feedback on your diagram. The system recognizes most design errors and points them out with suggestions to improve the design. Your instructor can obtain the database design system for your class. If it is available, you should use it to get the benefit of the immediate feedback. If it is not available, you can draw the class diagrams with paper and pencil or with a graphics package such as Visio or even PowerPoint.

Oracle Data Types

As a database designer, your job is to define the database tables that efficiently store the organization's data and support the business rules. In this process, you will define the tables in terms of the data columns (attributes) and the table relationships (associations). You will also need to know what type of data will be stored in each column. Also, for some columns, you will want to identify specify constraints (such as salary cannot be negative).

Selecting the proper data type can sometimes be a difficult step. Any DBMS supports only a limited number of domains and you have to understand the capabilities and limitations of each type. You must also understand the underlying business data—both the values collected today and the potential values that may be collected in the future. For example, workers may only use integer values to

represent a quality rating. But, in the future, it is likely that the company will want to use fractional values as well. Although database types are becoming more standardized over time, each DBMS uses its own type names. Even more confusing, the actual values supported can be different even if the data type name is the same. The numeric data type is variable length in Oracle, because you can specify the number of significant digits. A full 38 digits requires 21 bytes of storage.

Figure 2.1 shows the main data types available in Oracle 11g. The types you will use most often are NVARCHAR2, DATE, and NUMBER. When you need to store date or time values, be sure to use the DATE or TIMESTAMP type. It supports date arithmetic so users can subtract two dates to obtain the number of days between them. The LONG RAW and BLOB types can hold pictures or even spreadsheets or documents. Note that Oracle also supports the ANSI SQL keywords. In many cases, it is easier to use those instead of the Oracle types, but ultimately Oracle converts them into native types. For instance, SQL defines the INTEGER data type, which Oracle converts to NUMBER with a scale of zero.

Oracle essentially uses one numeric type for every type of number. This approach is relatively easy to use, but might not yield the most efficient use of storage space. On the other hand, storage space is cheap today, and no one really knows how many product item numbers the application will eventually need. So, using some extra storage space now is probably not a major problem.

The issue of precision and scale is sometimes confusing. Precision represents the total number of significant digits supported in the value—regardless of any decimal points or size of the number. For example, a number with a precision of 5 digits would include 12345 as well as 12.345. If the scale is specified, it indicates a fixed number of decimal points and controls round off to that value. It is particularly useful for handling currency values. Oracle automatically allocates a number of physical bytes for storing numbers based on the specified precision. The space required to store a number can range from 1 to 22 bytes.

Figure 2.1

	Name	Data	Bytes
Text (Characters)			
Fixed	CHAR or NCHAR	2000 bytes	Fixed
Variable	VARCHAR2	4000 bytes	Variable
National/Unicode	NVARCHAR2	4000 characters	Variable
Memo	LONG	2 gigabytes	Variable
Numeric			
Byte (8 bits)	NUMBER(38)	38 digits	2-21
Integer (16 bits)	NUMBER(38)	38 digits	2-21
Long (32 bits)	NUMBER(38)	38 digits	2-21
(64 bits)	NUMBER(38)	38 digits	2-21
Fixed precision	NUMBER(p,s)	p: 1...38, s: -84...127	2-21
Float	NUMBER	38 digits	2-21
Double	NUMBER	38 digits	2-21
Currency	NUMBER(p,4)	38 digits	2-21
Yes/No	NA		
Date/Time	DATE, TIMESTAMP,	1/1/-4712 to 12/31/9999	7/11/13
Interval	INTERVAL YEAR/MONTH	(sec)	
Image	LONG RAW or BLOB	2 gigabytes, 4 gigabytes	Variable
Generated Key	SEQUENCE	Long (+/- 2 billion)	4

The other confusing issue in modern databases is the use of Unicode or “national” character sets. The older VARCHAR2 data type assigns one character to one byte and can only handle ASCII codes or essentially English-language characters. If your database needs to store text in additional languages, it will have to use Unicode character sets that typically assign two bytes to any character or ideogram. In this case, use the NVARCHAR2 data type, but note that it cuts the maximum length of text in half. VARCHAR2 can handle strings up to 4,000 bytes. NVARCHAR2 can also handle 4,000 bytes, but that is only 2,000 Unicode characters.

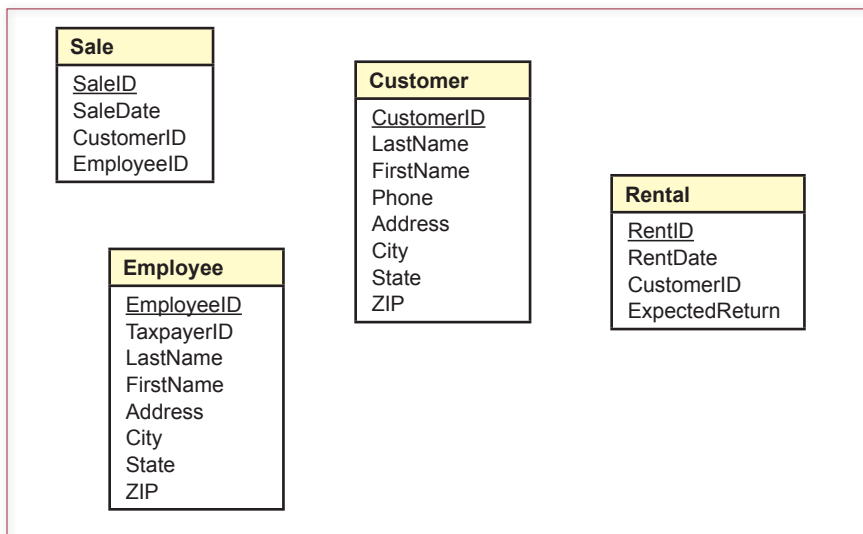
Case: All Powder Board and Ski Shop

With any database project, the first step is to understand the various elements of the organization and the components that will become part of the database application. This knowledge is critical, because the database design must reflect the business rules. In an actual work environment, you can ask workers about the processes and underlying assumptions. With a written case, it can be more challenging to determine all of the necessary rules. On the other hand, real life is messier, and people often give inconsistent answers. It takes experience to learn how to talk with users to identify exactly which components are the most important, and how the pieces relate to each other. Cases avoid this design complication but generally require you to make assumptions on your own. Because the goal is to make reasonable assumptions, you should search the Internet or read a few articles on snowboards and skis before you tackle the database design.

Business Objects: First Guess

One of the first steps in designing the database is to identify the business objects. In many ways, this case is a fairly typical business problem, so you would expect to see many of the traditional business objects, such as Customer, Employee, and Sale. Because the store also rents equipment, there will be a Rental object similar to the Sale object. Figure 2.2 shows initial versions of these four classes. These objects are relatively standard, but some issues arise in this case. Notice that you

Figure 2.2



must also begin to think about primary keys. In each of these four tables, the primary key is a new value that will be generated by the DBMS. In Oracle, you have to assign this column a NUMBER or INTEGER data type. Later, you can create a sequence that will generate the values needed. In terms of the design, you will choose an integer data type and set it as a generated key column. In most situations, the actual key values will be hidden from the users, and they will see only the relevant names.

Notice that several attributes are missing from these initial classes. The main reason is that it is important to ensure that the columns you include at this stage are correct. If there is any doubt about a column in a potential class, leave it out and think about it. A few other classes should be relatively obvious for this case. In particular, several support tables are used to provide look up data for other tables. Ultimately, you will have to define all of the objects, identify the columns for each table, and specify the data type for each column.

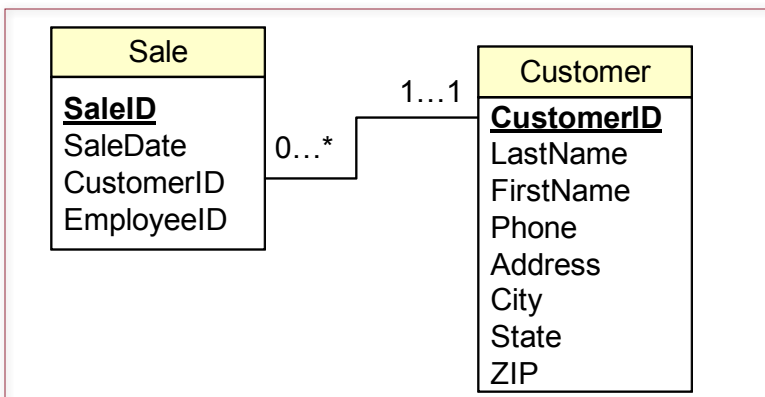
Your goal at this stage is to identify the primary entities needed in the case. Be sure you specify a primary key column. Then think about the main attributes that belong to each entity. For example, customers will be identified by an internal CustomerID, and data is collected on LastName, FirstName, Phone, Email, Address, City, State, and so on. Each of these attributes are single-valued and depend only on a specific customer. That is, once you are given a value for CustomerID, there will be only one matching name, phone number, and e-mail address.

Relationships

Classes or entities are related to other classes. For example, notice that the Sale table contains a CustomerID property. Values in this column match entries in the Customer table, which is keyed by CustomerID. So, if you found a CustomerID value of 112 in the Sale table, you could look up the matching customer data by finding the row in the Customer table that has a primary key value of 112. This association also expresses several business rules. In particular, (1) each sale can be placed by only one customer, (2) a sale must be identified with a customer, (3) any given customer can participate in many sales, but (4) a customer might not have bought anything yet.

Relationships are displayed on the diagram by drawing connecting lines between the two tables involved. The business rules are shown as annotations at the end of each connection. Each side of the connection displays minimum and maximum values. Figure 2.3 shows the association between the Sale and Customer

Figure 2.3



tables. Notice that the annotations match the four business rules described in the previous paragraph. The 1...1 notation on the Customer side represents rules 1 and 2. At a minimum, each sale requires at least one customer, and at a maximum, a sale can have no more than one customer. Likewise, the 0...* annotation represents rules 3 and 4. A customer can participate in zero to many sales. There is no maximum (*), so a customer can participate in any number of sales, and the zero means that a customer might not have bought anything yet. As a database designer, your job is to identify the entities and relationships needed for this case.

Lab Exercise

Database Design System

The database design system was built as an instructional tool, so your instructor should have already registered to obtain an instructor account. The instructor also chooses and schedules assignments for the class. You will need an Admit code to register for a class, so be sure you get the correct admission code from your instructor. When you purchased the Database book, you created a login account on the server to download the books. This account also gives you access to the DBDesign system. If your instructor chooses not to use the DBDesign system, you can still work with a couple of problems in the World/Open class. The Admit code is blank for that class, and the problems are not monitored—although work might be deleted eventually.

Action

Browser: <http://JerryPost.com/dbdesign>
Login in with your DB Book account.

Figure 2.4

The screenshot shows a web browser window titled "Student Register for Class - Microsoft Internet Explorer". The address bar shows the URL <http://jerrypost.com/dbdesign/StudentRegisterClass.aspx>. The page content is titled "Register for Class" and includes the following text: "Students must enroll in the correct class. Your instructor should have given you an AdmitCode, which enables you to enroll in a particular class. If you do not have a code, try enrolling by leaving the code blank. If that does not work, ask your instructor for the proper code." Below this text are four dropdown menus: "Country/State" (set to "USA All States"), "School" (set to "University of the Pacific"), "Class" (set to "Open Database Management Fall 2005 Open - 1"), and "Admit Code" (empty). At the bottom are three buttons: "Register", "Close", and "Cancel". Three callout boxes with arrows point to the form elements: "Select a country or state to narrow the school list." points to the Country/State dropdown; "Select your university and class." points to the School dropdown; and "Enter the admit code." points to the Admit Code input field.



Activity: Getting Started

Use your browser to navigate to the database design Web site and log in with your database book account. You should use the Personal Data link to ensure that your name, e-mail address, and Student ID number are correct. Your instructor will use the name and ID number to correctly identify you so you receive credit for working on assignments. Note that your ID and password are encrypted on the Web site database to protect them. However, if your university still uses your Social Security number as an identifier, you might want to enter only a portion of the number—and then go ask your university to wake up and create a safer number. Your e-mail address is important so the system can send you the username and password in case you forget what you selected.

Action

File/Open, choose All Powder case.

Right click/Add Table.

Type “Sale” as the new table name.

Drag columns from right onto table.

Right click name/set data type.

Once you have successfully created the new account, you must register for the specific class. As shown in Figure 2.5, you simply choose your university and your correct class. Enter the admission code provided by the instructor and click the button to register for the class. If you do not have the proper code and are unable to register, you can get the code and return later. If you did not register for a class when you first created the account, you need to do that now. Scroll down to the bottom of the DBDesign page and click the link at the bottom to register for a class. In fact, once you get to the design page, if you try to open a problem (File/Open) and the list is empty, it is most likely because you are not registered for a class.

All Powder Design



Activity: Create Tables and Columns

When you log into the system you are ready to begin designing the database. Figure 2.5 shows the main elements of the system with the beginning of the solution. When you begin, the various windows will be empty. You must first open a problem using the File/Open menu and select the All Powder Workbook case. When the problem loads, the right-hand window will display a list of available columns. Initially, it will probably not include the key columns. You will add those in a minute.

You create a table (class/entity) by clicking the right mouse button on the main screen where you want the table located. Then select the Add Table option. Rename the table by typing “Sale” as the new name, and pressing the Enter key.

Now you get to add columns to the table. All columns are added to a table by dragging them from the right-hand window and dropping them onto the desired table. In the case of the Sale table, you will need to generate a new primary key column (SaleID). To create a generated key column, drag-and-drop the top label for Generate Key. Then, rename the newly created column. You rename columns by double-clicking the name either in the table or in the right-hand window. Be careful: Do not give two columns the same name, even if they are in different tables. You will not be able to tell them apart in the main list of the right-hand window. You might want to use an abbreviation and separator, such as Cust_LastName. Later, the system can remove the prefix when it generates Oracle tables. Now you can add some of the other columns needed in the Sale table. Look through the

right-hand window to find the SaleDate and SalesTax entries. You can simplify your search if you sort the list by right-clicking on it and selecting Sort. Drag the desired column onto the Sale table. Once a column is in the table, you can change the order by dragging and dropping it higher or lower in the list.

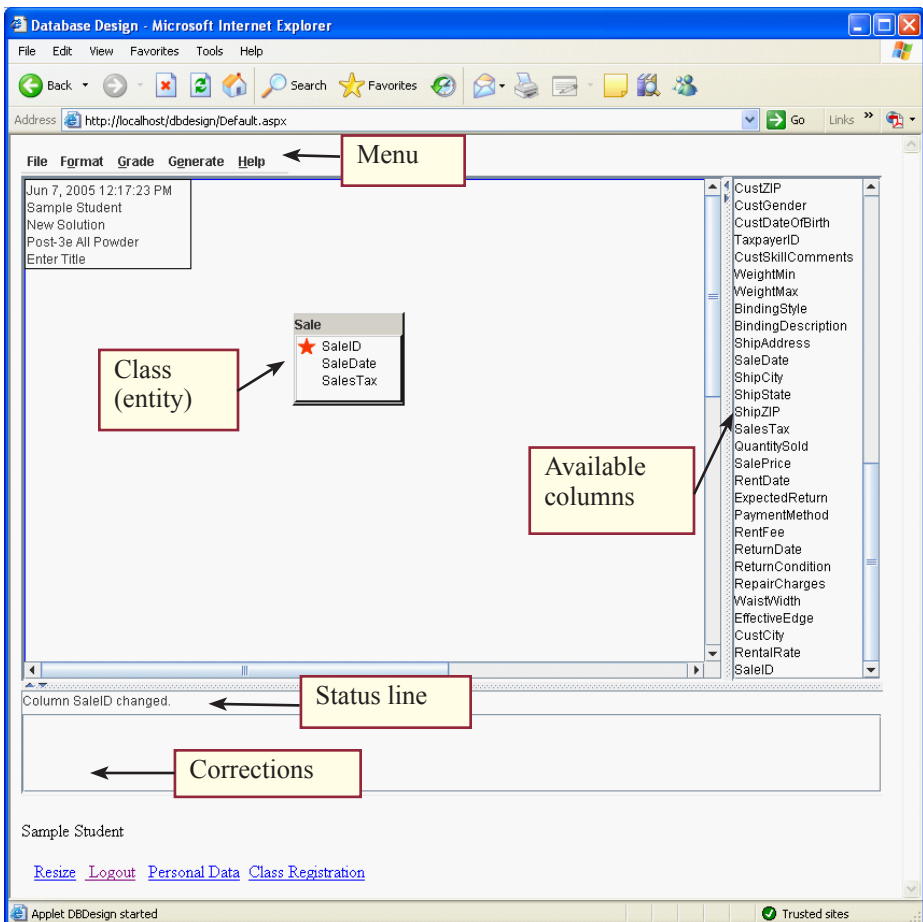
At this point, you should set the data types of the columns in the table. The default type is Text, so in many cases you will not have to change it. However, you should choose Date/Time for the SaleDate, and Currency for the SalesTax column. Double-click on the column name within the table to open the column editor. You can select the data type and change the data size if desired. You can also add a constraint and default value, but you should probably do those later. The default value is straightforward, but the constraint has to be expressed in Oracle's format. Be sure to save your work every few minutes in case you lose the Internet connection or the server times out.



Activity: Create Relationships

Associations or relationships are a key element of database design. In a relational database, columns in one table are connected to columns in other tables through common data. In the case, the Sale table needs to connect to a Customer table.

Figure 2.5



Eventually, both tables will contain a CustomerID column. First, you have to create the Customer table, so right-click on the design screen, add a new table, and rename it. Again, to ensure that each customer is assigned a guaranteed unique identifier, add a Generate Key column to it. Rename this new column as the CustomerID. It is critical that you understand that this key value will be generated for each new customer added to the table. This value can only be generated in the Customer table. You would never create another generated key column and call it CustomerID. Notice that the column is marked with a solid (red) star to indicate that it is a key with values generated in this table. Before attempting to build the relationship, add the other customer properties to the Customer table by dragging them from the right-hand window. You can use the Shift or Ctrl key to select multiple columns at a time, but moving them takes a little practice. You can double-click the table heading to automatically resize the table design box to fit the columns it contains. Set the appropriate data types.

Action

Add Customer and Sale tables.

Add GenerateKey to Customer table.

Rename it to CustomerID.

Drag new CustomerID from right side into Sale table.

Drag CustomerID from Customer and drop it on CustomerID in Sale table.

Fill out relationship box.

How do you get CustomerID into the Sale table? Scroll the right-hand window to the bottom and notice that CustomerID has been added to the list of available columns. You could also sort the list and find it alphabetically. You can now drag this new column into the Sale table. Make sure its data type is Integer32 (Long). Now that you have both the Sale and Customer tables, and they both have a CustomerID column, you can build an association or relationship between them. Figure 2.6 shows how to create this relationship in the design system. This relationship and the edit window are automatically created when you drag the foreign key CustomerID into the Sale table. You can edit relationships by double-clicking the slanted relationship line, or right-clicking the line and choosing the Edit option. You can also create new relationships by dragging a column from one table and dropping it onto a column in the second table. For example, if you delete the relationship between Sale and Customer, you can click on the CustomerID column in the Customer table and drag it to the Sale table. Release the mouse button to drop the cursor onto the CustomerID column in the Sale table.

The relationship window asks you to specify the minimum and maximum values for each side of the relationship. These values specify the business rules, and are often the most difficult items to identify. In the sale case, the typical assumptions are that exactly one customer can place an order, and a customer can place from zero to many orders. So, on the Sale side of the window, select the Optional and Many buttons. On the Customer side, choose the One option for both Min and Max values. In most cases, the system will automatically attempt to create the correct relationship for you when you add the CustomerID column to the Sale table. But, this method only works if the other keys are set correctly.

Remember that relationships generally involve at least one side in a primary key. The column names are often the same on each end, but they can be different. However, the data types do have to match, and the relationship has to be logical. For example, it would never make sense to connect an ItemID to a CustomerID, because that relationship would imply that a customer can also be an item and vice versa. Finally, notice that the integrity and cascade boxes are selected as the

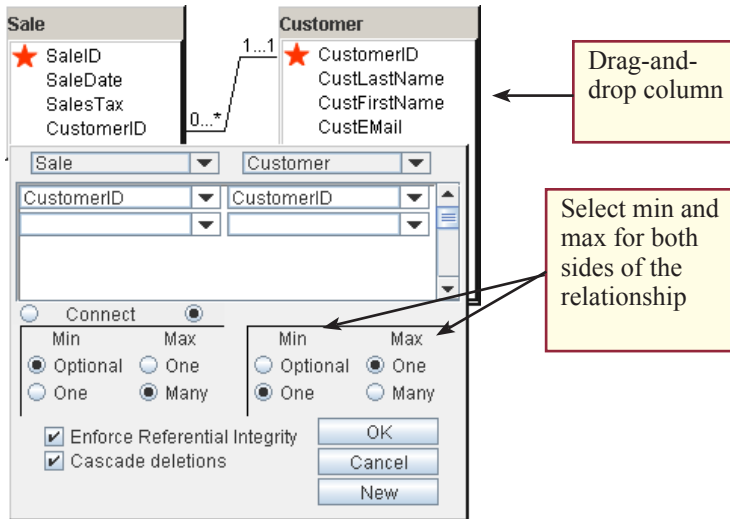


Figure 2.6

default. You should almost always leave these checked. In the database, cascade on delete means that if you delete a particular customer, all of the orders placed by that customer will also be deleted. If you do not specify the cascade, then you could end up with orders that contain a CustomerID, which has no matching customer data. After you close the relationship window with the OK button, you might have to refresh the display screen by right-clicking the design window and selecting Refresh.



Activity: Evaluate the Design

One of the most powerful aspects of the database design system is that it contains an expert system to help analyze your design for errors. You can quickly obtain comments by selecting the Grade/Grade and Mark option on the menu. At this point, you have only two tables partially created, so the most important comment you should receive is that overall, you are missing several tables. The system might also point out that you are missing columns from the Sales table, because you have not yet added the salesperson (employee) and the shipping information.

Action

- Choose Grade/Grade and Mark.
- Click messages in window.
- Fix errors by removing columns and adding new tables.

To illustrate the power of the system, you will add a new table (Item), and then build a new relationship that is incorrect. Add a new table for Inventory, and add the SKU column (a common retail abbreviation for stock-keeping unit) used to identify individual products. Right-click the SKU column in the Inventory table and set it as a key. Add the Size and QOH columns to the Inventory table. Set their data types to Single and Integer16 respectively. Now add the SKU column to the Sale table as an intentional error. Create a relationship from Inventory to Sale using the SKU columns.

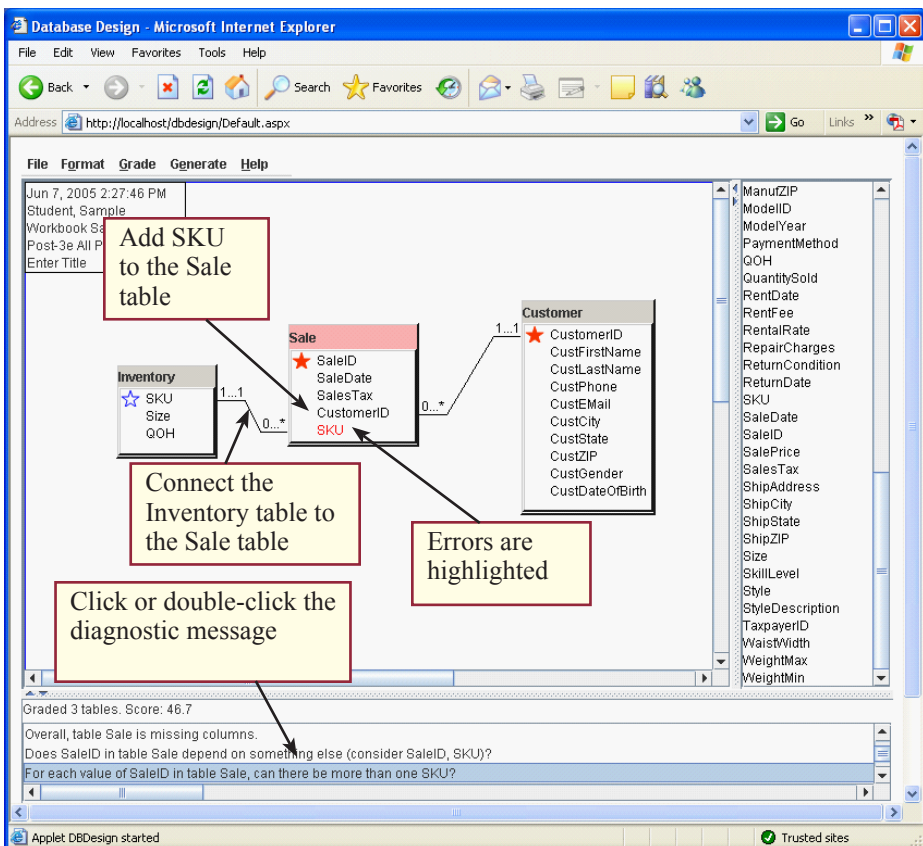
Choose the Grade/Grade and Mark menu option to save the changes and obtain comments on the design. Again, the design is not finished, so focus on the other error messages. In particular, find the message “For each value of SaleID

in table Sale, can there be more than one SKU?” and click it. Figure 2.7 shows the resulting diagnostic screen. The SKU column in the Sale table is highlighted as a potential problem. Indeed, it is an issue, because placing SKU into the Sale table as shown would mean that for each Sale, only one item (SKU) can be sold. You can usually double-click the comment to receive additional information about database design. In this case, notice that SKU is not part of the primary key. A primary key has a many-to-one relationship with the rest of the data. So a table of the form: SaleID, SKU means that there can be many SaleID values (keyed), but only one SKU for each sale.

You might try to fix the problem by making the SKU part of the primary key: SaleID, SKU. This action will solve the first problem, but it creates new ones. If you set SKU as a key and resubmit the problem for grading, it will return several messages. One of them will be the question “Does SaleDate in table Sales really depend on SKU?” That is, your table now includes SaleID, SKU, SaleDate, CustomerID. The system is asking whether the SaleDate (and CustomerID) actually depend on the SKU. That is, are the different items (SKU) sold to different people on different days; or do they really depend on just the SaleID?

Notice that sometimes a table has many errors, so you must carefully review the entire table to make sure you fix the primary problems first. The Grade menu also contains an option to generate a separate HTML file that lists all errors by table. This listing is easier to print.

Figure 2.7



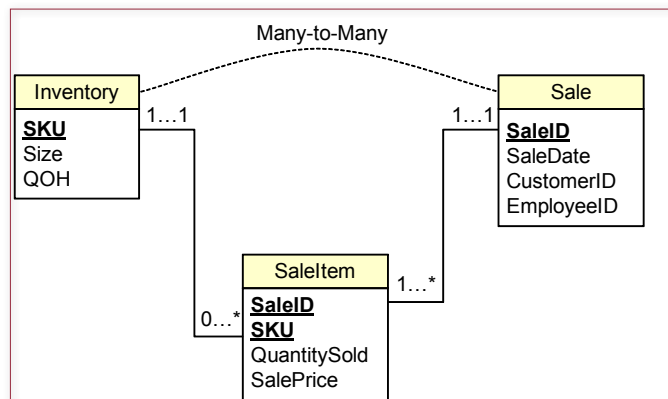
Primary keys are one of the most difficult things for students to understand when they first start designing databases. In particular, generated keys are tricky. In terms of the database design system, primary keys are critical because they are used to identify the tables. If you make major mistakes in the primary keys, the system will give confusing feedback because it cannot correctly identify your tables. For this reason, it is always best to begin with one or two tables, test them, and then slowly add more tables and relationships.

You still need to fix the problem with the Inventory and Sale table association. In a broad sense, it seems that there should be some type of connection between Inventory item and Sale to indicate which items were purchased by the customer. But placing the SKU attribute into the Sale entity appears to be a bad idea. The reason is straightforward. If there is an association between Inventory and Sale, it must be many-to-many. That is, a Sale can include many items (SKUs), and an Inventory item (SKU) can be sold many times. Relational databases do not handle many-to-many relationships directly. Instead, you must create an intermediary or junction table.

Figure 2.8 shows the creation of the intermediary table. It contains the key columns from both the Inventory (SKU) and Sale (SaleID) tables. Both columns are keyed in the new SaleItem table. Examining the keys within the SaleItem tables reveals that each sale can contain many items, and each item can appear on many sales. This is exactly the many-to-many relationship needed. The additional columns of QuantitySold and SalePrice indicate the number of items being purchased and any discounts applied—for an individual item on a specific sale. The dashed many-to-many line is never created, it is simply used here to show the goal of the two relationships.

The new SaleItem table corresponds to the repeating lines of items that you would see listed on a paper sale form. Examining the two new relationships reveals how the table works. Reading from the Sale to the SaleItem table, each sale can contain from one to many items, and in reverse, each SaleItem line (SaleID and SKU) refers to exactly one sale. Essentially the same association exists from Inventory to SaleItem. However, since items might not have been sold, each item can appear on zero to many sales lines, and a given sales line refers to exactly one item. All many-to-many relationships must be split and joined with a junction table that contains the keys from both of the original tables.

Figure 2.8





Activity: Fix Inventory Design

Return to the database design system and delete the association between Inventory and Sale. Then remove the SKU column from the Sale table. Now you can create the SaleItem table. Simply drag the two keys (SaleID and SKU) into the table from the right-hand window—do not attempt to re-create them with a generate key. Double-click to the left of both names to add the simple key icon (unfilled blue star). Build the two new relationships in the Figure 2.8 example and add QuantitySold and SalePrice to the SaleItem table. Make sure the SalePrice data type is Currency and that the data size does not exceed 38, the maximum number of digits allowed in an Oracle number.

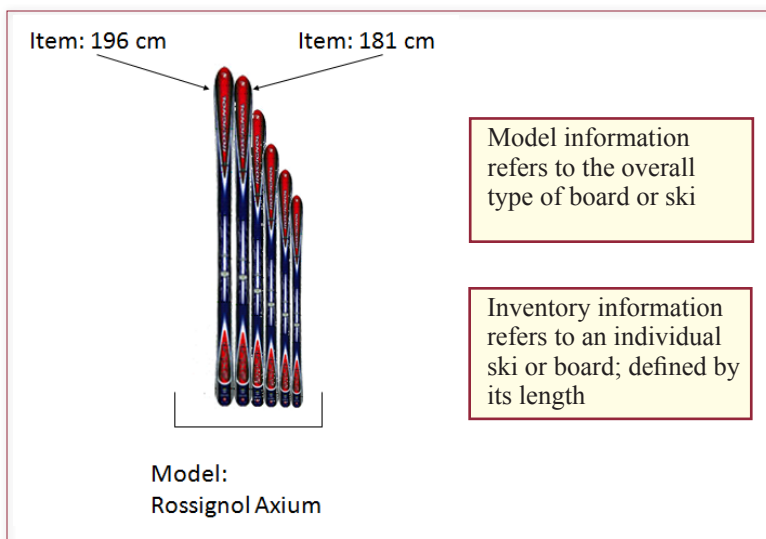
Action

- Create the SaleItem table.
- Create the ItemModel table.
- Include the proper columns.
- Set the keys.
- Set the data types.
- Grade/Grade and Mark.

If you grade this version, you will see that the detail issues have been corrected. However, some design issues still exist in terms of handling inventory. The inventory for a ski shop is somewhat more complicated than for a typical retail store. In particular, snowboards and skis are sold in varying lengths to match the individual customer. Figure 2.9 shows the two concepts. A manufacturer produces a model line that exhibits certain characteristics such as color, width, flexibility, and side cut. For each model type, several different lengths are available. From the perspective of the All Powder store, the database has to keep information on each model, but the actual inventory must refer to a specific item or length within the model type. Each item will receive a different SKU. For example, SKU 1173 might refer to a Rossignol Axium ski that is 196 cm in length, while SKU 1174 references a Rossignol Axium ski of 181 cm.

The catch is that it would waste considerable space to repeat all of the model data for every possible size of ski or board. Consequently, it is important to create two entities to handle the details: ItemModel and Inventory. Figure 2.10 shows the basic tables and the resulting relationships. Observe that each model results in many inventory items (multiple sizes of boards or skis), but each item can be

Figure 2.9



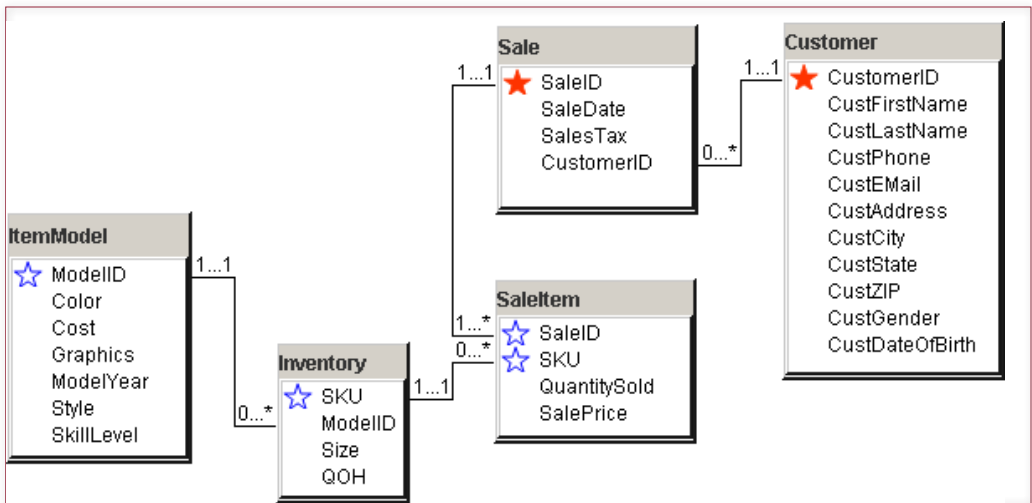


Figure 2.10

only one model type. At this point, you should be able to add more attributes and more tables to the design, but the completion of the design will be left to the next chapter.

Exercises



Crystal Tigers

Crystal Tigers is a service club with about 150 members. The club primarily sponsors events such as community pancake breakfasts, local concerts, and sporting competitions. The club successfully uses the events to raise money for various charitable organizations. The club needs a database to help track the roles of the various members, both in terms of positions within the organization and their work at the events. The following form represents the basic data that needs to be collected.

	Year	Position/Title	Comment
Last Name, First Name Phone, Cell Phone Address City, State, ZIPCode			
Event title Start Date End Date Charity Charity contact Phone Amount raised	Member Activities for Event		
	Date	Hours	Activity

1. Analyze the form and create the main classes and associations needed to maintain the data for this organization.



Capitol Artists

Capitol Artists is a partnership among several commercial artists that work on freelance and contract jobs for various clients. Some jobs are contracted at a fixed price, but complex jobs require billing clients for the number of hours involved in the project. To help the artists track the time spent on each project, the firm wants you to build an easy-to-use database. On a given day, the artist should be able to select the time slot, then choose a category and a job. All jobs are given internal numbers, and each job has only one client. But, it is helpful to list the client information on the form once the job has been selected. The artist then enters a short task description, the billing rate, and any out-of-pocket expenses. The billing rate is somewhat flexible and depends on the client, the job, the task, and the artist. For example, the company can charge higher rates for an artist's creative work time, but lower rates for copying papers. The following form contains the basic information desired.

Employee Last name, First name Date								
Time	Category	Client	Job#	Task	Description	Hours	Rate	Expenses
8:00 AM	Meeting	Name + Phone	1173					
8:30 AM								
9:00 AM								
9:30 AM								
...								

1. Analyze the form and create the main classes and associations needed to maintain the data for this organization.



Offshore Speed

The Offshore Speed company sells parts and components for high-performance boats. Some of the customers modify the boats for racing, others simply want faster boats for informal races. The engine parts tend to be highly specialized and new variations are released each year by manufacturers. Compatibility of parts is always a major issue, but most are tested by the manufacturers with data available from their websites. Customers tend to order parts through the store, but sometimes they will buy off-the-shelf components. The store also keeps many spare parts in stock because customers tend to break them often and the profit margins are good. The store also has arrangements with other firms that can help customers redesign and upgrade interiors and cabins, for example, provide new upholstery for seats and complete systems for beds and sinks for cabins. Lately, the store has been successful in selling and installing high-end GPS and communication systems. The form below is used to place custom orders for the clients. Discounts are given to customers based on several subjective factors that will not be entered into the database.

Customer Last name, First name Phone, E-mail Address City, State, ZIP				Employee Sale date Estiamted receive date		
Boat: Brand, year, # engines, length Engine 1: Brand, year, out drive, year Engine 2: Brand, year, out drive, year						
Manuf.	Mfg Part No.	Category	Description	Quantity	List Price	Extended
					Subtotal	
					Tax	
					Discount	
					Total Due	

1. Analyze the form and create the main classes and associations needed to maintain the data for this organization.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following task.

1. Analyze the forms and create the main classes and associations needed to maintain the data for this organization.

Data Normalization

Chapter Outline

Database Design, 42

Generated Keys: Sequences, 43

Case: All Powder Board and Ski Shop, 43

Lab Exercise, 44

All Powder Board and Ski Database Creation, 44

Exercises, 56

Final Project, 57

Objectives

- Understand how to use generated keys.
- Create tables and specify data types.
- Create relationships and specify cascades.
- Establish column constraints and default values.
- Create lookup lists for columns.
- Estimate the data volume for the database.

Database Design

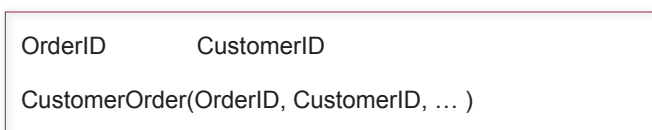
This chapter continues the concepts from Chapter 2 and adds a set of rules that specify when a column belongs in a table and when it does not. The main objective of database design is to define the tables, relationships, and constraints that describe the underlying business rules and efficiently store the data. The normalization rules are critical to properly identifying the columns that belong in each table. The first step is to make sure the keys are correct. A key uniquely identifies the rows in the table. If multiple columns are part of the key, it indicates a many-to-many relationship between the key columns.

Generated keys are guaranteed to be unique in the table in which they are created. For example, a generated key of CustomerID in the Customer table automatically creates a new key value for each customer row that is created. Hence, the generated key is always the only key column in that table. It would never make sense to add another key column to a table with a generated key—the generated key is always unique. So, in the designs, you should never have tables with both a generated key (solid red star) and any other key (open blue star). Other issues with keys are trickier and require the designer to understand the underlying business rules.

If you are uncertain about which columns should be keyed, write them down separately and evaluate the business rules between the two objects. Figure 3.1 shows a typical situation with orders and customers. First ask yourself, For a given order, can there ever be more than one customer? If the answer is “yes” based on the business rules, then you would mark the CustomerID column as key. But most businesses have a rule that each order is placed by only one customer, so CustomerID should not be keyed. Second, reverse the question and ask yourself, For a given customer, can there be more than one order? Obviously, most businesses want customers to place repeat orders, so the answer is “yes.” So you mark the OrderID as key. Since only OrderID is keyed, both columns belong in the CustomerOrder table, which is keyed by OrderID and contains the unkeyed CustomerID column.

Once the keys are correct, you need to check each nonkey column to ensure that it follows the three main normalization rules. First, each column must contain atomic or nonrepeating data, for example, a single phone number, but not multiple values of phone numbers. Second and third, each non-key column must depend on the whole key and nothing but the key. You need to examine each potential table, determine that the keys are correct, and then check each column to ensure that it depends on the whole key and nothing but the key. If there is a problem, you generally need to split the table. Remember that any time you make a change to the keys in a table, you have to reevaluate all of the columns in that table.

Figure 3.1



Generated Keys: Sequences

Key columns play a critical role in a relational database. The key values are used as a proxy for the rest of the data. For instance, once you know the CustomerID, the database can quickly retrieve the rest of the customer data. That is why you only need to place the CustomerID column in the CustomerOrder table. However, the database requires key values to be unique. Guaranteeing that key values are never repeated can be a challenging business problem. In some cases, businesses have separate methods to create key values. For instance, the marketing department might have a process to assign identifier numbers to customers and products. But the process must ensure that these values are never duplicated. In many situations, it is easier to have the database generate the key values automatically. In particular, orders often require keys that are generated quickly and accurately.

Oracle has a sophisticated sequence process to generate new key values. You assign a NUMBER type to the primary key in a table where you want the key value created. This data type does not actually create the number. To create numbers, you need to define a sequence that will generate the numbers on demand. The sequence generator is relatively flexible and you can specify a starting value and an increment. The final step is to create a database trigger that automatically gets a newly generated key value and places it into the primary key column. This step is a little trickier since database triggers are covered in a later lab. However, sequences really should be set up when you define the table so that you remember to do it. One of the activities in this lab will show you how to set up an automatically generated key value; you can copy the process for your other projects.

For now, you must carefully identify the key columns that might need generated values. For instance, the CustomerID column in the Customer table, or the OrderID in the Order table might be assigned a generated value. But the CustomerID column in the Order table would never be a generated key. It would be given the same numeric data type, although the actual key generation can take place only in the original (Customer) table. Make sure you understand the difference. The CustomerID is the only column that is a primary key in the Customer table, and it is the source table for customers. Consequently, it is acceptable to generate key values for CustomerID in the Customer table. On the other hand, the CustomerID is a placeholder in the Order table—it represents the customer placing the order. The customer is not created in the Order table, so the CustomerID value cannot be generated in the Order table. The CustomerID must already exist in the Customer table before it can be assigned to a row in the Order table.

Case: All Powder Board and Ski Shop

When you first approach a database design problem, you will often experience one of two perspectives: the project seems immensely complicated, or the project seems too easy. Usually, both perspectives are wrong. Even a difficult project can be handled if you divide it into small enough pieces; and few projects are as easy as they first appear. The main issue is to correctly identify the business rules. And there always seem to be complications with some of the rules. For the All Powder case, consider the issue of customer skill level. Whether a customer is renting or buying a board or skis, the salespeople need to match the person to the proper board or ski based on the customer's skill level. In terms of business decisions,

Consider what happens if you (incorrectly) try to place Style and SkillLevel in the Customer table:

CustomerID, LastName, ... Style, SkillLevel
CustomerID, LastName, ... **Style**, SkillLevel

Business rule: Each customer can have one skill in many styles.

*Business rule: Each style can apply to more than one customer.
Need a table with both attributes as keys.*

CustomerID, LastName, ... **Style**, SkillLevel

But you cannot include LastName, FirstName and so on, because then you would have to reenter that data for each customer skill.

Figure 3.2

managers need to identify the types of customers to plan for the models and inventory decisions for next season.

As shown in Figure 3.2, consider what happens if you try to place the Style (downhill, half pipe, and so on) and SkillLevel directly into the Customer table. The problem is that the business rules state that each customer can have one skill level in many styles, and each style can apply to more than one customer. For example, customer Jones could be an expert downhill skier, but only a beginner in half-pipe snowboard. However, customer Sanchez is an expert at half pipe, but has never tried any type of skiing. If you place Style and SkillLevel in the Customer table, you might try keying only CustomerID. But that action would state that each customer participates in only one style, with one skill level. On the other hand, if you key just the Style column, you would be indicating that each style can be performed by only one person. The only solution is to key both the CustomerID and the Style columns. Then, each customer can participate in many styles (with one skill rating per customer per style), and each style can apply to many people (with possibly different skill ratings). But you cannot leave the Style and SkillLevel columns in the main Customer table along with columns such as LastName. It is clear that a customer's last name does not change for each different style. A customer's last name depends only on the CustomerID, so you need to split the tables.

Figure 3.3 shows the resulting design. The Customer table is keyed only by CustomerID and contains attributes that describe each customer. The Style and SkillLevel tables are used as lookup tables to ensure that clerks select from the defined list of choices. Without them, the database would quickly become a mess because everyone would use different spellings and abbreviations for the entries. The CustomerSkill table contains the CustomerID and Style as key columns to support the business rules.

Lab Exercise

All Powder Board and Ski Database Creation

You should use the database design system to refine your table definitions. The system is designed to check the main design rules and ensure that your tables meet the requirements of good database design. However, if you make different assumptions about the underlying business rules, you can create slightly different tables than those recommended by the design system.

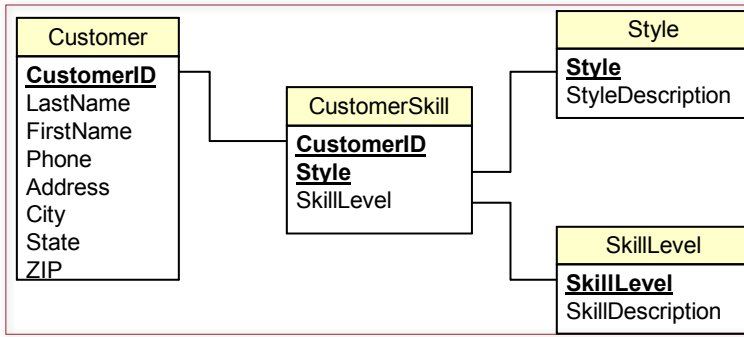


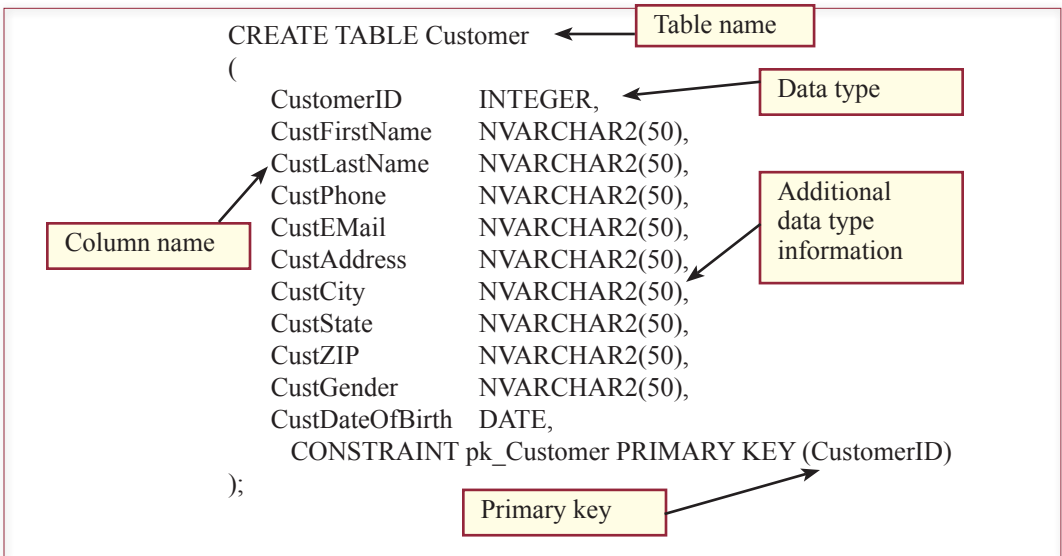
Figure 3.3

*Activity: Create Tables*

Technically, Oracle has two methods for creating tables: (1) SQL and (2) the SQL Worksheet (in SQL Developer and Enterprise Manager). SQL Developer has a visual tool for entering column names and selecting data types. It then generates the matching SQL CREATE TABLE commands. For beginners, the visual tool is relatively easy to use because you do not have to worry about syntax issues such as where to put the commas. However, the visual designer requires you to switch to Advanced mode to use the NVARCHAR2 (Unicode) data type, and you should always use this data type instead of the older VARCHAR2. But, advanced mode is cumbersome, so most designers ultimately just write the complete list of SQL statements. Even if you use the visual table designer, you need to switch to the SQL (DDL) and copy the CREATE TABLE command to your lab notebook. This way can recreate or modify the database design later.

Figure 3.4 uses the Customer table to show the basic structure of the CREATE TABLE command. Note that the entire description is contained in parentheses, and you need the semicolon at the end to tell Oracle you have finished entering

Figure 3.4



the command. Generally, each column is listed on a separate row followed by its data type. The NVARCHAR2 (text) data type requires a value for the maximum number of characters. Note that column descriptions are separated with a comma. The constraint row defines the primary key column. If you need to define multiple columns in the key, simply add the column name and separate it with a comma.

For text data you should generally use NVARCHAR2 instead of the older VARCHAR2. You never know when someone will want to store names or other data using a different language alphabet. Some data types have size limits. For example, you should specify the maximum number of characters expected in a text column. Oracle will efficiently store the data even if it takes less than the specified number of characters, but it will not allow anyone to enter a value with more than the number entered. Oracle will allow up to 2,000 characters for the NVARCHAR2 data type, but try to be somewhat conservative because Oracle uses those values to set default format widths.

Columns have additional options, such as NOT NULL, which forces the user to enter a valid value for the column. You can also provide default values that will be entered if the user does not specify a particular value. In general, the NOT NULL statement is automatically applied to primary key columns. You should avoid it for other columns because it forces users to enter data, and sometimes they need more flexibility.

Primary keys are a little tricky in SQL, since they are entered as constraints on the table. Every constraint needs a name and you should choose a name that is recognizable later. Using pk_table is a useful convention. If you receive an error message later, it will include the name of the constraint. If you see the primary key constraint name, you immediately know that either the key value is missing, or someone tried to enter a duplicate value that already exists.

Although Oracle handles all numerical data with the NUMBER data type, you must still be careful about selecting the size and scale of the number. In particular, you have to make a decision about decimal places. If the column will contain only integer values, you enter a zero for the scale since there are no digits to the right of the decimal point. If the column will hold currency data, you will usually specify a scale of 2, but you could use additional digits if you want to examine round-off issues. To get floating point numbers, enter zero for both the size and scale values.

In the All Power case, most skis and boards are measured in centimeters, so the numbers are not overly large. However, some manufacturers might choose to use fractional lengths, so the single-precision floating point is appropriate. This step is sometimes difficult for beginners to catch. If you forget to choose the single- or double-precision subtype, you will not be able to enter fractional values (with decimal points). If you ever encounter that problem, simply return to the Design view and set the proper size and scale values.

Action

Start SQL Developer and type in the CREATE TABLE Customer command. Enter column names and data types. Assign the primary key. Make sure the command runs with no errors.

Action

Write the CREATE TABLE commands to build the ItemModel table. Be sure to include the CHECK constraint. Test the constraint with sample data, using a INSERT INTO statements.

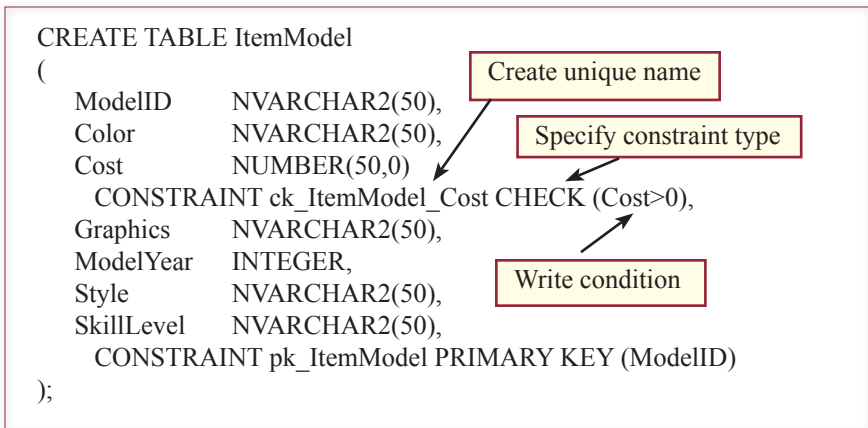


Figure 3.5



Activity: Create Constraints and Default Values

In many cases, you will want the database to enforce the business rules. Placing the rules in the database means that they will be enforced in all situations, without relying on other programs. Figure 3.5 shows the statements for setting a condition to ensure that cost values are always positive. Pay particular attention to the commas—there are no commas within a column definition, only at the end of each column. The condition must be entered in parentheses and must represent a valid SQL WHERE statement. Almost any SQL condition can be used and they will be explained in detail in Chapter 4.

As a more complex example, you might want to return to the Customer table and add a constraint that limits the values that can be entered for Gender. The easiest approach would be to drop the existing table (`DROP TABLE Customer;`) and create it again; adding a new CHECK constraint. However, you could also use the ALTER TABLE command. In either case, you want people to enter data from a fixed list of items (female, male, and unidentified). You could probably get by without the “unidentified” option by using null values for that purpose, but it is a little easier for users if you specify it as a possibility. The condition that enforces this constraint is `UPPER(Gender) IN ('FEMALE', 'MALE', 'UNIDENTIFIED')`. The UPPER function converts whatever text is entered into all uppercase characters because the comparison is case sensitive. The three acceptable items are entered in the list with single quotes around each word or phrase and separated by commas.

Notice that it is straightforward to specify default values. These are values that you want entered whenever the user does not provide a value for the

Action

- Write the CREATE TABLE statement for Department.
- Be sure the Department column is keyed.
- Write the CREATE TABLE statement for Employee.
- Set EmployeeID as a primary key constraint.
- Create a new foreign key constraint.
- Name it `fk_EmployeeDept`.
- Specify the Department table as a reference.
- Add the Cascade On Delete option.
- Run the two CREATE statements and fix any errors.
- Insert a Department row, then an Employee row.

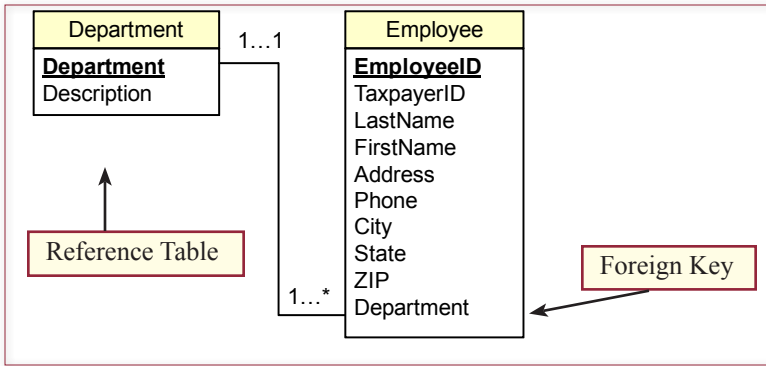


Figure 3.6

specified column. The user can override the default value and enter something else, but it is often convenient to display a commonly used value to save time for users entering data. For example, a **SaleDate** can be set to the **SYSDATE** function so that the current date is automatically entered. For example, to specify a default value of 1 for **Cost**, simply add the line; **DEFAULT 1** (with no commas or equal sign).

Figure 3.7

```
CREATE TABLE Department
(
  Department      NVARCHAR2(50),
  Description     NVARCHAR2(150),
  CONSTRAINT pk_Department PRIMARY KEY (Department)
);
CREATE TABLE Employee
(
  EmployeeID     NUMBER(12),
  TaxpayerID     NVARCHAR2(50),
  LastName       NVARCHAR2(25),
  FirstName      NVARCHAR2(25),
  Address        NVARCHAR2(50),
  Phone         NVARCHAR2(25),
  City          NVARCHAR2(50),
  State         NVARCHAR2(15),
  ZIP           NVARCHAR2(15),
  Department     NVARCHAR2(50)
  DEFAULT 'Sales',
  CONSTRAINT pk_Employee PRIMARY KEY (EmployeeID),
  CONSTRAINT fk_DepartmentEmployee FOREIGN KEY (Department)
  REFERENCES Department(Department)
  ON DELETE CASCADE
);
```



Relationships

Activity: Define Relationships

Relationships in Oracle and SQL can be somewhat difficult to see, since there is no visual representation. In the database design system class diagram or an entity-relationship diagram, relationships are shown as a line between two tables. Figure 3.6 shows a typical relationship between the Department and Employee tables. Employees are

assigned to a department, but the department comes from a list in the Department table. In this example, the Department column in the Employee table is a foreign key because it refers to a primary key in a second table. The Department table is the reference table because it supplies the data to the Employee table.

Relationships are created in Oracle by defining foreign key constraints. In this example, any department value entered into the Employee table must already exist in the Department table. Any other value would be invalid and an error message will be presented to the user. Creating a foreign key constraint has the same requirement: You must first define the reference table before you can create the foreign key relationship. In this case, you must first create the Department table. At a minimum, the Department table should have a Department column as the primary key. You might also consider adding a Description column in case the names of the departments need a longer explanation. You can create this table with the designer or with SQL. Just remember that it must be created before the Employee table is defined!

You create foreign key relationships within SQL when you define the Employee table that holds the reference to the Department table. Figure 3.7 shows the definitions of the Department and Employee tables. Foreign key relationships are the main reason that it is often easier to create tables within a text file first and then execute the text file. Remember that the tables must be created in a specific order. In this example, the Department table has to be defined before the Employee table. With a text file, you save the entire database structure and re-create it almost instantly. The foreign key constraint is straightforward, but you have to enter the keywords in the specified order. You begin with the CONSTRAINT keyword followed by the name of the constraint as usual. The FOREIGN KEY phrase specifies the type of constraint, and it is followed by the name of the column (or columns) in the Employee table that is affected by the constraint. The keyword REFERENCES is followed by the name of the reference table (Department), and the column referred to is listed in parentheses. You can include multiple columns in both the FOREIGN KEY list and the REFERENCES list, and the relationship will pair-match all of the columns.

Note the use of the ON DELETE CASCADE command to set the Cascade option. This option helps ensure the data remain consistent, and it makes it easier to delete items from the database. In this example, if you delete a department from the Department table, all employees assigned to that department will also be removed from the database. Because cascade deletes can remove substantial chunks of the database, you will eventually need to impose some strict security limits on who can delete departments.

Action

- Choose the Generate/Set DBMS option on the main menu.
- Choose Oracle and click Save.
- Choose Generate/Generate Tables.
- Copy the SQL commands and paste them into WordPad.
- Save the file as AllPowder.sql.
- Start SQL Developer and run the file with @<location>\AllPowder.sql.

One table can have several relationships with other tables. You simply list each one as a new foreign key constraint. However, make sure that each reference is valid.

Figure 3.7 also shows how to specify a default value for the Department column. In this case, employees will be assigned to the Sales department if no other value is entered. Of course, you should make sure that the Sales department is listed in the Department table.



Activity: Create Tables with Database Design

The basic syntax of the CREATE TABLE command is not too difficult, but the typing becomes tedious after a couple of tables. And the foreign key relationships require some thought. Also, if you make a mistake, you usually have to drop the tables and start over. That is why it is so important to keep the CREATE TABLE commands in a separate file. For example, if you name the file something like AllPower.sql, you can run it from within SQL Developer with the command: Start location\AllPower.sql, where location is the full drive and file location on your computer.

But, it would be nice if there were an easier way to create the tables. So, you should be happy to learn that the current version of the database design system generates the SQL statements for you. The Generate option on the main menu has two selections: Set DBMS and Generate Tables. The Set DBMS choice specifies the target DBMS. It also controls which data types are displayed in the column

Figure 3.8

Use Ctrl-A to select all text in the box, and Ctrl-C to copy it to a text editor.

```

DROP TABLE SaleItem;
DROP TABLE Inventory;
DROP TABLE Sale;
DROP TABLE ItemModel;
DROP TABLE Employee;
DROP TABLE Customer;
DROP TABLE Department;
DROP TABLE ItemCategory;

CREATE TABLE ItemCategory
(
    Category                NVARCHAR2(50),
    CategoryDescription     NVARCHAR2(50),
    CONSTRAINT pk_ItemCategory PRIMARY KEY (Category)
);

CREATE TABLE Department
(
    Department              NVARCHAR2(50),
    Dept_Description       NVARCHAR2(50),
    CONSTRAINT pk_Department PRIMARY KEY (Department)
);

CREATE TABLE Customer
(
    CustomerID              INTEGER,
    Cust_FirstName         NVARCHAR2(50),
    Cust_LastName          NVARCHAR2(50),
    Cust_Phone              NVARCHAR2(50),
    Cust_Email              NVARCHAR2(50),
    Cust_Address            NVARCHAR2(50),
    Cust_City               NVARCHAR2(50),
    Cust_State              NVARCHAR2(50),
    Cust_ZIP                NVARCHAR2(50),
    Cust_Gender             NVARCHAR2(50),
    CONSTRAINT ck_Customer_Cust_Gender CHECK (Upper(Gender) IN
('FEMALE', 'MALE', 'UNIDENTIFIED')),
    Cust_DateOfBirth        DATE,
    CONSTRAINT pk_Customer PRIMARY KEY (CustomerID)
);

```

Drop existing tables

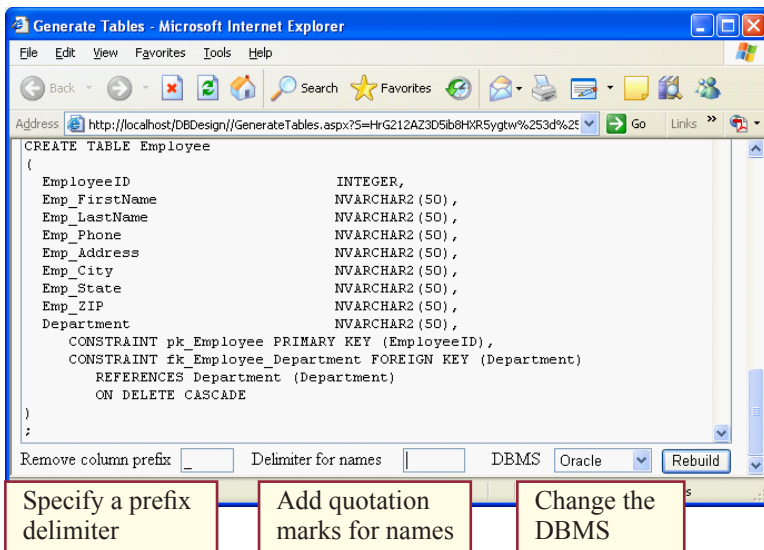
edit box. You should use the menu to set the target database to Oracle. Then run Generate/Generate Tables.

Your results will depend on which tables you have created, but Figure 3.8 provides an example. Notice that the system first writes commands to drop any existing version of the tables. Oracle does not have a way to replace tables, so the CREATE TABLE commands would crash if you had older versions with the same name. On the other hand, if you do not have older versions of the tables, the DROP statement will issue a warning message. You can ignore those messages.

The database design system automatically adds any CHECK constraints and DEFAULT value items you entered into the system for a column. It also creates the primary key constraints and defines any sequences used to generate key values. More importantly, it builds the foreign key constraints for all of the relationships. In fact, remember that with foreign key constraints, you have to create the referenced tables first (such as Department before Employee). The design system automatically sorts the tables and writes the CREATE TABLE statements in the proper order. It also writes the DROP TABLE statements in the reverse order. From this point, you can copy the text from the screen (Ctrl-A, Ctrl-C), start a text editor (WordPad), and paste the SQL statements into the editor. Then save the file with the SQL suffix. Start SQL Developer and run the new file. All of your tables will be created. You could also paste the commands directly into SQL Developer, but it is better to save them as a file so you have a backup copy.

As shown in Figure 3.9, the generator has some options that might be useful to you. Scroll to the bottom of your browser to see the three options and the Rebuild button. First, go back and look at the columns used in the Customer and Employee tables. Since they both represent people, those tables have columns for LastName and FirstName. To tell a customer LastName apart from an employee LastName in the right hand listing, the columns were originally named CustLastName and EmpLastName. But, when you create the tables, you will be able to identify them simply by the table in which they appear (Employee.LastName and Customer.LastName). So, it would be nice to drop the prefixes (Cust and Emp). You could

Figure 3.9



edit the table lists by hand and manually delete each prefix. Or, you can go back to the design screen and add an underscore (or any other separator) to each name. As you can see with the Employee table, you end up with column names similar to Emp_LastName. Now, you simply enter the underscore character (`_`) into the Remove column prefix box. When you click the Rebuild button, any characters found before the underscore will be removed from the name. They will remain within the design system; this change only applies to the generated names.

The second option is one you can consider, but it carries a cost in the future. By default, Oracle converts and stores all table and column names in uppercase letters. If you want mixed case names, you can enclose the names within double quotes (`"`) when you create the table. The Delimiter for names box provides a shortcut to making all of the changes by hand. Simply enter the double-quote character into the box and click the Rebuild button. Note that SQL Server and Access use square brackets, so you can also enter a square bracket character (`[`). However, before you decide you really like mixed case names, note that for every future SQL command, you will have to enclose the name in double quotes. You have not yet seen how many times you will have to type table and column names, but it is substantial. So, although mixed case names are easier to read, they might not be worth the additional work.

You can also change the target DBMS from this screen when you rebuild. This option is merely there to save you a step in case you forgot to set the correct

Action

Start the Enterprise Manager.

Click the Administration tab, then the Tables link under the Schema section.

Enter the columns for the Employee table, but name it Employee2.

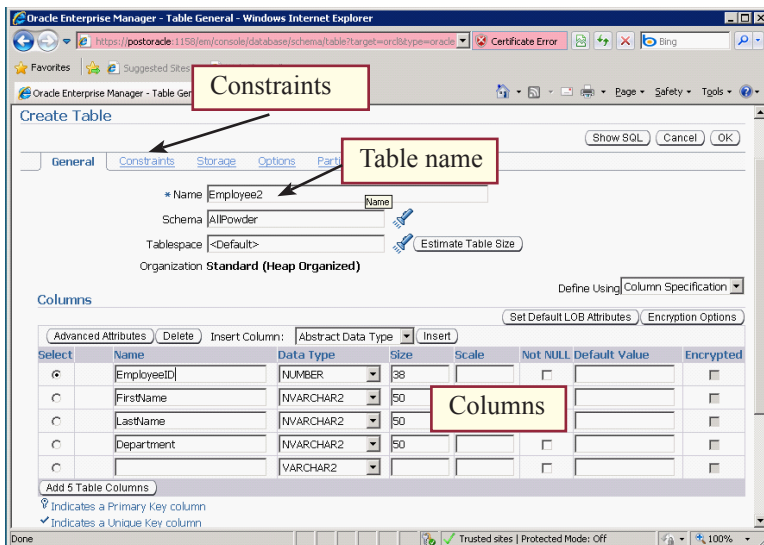
Create the primary key constraint.

Create the foreign key constraint.

Look at the SQL and create the table.

Delete the table.

Figure 3.10



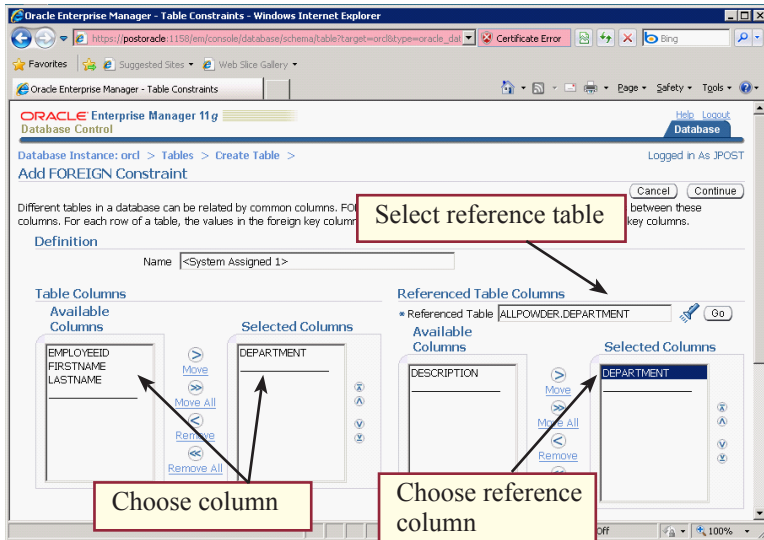


Figure 3.11

DBMS back in the design screen. The selected value is not saved with your design, so you can experiment and see that the other system use similar syntax.

Finally, note that Oracle supports many more options in the CREATE TABLE command. Most of them control where and how the table is stored. A DBA would use these options to adjust the performance of the database. However, they are highly specialized and not required, so you can worry about them later.



Activity: Create Tables with Enterprise Manager

Although the Enterprise Manager requires DBA privileges, it is worth examining some of the tools it provides to create and organize the database. If you do not have DBA permissions at the moment, you can just read this section for an overview. After logging into the Enterprise Manager (EM) (<http://yourserver:1158/em>), you will see the three main tabs: Performance, Administration, and Maintenance. To create or alter tables, you need to select the Administration tab. On that page, look for the Schema section and click the link for Tables. When you click the Create button, you can follow the prompts to create a new table.

As shown in Figure 3.10, you basically fill out a form to create the table. Make sure you provide the name for the table. Then enter each column name and select the appropriate data type. It is also easy to specify a default value for each column.

Creating the primary key and foreign key constraints takes a few more steps. Begin by clicking the Constraint tab for the table. You then select the type of constraint (Primary, Foreign, Check, and so on). For a primary key constraint, you need only select the columns that make up the primary key. The process is similar to the first step in the creation of a foreign key.

As shown in Figure 3.11, creating a foreign key requires several steps. First you select the columns in the main table that are the foreign key. For the sample Employee table, select the Department column and move it to the next window. The next step is to choose the table that is being referenced. In this case, you need to choose the Department table. The trick is to click the flashlight icon to ini-

tiate a search. A window pops up that shows you a list of available tables. Choose the department table and return to the main constraint screen. You will probably have to click the Go button to load the table columns for the selected table. Then choose the Department key column and move it to the selected window. You can now add more constraints or return to the column-list window by clicking the appropriate tab. As a DBA, one of the nice features of the EM is its ability to help you configure storage options for each table.

On the main table design page, you can click the Show SQL button to see the SQL CREATE TABLE command. When you are finished with the design, you should look at this code and copy it to a text file so you have it available in the future. Then you can click the OK button to execute the statement and create the table.

The visual approach to creating tables is useful—especially for beginners—because you do not have to memorize the CREATE TABLE syntax details. Just remember to copy the SQL statement and store it in a text file to make it easier to rebuild the table later. Eventually, as you become more familiar with SQL, you will find it easier to use straight SQL to create tables. Or, you will use a graphical designer that generates the SQL statements with even less effort.



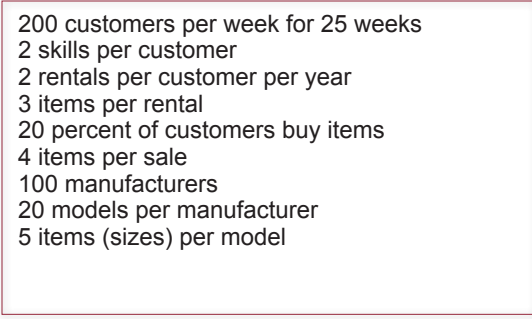
Activity: Estimate the Database Size

At some point, you need to estimate the size of the database project. Of course, any estimate at this early stage will be very rough. Your goal is not to be perfect, but to be able to categorize the overall project size. The information will help you identify the basic category of database server and perhaps narrow your choice of tools. In particular, it will help you determine how much disk space you need to purchase, and whether you will need more servers and faster processors. Note that

Action
Create a spreadsheet.
Enter table names as rows.
Add columns for: Bytes, Rows, Totals.
Calculate the bytes per table row.
Estimate the number of rows.
Compute the table and overall totals.

Figure 3.12

CustomerID	NUMBER(12)	8
LastName	NVARCHAR(50)	30
FirstName	NVARCHAR(50)	20
Phone	NVARCHAR(50)	24
Email	NVARCHAR(150)	100
Address	NVARCHAR(50)	50
State	NVARCHAR(50)	4
ZIP	NVARCHAR(15)	20
Gender	NVARCHAR(15)	20
DateOfBirth	Date	7
Average bytes per customer		283
Customers per week (winter)		*200
Weeks (winter)	*25	
Bytes added per year		1,415,000



- 200 customers per week for 25 weeks
- 2 skills per customer
- 2 rentals per customer per year
- 3 items per rental
- 20 percent of customers buy items
- 4 items per sale
- 100 manufacturers
- 20 models per manufacturer
- 5 items (sizes) per model

Figure 3.13

with Oracle, the database administrator has to set aside table space files to hold the data, so it helps to have some idea of the storage requirements early in the process.

To estimate the database size, you begin by estimating the size of each data table. You must already know which columns belong to each table. Figure 3.12 shows the process for the Customer table. Some of the column size estimates are straightforward. Look back to Chapter 2 for a reminder that a NUMBER data type can use 8 bytes of storage in Oracle. The text columns are a little trickier. For instance, although the database will allow up to 50 characters of text for the last name, almost no names will actually be that long. Instead, you need to estimate the average length of customer last names. You could use existing data, or perhaps a sample from a phone book. Perhaps an average last name is 15 characters long. But the DBMS stores text in Unicode format, which requires 2 physical bytes of storage for each character, so the average storage space needed for a last name is 30 bytes. Use a similar process to estimate the number of bytes needed to store an average row of customer data.

Next, you need to estimate how many new customers will arrive each year. In a real case, you could look at past records or talk with the expert users. Here, assume it is about 200 per week, but there are only 25 weeks of the ski season; so there are about 5,000 new customers a year. Multiplying the estimated number of customers by the size of an average row yields the initial data size of the Customer table to be about 1 million bytes.

You need to follow a similar process for all of the tables in the case. Figure 3.13 lists some of the basic assumptions you can use. You should build a spreadsheet that lists each table, the average number of bytes per row, the estimated number of rows, and the total estimated size for the table. There is still some flexibility in the final number, but your estimate should be around 5 to 6 megabytes. Remember that this is data for only one year. Also, additional space will be required for indexes, overhead, queries, forms, and reports. But even if the final number is closer to 20 megabytes, Oracle can easily handle this database on a PC-based server. If you look closely at the Oracle table designer in the Enterprise Manager, you will see a button that performs most of the estimation for you. It asks you to estimate the number of rows in the table, then it computes a quick estimate of the storage size required for that table. You still need to get good estimates of the number of rows in each table.

Exercises



Many Charms

Samantha and Madison want you to build the database for their charms sales. They emphasized that the system has to be easy to use. They also pointed out that a key element of their business is tracking all of the products and the various suppliers, then monitoring the costs so they can set their prices accurately. They are also concerned about monitoring how quickly their charms sell. They figure they will need to start with at least 200 basic charms, but most charms come in two sizes, along with the different metals and finishes. When asked, the women indicate they are uncertain how many customers they will have but would like to get at least 50 sales a week. Although some of the sales might be small, they hope to build a solid list of clients who return for new purchases on a monthly basis. To encourage return customers, they are thinking about offering some type of frequent-buyer program, where customers receive discounts or maybe a free charm, after purchasing a specified number of charms.

1. Define the final tables needed for this case.
2. Create the database.
3. Estimate the size of the database for one year of operation.



Standup Foods

Laura's business has been established for several years. Many of her clients are old customers, and she has a couple of thousand in her files—although some have gone out of business. Her business has grown considerably based on referrals from existing clients. She gets so many good comments and referrals, she is thinking that she needs to track which customers pass her name on to others so she can call them or send thank-you gifts. But, her more immediate concern is tracking employees. Over the course of a year, she has a relatively high turnover in some positions. Other employees have been with her for years. In total, she probably deals with 400 to 500 employees a year. Employees are rated after each job, and typically employees work 15 to 20 jobs a year for her. On average, employees tend to have three tasks per event. For instance, a driver will also be a server, and possibly also a busboy or dishwasher. They are evaluated on 10 items for each task they perform, as well as given an overall rating. Client food preferences are somewhat more complex, so Laura wants the capability to add free-form comments to cover extreme cases. For common elements, such as allergies to nuts, she wants to keep itemized lists—both for desired items and forbidden items. Some clients are easy going, but this is Hollywood, so many have long lists of items—often ranging to 50 or even up to 100 items.

1. Define the final tables needed for this case.
2. Create the database.
3. Estimate the size of the database for one year of operation.



EnviroSpeed

For good or bad, Tyler and Brennan have been busy. Their firm has been averaging four to five cleanups a week. Although there are not many permanent employees (fewer than 100), they have close associations with about 200 experts in various areas. All of these people need access to the environmental documents and other information. Additionally, about 400 crews around the world are called in to work on various problems. The crews consist of 10 to 20 people. Initially, experts contribute the most information. Sometimes an expert will contribute hundreds of pages of documents and comments. Once an incident is opened, most of the new data and the searches come from the emergency crews. Time schedules, environmental factors, and comments can arrive quickly from all of the crew members. Some of the notes are on paper and saved until the emergency is over, when clerks enter the basic data to the database. A typical incident can generate dozens of pages of notes and schedules from each crew member. Although there are hundreds of possible chemicals, the firm has found that only about 50 major chemicals are typically involved in critical incidents. One important aspect of this case is the need for experts and crew members to search through documentation based on key words. For example, crews will need to search for certain chemicals, possibly in combination with other chemicals, and often include the type of problem, such as water or road spill. Brennan estimates a typical document needs to include at least 20 keywords to identify the exact purpose of the document.

1. Define the final tables needed for this case.
2. Create the database.
3. Estimate the size of the database for one year of operation.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following instructions.

1. Finalize your database design.
2. Create the tables in the DBMS.
3. Estimate the amount of data that might be generated for one year.

Database Queries and SQL

Chapter Outline

Database Queries, 59

Case: All Powder Board and Ski Shop, 59

Lab Exercise, 60

All Powder Board and Ski Data, 60

Computations and Subtotals, 69

Exercises, 75

Final Project, 76

Objectives

- Create or import sample data into a database.
- Create basic queries to answer common business questions.
- Use joins to create multitable queries.
- Use queries to perform simple calculations.
- Answer business questions involving totals and subtotals.

Database Queries

Relational databases are designed to efficiently store data. Efficiency results in splitting the data into many tables, interconnected by the data. Consequently, you need a good query system to retrieve data. SQL is a powerful standard designed to perform several tasks in retrieving and manipulating data in relational database systems. Most modern systems implement some version of SQL. The catch is that the standard continues to evolve, and it takes time for the DBMS vendors to catch up. Also, vendors tend to include proprietary extensions to provide additional features. At one time, Oracle included a visually oriented QBE system, but as of 9i, it no longer exists as a standalone system. So you really need to learn and understand the straight-text SQL. The logic of SQL is the same as for QBE, but it can be cumbersome because you have to type more text. Also, the JOIN statements are a little more confusing in SQL.

Oracle has two related methods to enter SQL commands: (1) SQL Plus, (2) and SQL Developer. Both require you to type SQL statements and have limited or no graphical or QBE features. Of the two, SQL Developer has a better editor and is easier to use. The current (11g) version of SQL Plus runs from command mode with limited editing features. SQL Developer also displays a list of tables and columns in a navigation tree. The tree also shows the syntax for built-in functions.

There is one other important issue you need to know about Oracle SQL. You often need to issue a COMMIT command to ensure that your changes are written to the database. It is part of the transaction processing system that is explained in more detail in Chapter 7.

This chapter focuses on the data retrieval aspects of queries. SQL can also be used for data definition (e.g., CREATE TABLE), and for data manipulation (e.g., UPDATE and DELETE). These features and more complex queries are covered in Chapter 5. Once you learn the foundations of queries presented in this chapter, the other topics are easier to understand.

In any database, when you are writing queries, it helps to have a copy of the class (relationship) diagram handy. One of the more difficult aspects to creating a query is to find which tables hold the data you need. This problem is one of the reasons it is so important to label your tables and columns carefully when you create the database. Managers need to be able to identify the tables and columns that match the business questions. With dozens or even hundreds of tables with confusing or abbreviated names, it can be difficult to find the correct data.

Case: All Powder Board and Ski Shop

Before you can build queries, you need data in the tables. Even with a small number of tables, it is time-consuming to create reasonable data. You have to match the foreign keys across the relationships. For instance, it is straightforward to create basic customer data, although it would take a while to type in data for a thousand customers. Then, when you want sales data, you have to select CustomerID values from the existing list. You also have to create ski and board models, generate data for items with appropriate attributes, and then choose the proper ID values for the sales and rentals. In a typical business project, you can test the database with a few dozen examples, and then wait for the business to generate real data to analyze. In a class setting, it is better to use sample data. For that reason, sample data is available for the tables in the All Powder case. The one catch is that your tables might not contain exactly the same columns. This data was randomly generated

with specially built generators. You could edit the data to match your tables, but it is easier to delete (or rename) your tables, then run the build script that creates and loads the new tables.

Lab Exercise

All Powder Board and Ski Data

At this point, the main tables of your database should be similar to those in Figure 4.1, although several supporting tables have been removed from the figure. The Manufacturer, Customer, Sale, and SaleItem tables are common to most business databases. The Rental and RentItem tables simply mirror the sale aspects. The Inventory and ItemModel tables arose because of the characteristics of the board and ski products. To save time and effort, sample data files are provided on the Web site for each of these tables, plus the common supporting tables.



Activity: Import Data

Oracle supports importing data from flat files, but the process requires some advanced permissions. The scripts for this book use this method, and require that you have the role: CREATE ANY DIRECTORY. Another method of loading data is to write text files with hundreds (or thousands) of lines of INSERT INTO statements. This process is somewhat inefficient and relatively slow—because each row contains extra characters and each row is processed separately. It is reliable and works with any version of Oracle, but it takes time to build the INSERT INTO statements—even

Action

Download the initial All Powder database.

Start SQL Developer.

Drop or rename conflicting tables:

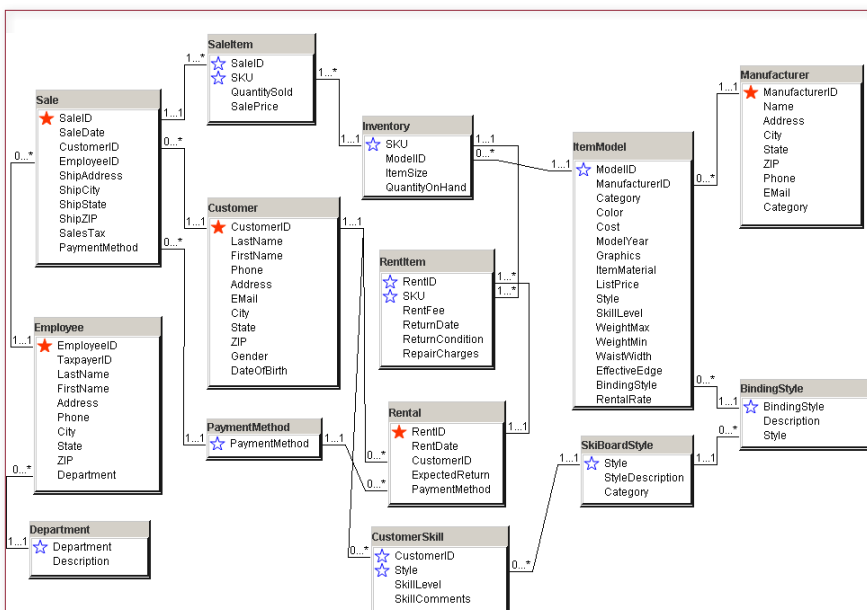
Start <path>\0DropAllPowderTables.sql.

Edit the csv_dir path in the 1Build... file.

Start <path>\1BuildOracleAllPowder.sql.

Wait a few minutes for the data to load.

Figure 4.1



with a program to generate them. The CSV files also might have to be stored on the Oracle server. A load version of the initial database exists using INSERT commands if you need to use that one instead of the main one.

In any case, you should download a copy of the Build statements that will create the initial All Powder database from the Web site. You should rename your existing files if you want to keep them. Then you should run the DropAllPowderTables.sql script to ensure that the script does not crash because of tables with duplicate names.

Copy all of the files in the BuildAllPowder folder to a directory on the computer running Oracle Developer. Edit the main file (1BuildOracleAllPowder.sql) in a text editor (such as Notepad). Change the name of the csv_dir so that it matches the location of your data files. Currently, it is set to c:\Books\OracleWorkbook. Save the file and close the editor. From within SQL Developer, run the modified file with: Start <location>1BuildOracleAllPowder.sql, where <location> is the full path name of the directory holding the file. For instance:

```
start c:\Books\ORacleWorkbook\1BuildOracleAllPowder.sql
```

Figure 4.2

```
Start D:\Books\Chapter04\BuildAllPowderCh04.sql
Starting...
Tables Created
Loading Employee...
Loading Manufacturer...
Loading ItemModel...
Loading SaleItem...
Loading Rental...
Analyzing Data...
Finished.
  COUNT(*)
    3953
1 row selected.
SELECT table_name FROM user_tables
WHERE table_name Not Like 'BIN%' ORDER BY table_name;

TABLE_NAME
CUSTOMER
CUSTOMERSKILL
DEPARTMENT
EMPLOYEE
INVENTORY
ITEMCATEGORY
ITEMMODEL
MANUFACTURER
PAYMENTMETHOD
PRODUCTCATEGORY
RENTAL
RENTITEM
SALE
SALEITEM
SKIBOARDSTYLE
SKILLLEVEL
16 rows selected.
```

After a few minutes, the tables should be created, the data copied, and the tables analyzed to improve performance. As shown in Figure 4.2, you can query the underlying metadata view to see if the tables were created, using the command:

```
SELECT table_name FROM user_tables
WHERE table_name Not Like 'BIN%' ORDER BY table_name;
```



Activity: Create Basic Queries

Creating a query requires that you translate a business question into a format the query system can process. Sometimes this step is straightforward; at other times it is difficult. It helps if you format your query in terms of the four main questions: (1) What do you want to see? (2) What do you know or what are the constraints? (3) What tables hold the data? (4) How are the tables connected? In Oracle, these questions are all entered using SQL text commands. As shown in Figure 4.3, you type these commands in a SQL Worksheet within SQL Developer. You enter SQL statements in the top half of the screen. When you click the Execute arrow button, the results of the query and any messages show up in the bottom window of the form. You can use the History button to retrieve earlier queries. You use standard mouse editing actions to edit the SQL statements. Queries that retrieve many rows are automatically paginated and data is displayed in columns.

Action

Run the query:

```
SELECT Category, ListPrice,
WeightMax, Color, Graphics
FROM ItemModel
WHERE Category='Board'
AND ListPrice<300
AND WeightMax>150;
```

If you know that a query is going to contain a large number of rows and you need to see only a few of them, you can use the built-in rownum to limit the results. Just include a WHERE (rownum < 20) or whatever count number you want to see. The rownum is displayed in a column on the left side.

Figure 4.3

The screenshot shows the Oracle SQL Developer interface. The main window displays a SQL query in the editor:

```
select *
from customer
where rownum < 3;
```

Below the editor, the 'Statement Output' and 'Query Result' panes are visible. The 'Query Result' pane shows the following data:

CUSTOMERID	LASTNAME	FIRSTNAME	PHONE	EMAIL	ADDRESS
1	0 Walk-in	(null)	(null)	(null)	(null)
2	1 Jones	Jack	111-222-3333	JonesJ202@msn.com	123 Main S

Annotations in the image include:

- Execute query**: Points to the green play button in the toolbar.
- Prior commands history**: Points to the 'History' button in the toolbar.
- SQL statement**: Points to the query text in the editor.
- Results**: Points to the 'Query Result' pane.

Another shortcut available in SQL Developer is that you can use the mouse to drag a table into the edit window. Choose the SELECT option and the developer will write the simple SELECT statement that contains a list of all of the columns from that table.

Action

Format the ListPrice column using the TO_CHAR function
 SELECT Category,
 to_char(ListPrice,'\$9999.00') As List_Price, ...

As you work through the queries in this lab, you will want to keep two things handy: (1) the list of tables in the database and (2) the syntax chart for SQL statements. Both of these are available in the navigation window on the left side of SQL Developer, but sometimes it helps to have a copy of the database diagram.

Begin with a straightforward query: Display the snowboards with a list price under \$300 for riders over 150 pounds. The potential buyer wants to know what color and graphics are available for boards that meet those conditions. The most difficult step in this query is to identify the table and columns that match the conditions. For example, snowboards are identified by the Category column in the ItemModel table. If you examine the data, you will see a “Board” entry for each item that is a snowboard. The list price, maximum weight, color, and graphics columns are also in the ItemModel table.

Figure 4.4 shows the basic query and the results. To create the query, first enter the main three SQL keywords on separate lines: SELECT, FROM, and WHERE. Now you can build the query simply by filling in the blanks after those keywords. Generally, determining what you want to see (SELECT) is straightforward, so enter the column names on that row: Category, ListPrice, WeightMax, Color, and Graphics. Next, enter the criteria given for the problem on the WHERE line: Category='Board' AND ListPrice<300 AND WeightMax>150. Note that text qualifiers (Board) have to be enclosed in single quote marks ('). Also remember to put the semicolon (;) at the end of the statement. Finally, check the column names

Figure 4.4

Question	Display snowboards with a list price under \$300 and max weight over 150 pounds.				
SQL	COLUMN Category Format A10 COLUMN Color Format A10 COLUMN Graphics Format A15 SELECT Category, ListPrice, WeightMax, Color, Graphics FROM ItemModel WHERE Category='Board' AND ListPrice < 300 AND WeightMax > 150;				
	CATEGORY	LISTPRICE	WEIGHTMAX	COLOR	GRAPHICS
	Board	292	188	Orange	Fade
	Board	263	181	Magenta	Geometric
	Board	262	179	Purple	Space
	Board	290	194	Blue	Abstract
	Board	294	158	Red	Sunrise
	Board	270	191	Yellow	Landscape
	Board	255	239	Red	Gothic
	Board	256	171	Magenta	Sunrise
	Board	283	226	Blue	Gothic
	Board	277	163	White	Gothic
	Board	259	223	Magenta	Linear

in the SELECT and WHERE statements and see which tables they fall in. In this case, all of them are in the ItemModel table, so simply enter that table name on the FROM line. Run the query to see the 11 boards that meet the conditions.

SQL Plus uses COLUMN commands to format output on the screen, but these are not used in the SQL Worksheet results. Instead, you have to format each column within the SELECT statement—typically by using the TO_CHAR function of Oracle.

As shown in Figure 4.5, the TO_CHAR formatting function can be used in any SQL statement. You can find additional formats and more details about the TO_CHAR command in the Oracle documentation. It is commonly used for currency and date data types.



Activity: Create and Test Multiple Boolean Conditions

Interpreting business questions can sometimes be difficult because of the ambiguity of natural languages. It is one of the reasons SQL remains so important. SQL requires you to specify exactly what you want to see and to write the conditions mathematically. Of course, these conditions can become relatively long when the business question is complex. Consider a customer who wants skis for jumping. She wants them made from composite materials, and the main color can be red or yellow. She does not want to spend more than \$300, but if they are red, she is willing to pay up to \$400.

Begin with a new query, and again recognize that all of the attributes are in the ItemModel table. Looking through the data, the first three conditions are straightforward: the Category is Ski, the ItemMaterial is Composite, and the Style is Jump. The colors appear to be straightforward, except that the choice is connected with Or. Whenever a query contains both And and Or conditions, you must be careful, so start with basic conditions and check the results as you go. Figure 4.6 shows the initial query with the three main conditions that must always hold (Ski, Jump, and Composite). Note that whenever an item is placed in quotation marks, Oracle treats it as case sensitive. Hence, you must type the condition values care-

Figure 4.5

The screenshot shows the Oracle SQL Developer interface. The main window displays the following SQL query:

```
SELECT Category,
to_char(ListPrice, '$9999.00') As List_Price,
WeightMax,
Color,
Graphics
FROM ItemModel
WHERE Category='Board'
AND ListPrice < 300
AND WeightMax > 150;
```

An arrow points from a red-bordered box labeled "Format currency" to the `to_char` function in the query. Below the query editor, the results are displayed in a table:

CATEGORY	LIST_PRICE	WEIGHTMAX	COLOR	GRAPHICS
1 Board	\$294.00	158	Red	Sunrise
2 Board	\$270.00	191	Yellow	Landscape
3 Board	\$255.00	239	Red	Gothic
4 Board	\$292.00	188	Orange	Fade
5 Board	\$263.00	181	Magenta	Geometric
6 Board	\$262.00	179	Purple	Space

The status bar at the bottom indicates "All Rows Fetched: 11 in 0.15 seconds" and "Formatting Succeeded".

fully. Since the rest of the SQL statement, such as the table name, is not in quotation marks, it is not case sensitive.

Now you can think about how to add the other two aspects of the question. Yellow skis are required to cost less than \$300, so what happens if you add both conditions to the query? Figure 4.7 shows the query and the results. Since all of the conditions are on the same Criteria row, all five must be true at the same time. So, the query returns only yellow skis, and only one row matches the price condition.

To see the red skis, you have to add the option of Red as a color, but you also have to establish the higher acceptable price for red skis. The solution is to use parentheses. Anytime you encounter a query that contains both And

and Or connectors, you will have to use parentheses to specify how the conditions are grouped. Remember from algebra that conditions inside the innermost parentheses are evaluated first. The key in this example is to group the color yellow

Action

Clear any existing SQL and results.

Enter the keywords SELECT, FROM, WHERE.

SELECT Category, Color, ItemMaterial, Style, ListPrice.

Enter ItemModel as the table on the FROM line.

Enter conditions: Category='Ski'
And Style='Jump' And
ItemMaterial='Composite'.

Run the query to ensure it works.

Add the conditions for Color='Yellow'
and ListPrice<300.

Test the query.

Add the conditions for Color='Red' and
ListPrice<400.

Add the correct parentheses.

Run the query and test it.

Figure 4.6

Question	List jumping skis made from composite materials.			
SQL	<pre>SET PageSize 24 COLUMN Category Format A10 COLUMN Color Format A10 COLUMN ListPrice Format \$9999.00 COLUMN ItemMaterial Format A15 COLUMN Style Format A10 SELECT Category, Color, ItemMaterial, Style, ListPrice FROM itemModel WHERE Category='Ski' AND ItemMaterial='Composite' AND Style='Jump';</pre>			
CATEGORY	COLOR	ITEMMATERIAL	STYLE	LISTPRICE
Ski	Red	Composite	Jump	\$223.00
Ski	Orange	Composite	Jump	\$386.00
Ski	Blue	Composite	Jump	\$229.00
Ski	Black	Composite	Jump	\$410.00
Ski	Turquoise	Composite	Jump	\$99.00
Ski	White	Composite	Jump	\$142.00
Ski	Yellow	Composite	Jump	\$357.00
Ski	Purple	Composite	Jump	\$177.00
Ski	Black	Composite	Jump	\$233.00
Ski	Yellow	Composite	Jump	\$340.00
Ski	Magenta	Composite	Jump	\$372.00
Ski	Turquoise	Composite	Jump	\$248.00
Ski	Red	Composite	Jump	\$294.00
Ski	Magenta	Composite	Jump	\$425.00
Ski	Red	Composite	Jump	\$137.00
Ski	Yellow	Composite	Jump	\$345.00
Ski	Red	Composite	Jump	\$431.00
Ski	Turquoise	Composite	Jump	\$119.00
Ski	White	Composite	Jump	\$178.00
Ski	Yellow	Composite	Jump	\$70.00

Question	List jumping skis, made from composite materials. And Yellow And ListPrice < 300			
SQL	SELECT Category, Color, ItemMaterial, Style, ListPrice FROM itemModel WHERE Category='Ski' AND ItemMaterial='Composite' AND Style='Jump' AND Color='Yellow' AND ListPrice<300;			
CATEGORY	COLOR	ITEMMATERIAL	STYLE	LISTPRICE
-----	-----	-----	-----	-----
Ski	Yellow	Composite	Jump	\$70.00

Figure 4.7

with its price condition and group the color red with its price condition. As shown in Figure 4.8, you also need to put parentheses around both of these new groups. If you leave out these last major grouping parentheses, the query will return yellow jumping skis or any red ski less than \$400. The final query shows that four skis match the conditions. Check them carefully to ensure that all conditions are met. Even if all of the skis in the result are acceptable, how do you know if the query found all of the matches? This question highlights one of the difficulties of any query language. The only way you know if the query is right is if you carefully build it step-by-step and test the individual steps. In this example, the first query was straightforward and ignored color and price constraints. It returned 20 matches, so the four matches returned by the final query seem like a reasonable number. In this case, the two sets are small enough that you can check the results by hand.



Activity: Use Multiple Tables in a Query

Relational databases require the tables to be carefully designed so that the DBMS can efficiently store large amounts of data. This process entails placing data into multiple tables. Consequently, a key feature of SQL is its ability to join the tables to make it easy to retrieve data from many tables with one query. Oracle 9i supports the SQL standard for joining tables. As it is the easiest to understand, it will be used here. The older Oracle syntax is shown at the end of this section because you will still see many queries in Oracle that use it.

To understand the join process, create a new query using just the Sale table. The objective is to find all of the sales in May that were made with a cash payment.

Figure 4.8

Question	List jumping skis, made from composite materials (Yellow And ListPrice < 300) OR (Red And ListPrice < 400)			
SQL	SELECT Category, Color, ItemMaterial, Style, ListPrice FROM ItemModel WHERE Category='Ski' AND ItemMaterial='Composite' AND Style='Jump' AND ((Color='Yellow' AND ListPrice<300) OR (Color='Red' AND ListPrice<400));			
CATEGORY	COLOR	ITEMMATERIAL	STYLE	LISTPRICE
-----	-----	-----	-----	-----
Ski	Red	Composite	Jump	\$223.00
Ski	Red	Composite	Jump	\$294.00
Ski	Red	Composite	Jump	\$137.00
Ski	Yellow	Composite	Jump	\$70.00

Figure 4.9 shows the initial query. Note the use of the Between clause to specify the month of May. Also observe the date format carefully. It is generally easiest to use the standard Oracle date format (dd-Mon-yyyy). If you want to use a different format, you need to use the TO_DATE function to specify the conversion method. For example, each date could be replaced with TO_DATE('01/05/2010', 'mm/dd/yyyy').

Observe that the query returns the CustomerID. But no one is going to memorize CustomerID numbers. Instead, you need to look up the matching customer names. If you look at the relationship diagram (part of it is shown in Figure 4.1), you find that the CustomerID and matching names are stored in the Customer table. Now you could take each of the ID values returned by the Sale query and create a new query on the Customer table and manually enter the values to find the names. However, the table JOIN command is much easier and more powerful to use.

In the SQL query, add the INNER JOIN line that adds the Customer table and specifies how it is connected to the Sale table: INNER JOIN Customer ON Sale.CustomerID = Customer.CustomerID. You will also have to add the table prefix to the CustomerID column in the SELECT statement: Sale.CustomerID. Finally, add the Customer LastName and FirstName columns to the SELECT phrase.

Figure 4.10 shows the basic query design. Once the tables are joined correctly, you can add any column to the other clauses. In this case, place the Customer LastName and FirstName columns in the SELECT clause. Run the query to see that the DBMS automatically looks up the names that match the ID values. If you want to double-check the lookup, you can add the CustomerID column from the Customer table and see that it matches the CustomerID values from the Sale table. Just be sure to specify the table name (Customer.CustomerID).

To see the power of the SQL joins, consider a slightly more challenging business question: Which customers bought Atomic skis in January or February? Note that Atomic is the name of a ski manufacturer. Before leaping into the SQL, it is best to think about the query and look at the relationship screen for a minute. As shown in Figure 4.11, begin with what you want to see: the names of the customers. These are in the Customer table. Now, what facts do you know? In this case, you are given the name of the manufacturer, the ItemModel.Category, and

Action

Start with a blank query.

Add SELECT, FROM, WHERE.

Set SaleID, SaleDate, CustomerID, and PaymentMethod.

Use only the Sale table.

Set the SaleDate between 01-May-2010 AND 31-May-2010.

Set PaymentMethod to Cash.

Run the query to test it.

Figure 4.9

Question	List customers (ID) with sales in May who paid with Cash.
SQL	<pre>SELECT SaleID, SaleDate, CustomerID, PaymentMethod FROM Sale WHERE SaleDate Between '01-May-2006' AND '31-May-2006' AND PaymentMethod='Cash';</pre>
	<pre> SALEID SALEDATE CUSTOMERID PAYMENTMETHOD ----- 1495 13-MAY-06 645 Cash 1304 07-MAY-06 1309 Cash 1356 02-MAY-06 314 Cash 1376 10-MAY-06 69 Cash </pre>

Question	List customers with sales in May who paid with Cash.				
SQL	<pre>SELECT SaleID, SaleDate, Sale.CustomerID, LastName, FirstName, PaymentMethod FROM Sale INNER JOIN Customer ON Sale.CustomerID = Customer. CustomerID WHERE SaleDate Between '01-May-2006' AND '31-May-2006' AND PaymentMethod='Cash';</pre>				
SALEID	SALEDATE	CUSTOMERID	LASTNAME	FIRSTNAME	PAYMENTMETHOD
1495	13-MAY-06	645	Alexander	Marvin	Cash
1304	07-MAY-06	1309	Pratt	Adrian	Cash
1356	02-MAY-06	314	Rich	Manuel	Cash
1376	10-MAY-06	69	Forbes	Horace	Cash

Figure 4.10

the range for the SaleDate. You should also begin writing down the tables you need to provide these facts: Customer, Sale, ItemModel, and Manufacturer so far. When you examine the relationships for the database, you will see that these four tables are not enough—they do not connect together. You will also need the SaleItem and Inventory tables.

Figure 4.12 shows the final query in Design view. Notice the large number of tables involved. But, you need to

verify that each connection is correct for the specific problem. Once the tables have been selected and joined, you can quickly place the columns you need on the query grid, and then enter the desired conditions. Running the query reveals the two people who meet the desired conditions. The join statements are the key to creating this query. Begin with one table, then add each new table after an INNER JOIN command. Be sure to specify the table links using a collection of ON conditions. Once the tables and links have been defined, you can use columns from any

Action

- Add the INNER JOIN line after FROM.
- Add the Customer table.
- Add the join condition: ON Sale.
CustomerID = Customer.CustomerID.
- Change to Sale.CustomerID on the SELECT statement.
- Add Customer LastName and FirstName to the SELECT statement.
- Run the query to test it.

Figure 4.11

Which customers bought Atomic skis in January or February?	
What do you want to see?	Customer names, SaleDate
What do you know?	Manufacturer name, SaleDate range, Category is Ski
What tables are involved? How are they joined?	Customer ... Sale ... ItemModel, Manufacturer

SELECT LastName, FirstName, SaleDate
 FROM Customer, ..., Sale, ..., ItemModel, Manufacturer
 INNER JOIN ...
 WHERE Manufacturer.Name="Atomic"
 AND Sale.SaleDate BETWEEN 1/1/2004 And 2/29/2004
 AND ItemModel.Category="Ski"

of the tables. Just remember that if a column by the same name exists in more than one table, you refer to that column with its full Table.Column name.

INNER JOIN commands can seem daunting at first. The key is to build them in pairs. Start with one table in the FROM command, then identify a table that can be joined to the first one. For example, FROM Manufacturer INNER JOIN ItemModel; then write the ON statement that specifies which columns are connected: ON Manufacturer.ManufacturerID = ItemModel.ManufacturerID. Move on by finding a table that has something in common with one of the existing tables, add the INNER JOIN newtable command and the ON statement. Continue until the statement contains all of the tables needed to answer the question.

Older Oracle queries are based on the older SQL syntax. Join conditions represent one of the greatest differences in this syntax. To see the difference, the Sale/ Customer query will be rebuilt. Figure 4.13 shows the difference. Begin the query with the SELECT, FROM, and WHERE clauses. Enter the columns to be displayed, then the date condition in the WHERE clause. List the Sale and Customer tables in the FROM clause separated by a comma. Finally, add the join condition (Sale.CustomerID = Customer.CustomerID) to the WHERE clause. There are no INNER JOIN or ON statements. When you run the query, you should receive the same results as earlier.

Computations and Subtotals



Activity: Compute Values with Queries

In general, it does not make sense to store some columns in the database. In particular, the DBMS query system has the ability to perform common calculations. Figure 4.14 shows how the query system can easily calculate the profit margin for each item. In this case, the table holds the item's list price and the acquisition cost. The profit is simply the difference between the list price and the cost. In the SELECT clause you enter the calculation and give it a name using the AS key-

Figure 4.12

Question	Which customers bought Atomic skis in January or February?			
SQL	<pre> COLUMN LastName Format A10 COLUMN FirstName Format A10 COLUMN Category Format A10 SELECT LastName, FirstName, ItemModel.Category, Name, SaleDate FROM Manufacturer INNER JOIN ItemModel ON Manufacturer.ManufacturerID = ItemModel.ManufacturerID INNER JOIN Inventory ON ItemModel.ModelID = Inventory.ModelID INNER JOIN SaleItem ON Inventory.SKU = SaleItem.SKU INNER JOIN Sale ON SaleItem.SaleID = Sale.SaleID INNER JOIN Customer ON Sale.CustomerID = Customer.CustomerID WHERE ItemModel.Category = 'Ski' AND Name='Atomic' AND SaleDate BETWEEN '01-Jan-2006' AND '28-Feb-2006'; </pre>			
LASTNAME	FIRSTNAME	CATEGORY	NAME	SALEDATE
-----	-----	-----	-----	-----
Patterson	Gene	Ski	Atomic	15-FEB-06
Mahoney	Francis	Ski	Atomic	23-JAN-06

Question	List customers (ID) with sales in May paid with Cash. (Older syntax.)																														
SQL	<pre>SELECT SaleID, SaleDate, Sale.CustomerID, LastName, FirstName, PaymentMethod FROM Sale, Customer WHERE Sale.CustomerID = Customer.CustomerID AND SaleDate BETWEEN '01-May-2006' AND '31-May-2006' AND PaymentMethod='Cash';</pre>																														
	<table border="1"> <thead> <tr> <th>SALEID</th> <th>SALEDATE</th> <th>CUSTOMERID</th> <th>LASTNAME</th> <th>FIRSTNAME</th> <th>PAYMENTMETHOD</th> </tr> </thead> <tbody> <tr> <td>1495</td> <td>13-MAY-06</td> <td>645</td> <td>Alexander</td> <td>Marvin</td> <td>Cash</td> </tr> <tr> <td>1304</td> <td>07-MAY-06</td> <td>1309</td> <td>Pratt</td> <td>Adrian</td> <td>Cash</td> </tr> <tr> <td>1356</td> <td>02-MAY-06</td> <td>314</td> <td>Rich</td> <td>Manuel</td> <td>Cash</td> </tr> <tr> <td>1376</td> <td>10-MAY-06</td> <td>69</td> <td>Forbes</td> <td>Horace</td> <td>Cash</td> </tr> </tbody> </table>	SALEID	SALEDATE	CUSTOMERID	LASTNAME	FIRSTNAME	PAYMENTMETHOD	1495	13-MAY-06	645	Alexander	Marvin	Cash	1304	07-MAY-06	1309	Pratt	Adrian	Cash	1356	02-MAY-06	314	Rich	Manuel	Cash	1376	10-MAY-06	69	Forbes	Horace	Cash
SALEID	SALEDATE	CUSTOMERID	LASTNAME	FIRSTNAME	PAYMENTMETHOD																										
1495	13-MAY-06	645	Alexander	Marvin	Cash																										
1304	07-MAY-06	1309	Pratt	Adrian	Cash																										
1356	02-MAY-06	314	Rich	Manuel	Cash																										
1376	10-MAY-06	69	Forbes	Horace	Cash																										

Figure 4.13

word: ListPrice-Cost AS Profit. Notice that the query is sorted by Category and ListPrice. Simply add an ORDER BY clause at the end of the command with the columns you want sorted. The DESC option specifies a descending order.

Calculations written in this form are always performed on data on the same row. It does not calculate across rows. You can use the standard mathematical operators (add, subtract, divide, and multiply). You can also use several standard functions built into Oracle. Figure 4.15 shows some of the commonly used functions. Most are straightforward, but the date functions require a little explanation and practice. The TO_CHAR function enables you to specify detailed formats for date and numeric columns. This list is only a tiny fraction of the functions available in Oracle. The Oracle documentation contains a complete list with explanations and examples.

To illustrate the power of some of the date functions, create a new query using the Sale table and display the SaleID and SaleDate columns. Now, as shown in Figure 4.16, add a new column TO_CHAR(SaleDate, 'yyyy-mm') AS SaleMonth. Be sure to enter the quoted format correctly—it controls the way the date will be converted to character format and displayed. In this case, it will display the four-digit year, followed by a two-digit number for the month. You often want to format months in this way to ensure that they sort correctly. The TO_CHAR function has many options, and you can consult the Oracle Help documentation for details. Search for TO_CHAR or Format Models in the SQL Reference book. Later, you will see that this date conversion is useful for computing subtotals. By formatting the sale date as year and month, you can easily compute the total sales by month—a feature that is commonly requested by business managers.

SQL automatically performs data arithmetic with days. Adding or subtracting a number from a date results in a new date that is different by the specified number of days. Figure 4.17 shows how easy it is to add 30 days to a SaleDate to produce a common billing late date. Notice that the date arithmetic is correct in that it automatically handles months, years, and even leap years. If you want to add or subtract in increments

Action

- Create a new query using only the ItemModel table.
- In the SELECT row, add a new pseudo column to compute ListPrice-Cost As Profit.
- Add the ORDER BY line to sort by Category and List Price descending.
- Use the WHERE clause to limit the number of rows returned.
- Run the query.

Question	Compute the profit (Price - Cost) for items with price over \$575.		
SQL	<pre>SELECT Category, ItemMaterial, ListPrice, ListPrice-Cost As Profit FROM ItemModel WHERE (ListPrice > 575) ORDER BY Category, ListPrice DESC;</pre>		
CATEGORY	ITEMMATERIAL	LISTPRICE	PROFIT
Board	Wood	\$649.00	\$227.15
Board	Wood	\$647.00	\$226.45
Board	Wood	\$646.00	\$226.10
Board	Wood	\$644.00	\$225.40
Board	Fiberglass	\$642.00	\$224.70
Board	Wood	\$642.00	\$224.70
Board	Composite	\$633.00	\$221.55
Board	Wood	\$633.00	\$221.55
Board	Fiberglass	\$629.00	\$220.15
Board	Composite	\$626.00	\$219.10
Board	Composite	\$613.00	\$214.55
Board	Composite	\$608.00	\$212.80
Board	Wood	\$582.00	\$203.70
Board	Composite	\$579.00	\$202.65
Board	Composite	\$576.00	\$201.60

Figure 4.14

other than days, you need to use Oracle's `ADD_MONTHS` function. To subtract dates in terms of months, use the `MONTHS_BETWEEN` function. Both the day and month arithmetic can use fractional values. For example, you could add 1.5 months to a date. You will often see fractional values if you subtract a date from today's date, which is given by `SYSDATE`. Since `SYSDATE` also includes the time of day, you will get noninteger results. If you only want the integer portion, you can use the `Floor` or `Round` functions. The `Floor` function truncates fractional values by throwing away all digits to the right of the decimal point. The `Round` function performs standard rounding to the specified decimal place.

Figure 4.15

Lower	To lowercase
Length	Length/number of characters
Substr	Get substring
Trim	Remove leading and trailing spaces
Upper	To uppercase
SYSDATE	Current date
ADD_MONTHS	Add days, months, years to a date
MONTHS_BETWEEN	Subtract two dates
TO_CHAR	Highly detailed formatting
TO_DATE	Format dates
SYSDATE	Current date and time
Abs	Absolute value
Cos	Cosine, all common trig functions
Floor	Integer, drop decimal values
Round	Round-off

Question	Convert sale date into the year and month.
SQL	<pre>SELECT SaleID, SaleDate, TO_CHAR(SaleDate, 'yyyy-mm') AS SaleMonth FROM Sale WHERE rownum < 15;</pre>
<pre> SALEID SALEDATE SALEMON ----- 1121 19-NOV-06 2006-11 1122 26-NOV-06 2006-11 1123 13-JUL-06 2006-07 1124 20-APR-06 2006-04 1125 07-NOV-06 2006-11 1126 18-AUG-06 2006-08 1127 16-JUN-06 2006-06 1128 08-JUN-06 2006-06 1129 28-JUN-06 2006-06 1130 11-SEP-06 2006-09 1131 16-NOV-06 2006-11 1132 04-JAN-06 2006-01 1133 26-MAY-06 2006-05 1134 16-DEC-06 2006-12 </pre>	

Figure 4.16



Activity: Calculate Totals and Subtotals

Business managers often need to compute totals across rows of data. SQL provides several aggregation functions to perform these tasks. The most commonly used functions are Sum, Average, and Count. Of the three, the Count function can be the most confusing. Just remember that it simply counts the number of rows, while Sum adds up the numbers with-

Action

Create a new query.

Use only the Sale table.

SELECT SaleID and SaleDate.

Add 30 days to the SaleDate to get LateDate.

Use ADD_MONTHS to add one month to the SaleDate to get SaleMonth.

Run the query.

Figure 4.17

Question	Compare date calculations by day and month arithmetic.
SQL	<pre>SELECT SaleID, SaleDate, SaleDate+30 As LateDate, ADD_MONTHS(SaleDate,1) As LateMonth FROM Sale WHERE rownum < 15;</pre>
<pre> SALEID SALEDATE LATEDATE LATEMONTH ----- 1121 19-NOV-06 19-DEC-06 19-DEC-06 1122 26-NOV-06 26-DEC-06 26-DEC-06 1123 13-JUL-06 12-AUG-06 13-AUG-06 1124 20-APR-06 20-MAY-06 20-MAY-06 1125 07-NOV-06 07-DEC-06 07-DEC-06 1126 18-AUG-06 17-SEP-06 18-SEP-06 1127 16-JUN-06 16-JUL-06 16-JUL-06 1128 08-JUN-06 08-JUL-06 08-JUL-06 1129 28-JUN-06 28-JUL-06 28-JUL-06 1130 11-SEP-06 11-OCT-06 11-OCT-06 1131 16-NOV-06 16-DEC-06 16-DEC-06 1132 04-JAN-06 03-FEB-06 04-FEB-06 1133 26-MAY-06 25-JUN-06 26-JUN-06 1134 16-DEC-06 15-JAN-07 16-JAN-07 </pre>	

in a row. The challenge is to identify when you need to use Count instead of Sum.

The Sum function is straightforward. For example, how much sales tax does the company owe to the state of California? Begin by creating a new query based on the Sale table, because it has the ShipState and SalesTax columns. As a criterion for ShipState, enter the CA abbreviation for California. Ignoring totals for the moment, run the query, and you should see two columns: each row

will have CA in the state, and a value for the SalesTax. To compute the total, return to SQL. Remove the SaleState from the SELECT statement and add the Sum function around the SalesTax: `Sum(SalesTax) AS SumOfSalesTax`. Figure 4.18 shows the total you should receive when you run the query. Why was it important to run the query first without the total? Because the total shows you only one number. How do you know the number is correct? You should always run a straight retrieval to ensure that the correct rows are being selected before you perform calculations on them. Of course, most aggregation queries will also use multiple tables—which makes it even more important that you check the detail rows first.

To understand some of the power of SQL, what if you want to see the total tax owed to each state? Of course, it would be possible to edit the CA condition and replace it with each state, but there is an easier way. As shown in Figure 4.19, start a new query the same way as the last one. Use the Sale table and select the ShipState and SalesTax columns, but do not specify any limiting conditions. Use the Sum function to total the SalesTax column and be sure to set the alias name. Do not include a WHERE statement. The tricky part is the next line: Add a GROUP BY clause at the end of the command. Tell it to compute the totals for each state with `GROUP BY ShipState`. When you run the query, you will get a list of all of the states with sales followed by the total sales tax collected for that state. Of course, you could compute the average or count the number of items in a group just as easily. In fact, you can compute multiple functions at the same time, just by including multiple copies of the desired column and selecting a different aggregation function.

For practice, you should compute the total value of sales to customers in Colorado (the state code is CO). Create a new query and add the Sale and SaleItem tables. Use the ShipState column from the Sale table. To compute the total value of the actual sale is slightly trickier. You need to multiply the QuantitySold by

Action

Create a new query.

Add the Sale table.

```
SELECT ShipState and SalesTax
```

```
WHERE ShipState = 'CA'.
```

Run the query.

Verify the correct states are displayed.

Remove ShipState from SELECT.

```
SELECT Sum(SalesTax) AS
```

```
SumOfSalesTax.
```

Run the query.

Figure 4.18

Question	Compute the sales tax total for California.
SQL	<code>SELECT Sum(SalesTax) AS SumOfSalesTax FROM Sale WHERE ShipState='CA';</code>
<pre>SUMOFSALESTAX ----- 5332.11</pre>	

Question	Compute the sales tax total for each state.																																				
SQL	COLUMN ShipState Format A10 SELECT ShipState, Sum(SalesTax) AS SumOfSalesTax FROM Sale GROUP BY ShipState'																																				
<pre>SHIPSTATE SUMOFSALESTAX ----- -</pre> <table> <tbody> <tr><td>AK</td><td>0</td></tr> <tr><td>AL</td><td>784.77</td></tr> <tr><td>AR</td><td>313.25</td></tr> <tr><td>AZ</td><td>510.93</td></tr> <tr><td>CA</td><td>5332.11</td></tr> <tr><td>CO</td><td>347.48</td></tr> <tr><td>CT</td><td>254.38</td></tr> <tr><td>DC</td><td>285.95</td></tr> <tr><td>DE</td><td>0</td></tr> <tr><td>FL</td><td>1404.34</td></tr> <tr><td>GA</td><td>470.61</td></tr> <tr><td>HI</td><td>175.49</td></tr> <tr><td>IA</td><td>164.01</td></tr> <tr><td>ID</td><td>330.12</td></tr> <tr><td>IL</td><td>1200.57</td></tr> <tr><td>IN</td><td>952.35</td></tr> <tr><td>KS</td><td>302.96</td></tr> <tr><td colspan="2"><i>(other states not shown)</i></td></tr> </tbody> </table>		AK	0	AL	784.77	AR	313.25	AZ	510.93	CA	5332.11	CO	347.48	CT	254.38	DC	285.95	DE	0	FL	1404.34	GA	470.61	HI	175.49	IA	164.01	ID	330.12	IL	1200.57	IN	952.35	KS	302.96	<i>(other states not shown)</i>	
AK	0																																				
AL	784.77																																				
AR	313.25																																				
AZ	510.93																																				
CA	5332.11																																				
CO	347.48																																				
CT	254.38																																				
DC	285.95																																				
DE	0																																				
FL	1404.34																																				
GA	470.61																																				
HI	175.49																																				
IA	164.01																																				
ID	330.12																																				
IL	1200.57																																				
IN	952.35																																				
KS	302.96																																				
<i>(other states not shown)</i>																																					

Figure 4.19

the SalePrice from the SaleItem table then compute its sum. To be safe, first do the multiplication and check your progress. Create the formula on the SELECT row with the command: QuantitySold * SalePrice AS SaleTotal. To select the state, enter 'CO' as the criteria in the WHERE clause. Run the query and check the results to see if they make sense. You might want to list the QuantitySold and SalePrice separately, and then use a calculator or spreadsheet to verify some of the calculations. Returning to the SQL, you need to compute the total. As shown in Figure 4.20, simply add the Sum function and place the parentheses around the multiplied values.

Action

Create a new query.

Use the Sale table.

Select columns: ShipState and Sum(SalesTax) AS SumOfSalesTax.

Add a row at the bottom: GROUP BY ShipState.

Run the query.

There is one more trick you need to learn before finishing this lab. You need to be able to save a query so that you can use it in other queries or reports. In Oracle, a saved query is called a View. Figure 4.21 shows how to save a query as a view. Using the query you just finished, simply add one line at the top, CREATE VIEW ColoradoSales AS, and run this query. You now have a view called ColoradoSales

Figure 4.20

Question	Compute the total value of all sales to Colorado.
SQL	SELECT Sum(QuantitySold*SalePrice) As SaleTotal FROM Sale INNER JOIN SaleItem ON Sale.SaleID = SaleItem.SaleID WHERE Sale.ShipState='CO';
<pre>SALETOTAL ----- 4964</pre>	

that performs the SELECT statement. To test it, clear the SQL window and create the simple query: `SELECT * FROM ColoradoSales`. Run this query and it will execute the stored query to compute and display the total sales in Colorado. You can delete views that you create with the DROP command: `DROP VIEW ColoradoSales`.

Exercises



Crystal Tigers

Enter sample data for the Crystal Tigers service club database. You can make up data, but remember that it has to be consistent. You might want to share data with other students so that everyone has a larger database to work with. Then create queries to provide the following business information.

1. List all of the members who have been president of the organization.
2. List the charities for which the club has raised more than \$1,000.
3. Pick an event and list all of the members who worked at that event.
4. Count the number of events and the amount of money raised for each charity.
5. List the total number of service hours provided in the latest year.
6. List the number of service hours provided by each member.
7. List the members who have held the most number of officer positions.



Capitol Artists

Enter sample data for the Capitol Artists business. You can create random data, but remember that it has to be consistent. You might want to share data with other students so that everyone has a larger database to work with. Then create queries to provide the following business information.

1. Pick a date and an employee and list all of the tasks by that person on that date.
2. List all of the tasks performed for a specific job (e.g., Job #1173).
3. List all of the client jobs that had active tasks on a specific date.
4. Count the number of meetings held regarding one client (pick any client).
5. List the employees who have attended the most number of meetings.
6. Pick a job and compute the amount of money billed (hours * rate).
7. List the clients in order of the ones that have provided the greatest revenue (billing + expenses).



Offshore Speed

Enter sample data for the Offshore Speed company. You can create random data, but remember that it has to be consistent. You might want to share data with other students so that everyone has a larger database to work with. Then create queries to provide the following business information. If you have not created data that matches these questions, either add more data, or change the query to match your

data. For instance, if you do not have any sales of propellers, pick a category of item that you have sold several times.

1. Pick a month and list all of the customers who purchased propellers (Category).
2. List all of the parts sold on a particular day.
3. What is the most expensive steering wheel we have sold?
4. List the manufacturers sorted by the number of parts we sell from each one.
5. List the employees to identify the best salespeople in terms of value.
6. List the brands of boat for which we sell the most oil pumps (Description).
7. For a given order, compute the total value of the order and the sales tax, assuming a 6 percent tax rate.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks.

1. Create a few rows of sample data for all of the tables.
2. Identify at least five business questions that a manager would commonly ask and provide the queries to answer those questions. At least two of the questions should involve subtotals or averages.
3. Exchange three business questions with other students in your class and write the queries for the questions you receive.

Advanced Queries

Chapter Outline

Advanced Database Queries, 78

Case: All Powder Board and Ski Shop, 79

Lab Exercise, 79

All Powder Board and Ski Data, 79

SQL Data Definition and Data Manipulation, 90

Exercises, 96

Final Project, 98

Objectives

- Create more complex SELECT queries using subqueries.
- Understand the role of INNER and LEFT joins.
- Create theta joins using inequalities to match categories.
- Use a UNION statement to merge rows of data.
- Use DDL to CREATE and DROP tables.
- Use DML to INSERT, UPDATE, and DELETE data.

Advanced Database Queries

SQL is a powerful language. For many queries, you will not need the full power of SQL, but some seemingly innocent business questions can be tricky to answer. In these cases, you need some additional capabilities. Some of these capabilities can be challenging to understand, but if you follow the examples carefully, you should be able to use the ideas to create similar queries in the future.

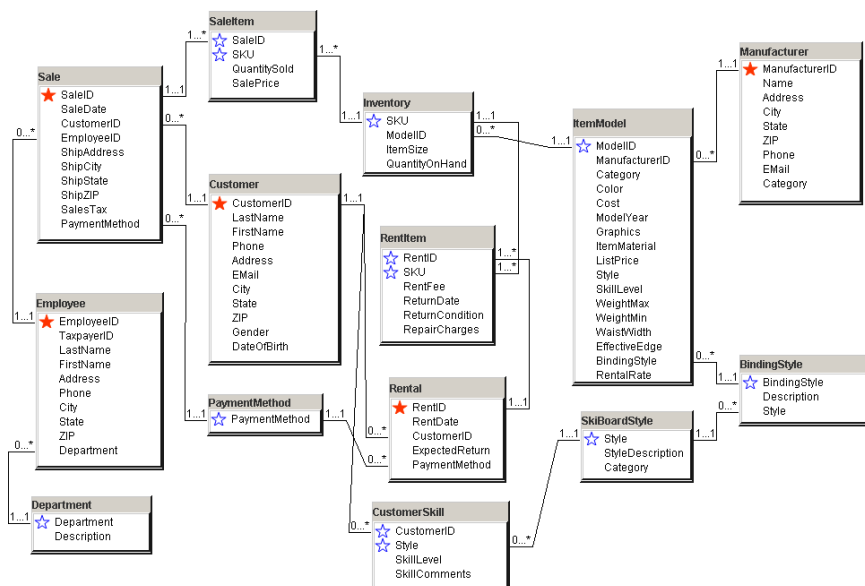
Subqueries are one of the more interesting features of SQL. A subquery is a query that calls a second query to obtain additional data. Instead of looking up a second set of numbers yourself, you can add a second query to do the work automatically.

Joins offer other powerful options. Joins are commonly used as a lookup link between tables, making it easy for you to build a query that uses data from multiple tables. However, joins have several options to help you answer even more complex questions. It is especially important that you understand the difference between inner and outer joins.

One of the strengths of SQL is that it operates on sets of data. Instead of thinking in terms of individual rows, you can concentrate on collections of rows that meet specified conditions. SQL offers some interesting set-operation commands that provide detailed control over rows of data. For example, the UNION statement combines rows of data from multiple SELECT statements.

Advanced queries generally rely exclusively on text-based SQL. Even if you have a visual QBE system available, it is much safer to use straight text to create difficult queries. One of the most dangerous aspects of any query is that the system will almost always return some type of data. You need to make sure the system is returning the correct data by ensuring that the query is actually asking for exactly what you think it is asking. Most of the data definition statements (such as CREATE TABLE, INSERT, DELETE, and UPDATE) will often be stored in text files that can be run as separate batches later to accomplish some larger task. Just remember to test all of them first.

Figure 5.1



Case: All Powder Board and Ski Shop

As the queries become more complex, it is better to work from a common set of data. Figure 5.1 shows the primary tables for the All Powder Board and Ski Shop. Your tables and sample data should be very close to these tables. Note that several supporting tables are not displayed in this diagram, but you will also need those in your database. As explained in Chapter 4, you can import the sample data to these tables. If you add more data, your query results may be slightly different from the ones shown in this chapter. While the query is more important than the actual results, the results are useful to help you decide if you have constructed the query properly.

One of the greatest challenges with any database query is that most queries return values, but they might not be answers to the question you thought you were asking. You must learn to carefully build the queries and test each intermediate step so that you can be sure the final result is an accurate answer to the question being asked.

Lab Exercise

All Powder Board and Ski Data

Subqueries are used to create a second (or more) query to look up additional data that can be used in the primary query. The value is often used within a WHERE clause to make comparisons in more depth. For example, Katy, the manager, wants to identify the best customers of the shop. In particular, she would like to know which customers have placed the most sales. You could just give her the complete list of customers and the sales made by each. However, eventually this list would be too long. Instead, she wants a list that displays the customers whose total purchases are larger than the average number of purchases per customer. Although the business question is reasonable, this question is slightly tricky because you have to build the query in pieces. Note that the WHERE clause restriction is used to temporarily reduce the number of rows returned.

Figure 5.2

Question	Compute total purchases by customer.		
SQL	<pre>SELECT Customer.CustomerID, LastName, FirstName, Sum(QuantitySold*SalePrice) As SalesValue FROM Customer INNER JOIN Sale ON Customer.CustomerID=Sale. CustomerID INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID WHERE Customer.CustomerID<41 GROUP BY Customer.CustomerID, LastName, FirstName;</pre>		
CUSTOMERID	LASTNAME	FIRSTNAME	SALESVALUE
7	Rice	Charlotte	94
40	Gentry	Arturo	1690
21	Jantzen	William	477
32	Hansen	Adam	845
19	Bell	Leslie	1110
22	Reynolds	Connie	692
33	Richmond	Hershel	323
18	Embry	Jahala	252



Activity: Create a Subquery

The first step in the query is to recognize that you need to compute total sales by customer. The phrase “by customer” is an indication that you need to compute subtotals using the `GROUP BY` clause. Figure 5.2 shows the initial query that computes these subtotals. Of course, it lists the sales for every customer, and Katy only wants the sales of greater than average amount. But this query is an important step and needs to be saved as `CustomerSales`: `CREATE VIEW CustomerSales AS SELECT ...`. Make sure you remove the `WHERE` clause constraint.

The next step is to use this first query to compute the average amount of sales for customers. This computation is straightforward. You simply build a new query using `CustomerSales` as the only table, and calculate the average of the sales column. Figure 5.3 shows the basic query and the result based on the current data. Notice that the SQL is straightforward. In this case, the SQL is critical for the next step. It is not necessary to save this query, but you might want to leave the SQL window open for the final step.

The last step is to create a new query that answers the overall question to determine which customers spend more than average. The new query will also be based on the `CustomerSales` query created in the first step, so just add that query. This time, select the `LastName`, `FirstName`, and `SalesValue` columns. If you ran the query at this point, you would get the same results as in the first query. Instead, you want to add a criterion to only display the customers with a `SalesValue` greater than the average. The simple approach is to enter the value 942.11 as a condition in the query. Although this approach works this time, it does not work very well over time. It would require the owner to run the average query first, then copy the value into the Design view of the main query. It makes more sense to automate the entire process. So instead of entering the actual number as the condition, you need to enter the subquery calculation within parentheses. Figure 5.4 shows the final query that you can give to Katy. Notice that it is sorted in descending order by `SalesValue` so the customers with the largest total purchases are listed at the top. Also, always remember to put the subquery inside parentheses—otherwise the query will not run at all. If you want to save some typing and reduce errors, you should create the subquery first in a separate query to test it. When it is correct, you can copy the SQL statement and paste it into the `WHERE` clause for the final query. Again, remember to add the parentheses around the subquery.

Action

Create a new query.

Tables: `Customer`, `Sale`, `SaleItem`.

Columns: `CustomerID`, `LastName`, `FirstName`, `Sum(QuantitySold*SalePrice)`
AS `SalesValue`

Group By the other columns.

Run the query.

Save as a view `CustomerSales`.

Create new query.

Table: `CustomerSales` query.

`SELECT Avg(SalesValue) ...`

Run the query.

Figure 5.3

Question	Compute average total purchases for customers.
SQL	<code>SELECT Avg(SalesValue) As AvgOfSalesValue FROM CustomerSales;</code>
AVGOFSALESVALUE 942.114155	

Question	List customers who purchased more than the average customer.	
SQL	<pre>SELECT LastName, FirstName, SalesValue FROM CustomerSales WHERE SalesValue > (SELECT Avg(SalesValue) FROM CustomerSales) ORDER BY SalesValue DESC;</pre>	
LASTNAME	FIRSTNAME	SALESVALUE
Lyons	Chester	3569
Hines	Arlene	2815
Dixon	Carol	2789
Gillespie	Audrey	2703
O'Connor	Carlos	2674
Ford	Manuel	2661
Nash	Joseph	2600
Rush	Bonita	2485
Warden	Jewell	2406
Turner	Guy	2358
Harvey	Simon	2314
Peck	Burt	2260
Crowe	Chelsea	2254
McCartney	JoAnne	2237
Crowe	Vicky	2165
More rows...		

Figure 5.4



Activity: Build Outer Joins

Joining tables is one of the more complex issues in SQL. Up to this point, the joins have been simple equality joins designed to show how a column in one table links to data stored in a related table. It is important that you understand the effect of this join. Jim, the sales manager, and David, the rental manager, want to know if customers who rent equipment also purchase items for sale. As with many questions, there are several different ways to build this query. Figure 5.5 shows the effect of an inner join. Build a new query and add the Rental and Sale tables. Join these tables by matching the CustomerID value from each table. When you display both CustomerID values in the query and run it, you can see that they are the same. The effect of this join is that the results show the customers (ID only) who participated in a sale and a rental—at any time.

If you want to know which customers made a purchase on the same day as the rental, you could add a condition that RentDate equals SaleDate. Or you could add a second join that connects RentDate and SaleDate. Figure 5.6 shows the query with the second join condition. Notice the use of the AND in the join statement. This query demonstrates the effect of the inner join.

Action

Create a new query.

Table: CustomerSales query.

Columns: LastName, FirstName, SalesValue.

Criteria for SalesValue

>(SELECT Avg(SalesValue) FROM CustomerSales).

Action

Create a new query.

Tables: Rental and Sale.

Columns: RentDate, SaleDate, and CustomerID from both tables.

Join the tables on CustomerID.

Run the query.

Add a join between the tables on RentDate=SaleDate.

Run the query.

Question	List rental customers who also purchased items, at any time.
SQL	<pre>SELECT RentDate, Rental.CustomerID, Sale.CustomerID, SaleDate FROM Rental INNER JOIN Sale ON Rental.CustomerID = Sale.CustomerID;</pre>
<pre>RENTDATE CUSTOMERID CUSTOMERID SALEDATE 28-MAR-10 1535 1535 25-JAN-10 02-DEC-10 1455 1455 17-SEP-10 12-NOV-10 1642 1642 25-DEC-10 04-NOV-10 1186 1186 30-APR-10 19-NOV-10 51 51 28-OCT-10 01-NOV-10 1602 1602 25-AUG-10 11-MAR-10 1452 1452 30-AUG-10 06-NOV-10 1455 1455 17-SEP-10 27-NOV-10 1645 1645 12-MAR-10 03-JAN-10 1992 1992 28-JUN-10 11-DEC-10 1861 1861 19-JUN-10 More rows...</pre>	

Figure 5.5

In many respects, it is equivalent to a WHERE clause. The inner join restricts the rows that you will see by forcing values to be equal.

On the other hand, perhaps Jim would like to see a list of all of the customers who participated in sales, and then check to see which of those have rented items. You need to build a new query. This time include the Customer table so their names can be displayed. Then add the Sale and Rental tables.

Do not include a join from Customer to Rental. That join would force all of the CustomerIDs to be equal, which is not what Jim wants. Instead, connect Rental to Sale by CustomerID, but with a different type of join. Figure 5.7 shows the basic query. As shown in the SQL, this query uses a LEFT JOIN, which displays all values in the Sale table (the left table in the SQL query list), even if the customer never rented items. You often need to include parentheses in a JOIN statement to

Action

Create a new query.

```
FROM (Customer INNER JOIN Sale
ON Customer.CustomerID=Sale.
CustomerID)
```

```
LEFT JOIN Rental ON Sale.CustomerID
= Rental.CustomerID.
```

Columns: LastName, FirstName, and CustomerID from Sale and Rental.

Run the query.

Figure 5.6

Question	List rental customers who also purchased items on the same day.
SQL	<pre>SELECT RentDate, Rental.CustomerID, Sale.CustomerID, SaleDate FROM Rental INNER JOIN Sale ON (Rental.CustomerID = Sale.CustomerID AND Rental.RentDate = Sale.SaleDate);</pre>
<pre>RENTDATE CUSTOMERID CUSTOMERID SALEDATE 09-NOV-10 930 930 09-NOV-10 07-JAN-10 1291 1291 07-JAN-10 10-NOV-10 1629 1629 10-NOV-10</pre>	

Question	List all customers who rented and did or did not make a purchase.		
SQL	<pre>SELECT LastName, FirstName, Sale.CustomerID, Rental.CustomerID FROM (Customer INNER JOIN Sale ON Customer.CustomerID=Sale. CustomerID) LEFT JOIN Rental ON Sale.CustomerID=Rental.CustomerID ORDER BY LastName, FirstName;</pre>		
LASTNAME	FIRSTNAME	CUSTOMERID	CUSTOMERID
Abel	Marshall	1406	1406
Abel	Marshall	1406	1406
Abel	Melinda	1467	1467
Abrams	Marc	603	(null)
Adkins	April	413	413
Adkins	Manuel	1499	1499
Aldrich	Jewell	142	142
Aldrich	Jewell	142	142
Aldrich	Jewell	142	142
Aldrich	Jewell	142	142
Alexander	Marvin	645	645
Allen	Laura	1085	(null)
Allen	Orson	1928	1928
Allen	Orson	1928	1928
Baez	Agnes	302	(null)
Bailey	Takao	879	879
Baker	Hazel	1664	1664

Figure 5.7

force the left join to be performed first. Even so, if you have problems running a LEFT JOIN query, you might have to remove tables until you have only two tables or views in the query. Sometimes you have to build the left join with only two tables, save the query, then create a second query based on the saved query and any other tables needed.

Figure 5.7 also shows some of the results from running the query. Notice that several of the rows show missing values for the Rental.CustomerID. These are the customers who purchased items but have never participated in a rental. If you want to see only this list of people, you can add the condition that Rental.CustomerID Is Null. Observe that the full list from the main query might not include all of the customers. To review your knowledge of joins, you should be able to identify the customers that might not be in this list. Looking at the design, notice that there is still an inner join between the Customer and Sale tables. Consequently, customers who have not participated in sales at all will not be displayed in this list. If you truly wanted a list of all customers, you would have to use a left join from the Customer to the Sale table. However, you will probably have to do one of the joins at a time, save the query, and then do the second join.

Recall the question of listing the customers who have purchased items but have not rented anything. With the left join, it is straightforward to get this list by adding the Is Null condition. But you must be very careful when creating this query. If you forget to specify the left join and stick with the standard inner join, the query will indicate that no customers match that condition. The reason is because an inner join automatically leaves out the customers you are searching for. This question can also be answered with a subquery. Figure 5.8 shows the subquery approach. Start a new query and add the Customer and Sale tables. Sort the columns by LastName and FirstName. Then add the condition CustomerID Not In (SELECT CustomerID FROM Rental). As always, remember to put the subquery

Question	List customers who bought items but never rented.	
SQL	<pre>SELECT LastName, FirstName, Customer.CustomerID FROM Customer INNER JOIN Sale ON Customer.CustomerID = Sale. CustomerID WHERE Customer.CustomerID NOT IN (SELECT CustomerID FROM Rental) ORDER BY LastName, FirstName;</pre>	
LASTNAME	FIRSTNAME	CUSTOMERID
Abrams	Marc	603
Allen	Laura	1085
Baez	Agnes	302
Baldwin	Orville	403
Bell	Leslie	19
Brown	Tony	832
Brown	Tony	832
Buchanon	Orson	66
Cardwell	Christina	1377
Cardwell	Christina	1377

Figure 5.8

in parentheses. This query will retrieve all Customers who have participated in sales but have not rented any items. You should compare the results from this version to the left join version to ensure that both queries return the same results.

Most systems support either method to answer the question, but there can be performance differences between the two approaches. Oracle optimizes all queries, so you can write the query as either a subquery or a left join and Oracle will rewrite it to achieve the best performance. In fact, Oracle examines statistics about the various tables to help make the best choices. Depending on the data within a table, it might be faster to apply a where condition first, Oracle examines the distribution of data to build the best query method. Particularly on large tables, you should run scripts to tell Oracle to generate the statistics, such as:

```
ANALYZE TABLE Customer COMPUTE STATISTICS;
```

Activity: Create Complex Joins

Jim, the sales manager, is concerned about excess inventory. He wants to be able to monitor the status of quantity on hand for all inventory items. He is particularly concerned about identifying which models are selling quickly versus models that have large numbers of items sitting around. Remember that models are product lines from the manufacturers, while individual items are specific sizes within a model group. He wants the totals for the model. To see if

Action

Create a new query.

Tables: Customer and Sale.

Columns: LastName, FirstName, and CustomerID.

WHERE CustomerID Not In (SELECT CustomerID FROM Rental).

Run the query.

Action

Create a new query.

Table: Inventory.

Columns: ModelID and Sum(QuantityOnHand).

Sort by the Sum descending.

Run the query.

Save it as ModelsOnHand.

Create a new table: SalesCategory.

Columns: CategoryID, CategoryName, LowLimit, HighLimit.

Enter data from Figure 5.10.



Question	Compute quantity on hand by ModelID.
SQL	<pre>SELECT ModelID, Sum(QuantityOnHand) As SumOfQuantityOnHand FROM Inventory GROUP BY ModelID ORDER BY Sum(QuantityOnHand) DESC;</pre>
MODELID	SUM(QUANTITYONHAND)
YXY-385	70
YCG-584	70
QDV-720	70
MHQ-568	60
LDK-181	60
YKU-321	60
NTE-526	50
More ...	

Figure 5.9

there is a problem, construct a new query that totals the quantity on hand and sorts it in descending order by ModelID. Figure 5.9 shows the total quantity on hand for a few of the models. Save the query as ModelsOnHand.

But Jim does not want to wade through the entire query every day. Instead, he is proposing a categorical system, where items with more than a certain QOH will be called slow sellers, and items with minimal QOH will be hot sellers. He also wants a few categories in between. While you have the tools to build this query, there is one catch: he wants the ability to fine-tune the numbers on the ranges for each category. The solution is to create a new table that defines the category and the upper and lower limits for each category: SalesCategory(CategoryID, CategoryName, LowLimit, HighLimit). If the QOH for a model is greater than or equal to the LowLimit and less than the HighLimit, it falls into the specified category. The CategoryID ensures a unique key and could be used to sort the rows if necessary. You need to create the table and then load some initial

Action

Create a new query.

Columns: ModelID,
SumOfQuantityOnHand, CategoryID,
and CategoryName.

Tables: ModelsOnHand and
SalesCategory.

Add the inequality join.

Run the query.

Figure 5.10

Question	Data for the new SalesCategory table.		
SQL	<pre>INSERT INTO SalesCategory VALUES (1, 'Hot', 0, 6); INSERT INTO SalesCategory VALUES (2, 'Good', 6, 10); INSERT INTO SalesCategory VALUES (3, 'OK', 10, 20); INSERT INTO SalesCategory VALUES (4, 'Weak', 20, 40); INSERT INTO SalesCategory VALUES (5, 'Slow', 40, 1000); SELECT * FROM SalesCategory;</pre>		
CATEGORYID	CATEGORYNAME	LOWLIMIT	HIGHLIMIT
1	Hot	0	6
2	Good	6	10
3	OK	10	20
4	Weak	20	40
5	Slow	40	1000

Question	Models categorized based on quantity on hand.
SQL	<pre>SELECT ModelID, SumOfQuantityOnHand, SalesCategory. CategoryID, CategoryName FROM ModelsOnHand INNER JOIN SalesCategory ON (ModelsOnHand.SumOfQuantityOnHand >= SalesCategory.LowLimit) AND (ModelsOnHand.SumOfQuantityOnHand < SalesCategory.HighLimit);</pre>
MODELID	SUMOFQUANTITYONHAND CATEGORYID CATEGORYNAME
ENW-975	2 1 Hot
EZX-852	2 1 Hot
IQE-600	2 1 Hot
KSB-825	2 1 Hot
XUW-452	2 1 Hot
PKT-115	2 1 Hot
RFL-870	2 1 Hot
SXT-833	2 1 Hot
WER-904	2 1 Hot
LNH-128	2 1 Hot
<i>(many rows not shown)</i>	

Figure 5.11

data. Figure 5.10 shows the initial categories. The CREATE TABLE command should be easy now:

```
CREATE TABLE SalesCategory
(
  CategoryID    INTEGER,
  CategoryName  NVARCHAR2(15),
  LowLimit      INTEGER,
  HighLimit     INTEGER,
  CONSTRAINT pk_SalesCategory PRIMARY KEY(CategoryID)
);
```

Using the categories in a query requires slightly tricky join conditions. You need to use inequality (theta) joins. Begin with a new query based on the ModelsOnHand query and the SalesCategory table. Select the ModelID and SumOfQuantityOnHand along with the CategoryName. But, the JOIN command is considerably different from the others you have done. Figure 5.11 shows the inequality

Figure 5.12

Question	Count the number of models in each sales category.
SQL	<pre>SELECT SalesCategory.CategoryID, CategoryName, Count(ModelID) FROM ModelsOnHand INNER JOIN SalesCategory ON (ModelsOnHand.SumOfQuantityOnHand >= SalesCategory.LowLimit) AND (ModelsOnHand.SumOfQuantityOnHand < SalesCategory.HighLimit) GROUP BY SalesCategory.CategoryID, CategoryName ;</pre>
CATEGORYID	CATEGORYNAME COUNT (MODELID)
2	Good 253
1	Hot 179
5	Slow 9
3	OK 29
4	Weak 35

statements used to join the two views. Figure 5.11 also shows the sample result from the query. Save the query as ModelSales so Jim can perform some additional analysis on the data.

Jim might create a new, simpler query that counts the number of models that fall within each of the categories. Figure 5.12 shows the basic query. It is built using the results of the previous query. This query hides the complicated details and Jim needs to see only the simple data results. The final aggregation query uses the CategoryID to sort the results logically; otherwise, they would be sorted alphabetically by the category name. Fortunately, most of the models appear to be in the categories indicating that they sell relatively quickly. However, the category definitions might not be accurate, but Jim can quickly alter the range numbers and rerun the query to see the results.

Action

Create a new query.

Columns: CustomerID, LastName, FirstName, and SaleDate.

Tables: Customer and Sale.

Set January sale date in WHERE.

Copy the entire statement.

Add the word Union.

Paste the SELECT statement and change the date condition and name to March.

Run the query.



Activity: Combine Data Rows with UNION

You need to understand the role of the UNION command. It is designed to combine rows from multiple queries. Read that sentence carefully. It says combine rows not columns. If you have two queries that retrieve similar columns of data, the UNION statement will combine the results into one set of data. To illustrate the process, consider a request that Katy made to see a single list of customers who purchased items in January or in March. You could build two queries using

Figure 5.13

Question	List customers who bought items in January or March.		
SQL	<pre>SELECT Customer.CustomerID, LastName, FirstName, 'Jan' As SaleMonth FROM Customer INNER JOIN Sale ON Customer.CustomerID = Sale.CustomerID WHERE SaleDate BETWEEN '01-Jan-2006' AND '31-Jan-2006' UNION SELECT Customer.CustomerID, LastName, FirstName, 'Mar' As SaleMonth FROM Customer INNER JOIN Sale ON Customer.CustomerID = Sale.CustomerID WHERE SaleDate BETWEEN '01-Mar-2006' AND '31-Mar-2006' ;</pre>		
CUSTOMERID	LASTNAME	FIRSTNAME	SAL
19	Bell	Leslie	Mar
92	Shelton	Kelly	Mar
119	Garrison	Hershel	Mar
120	McDougal	Andrew	Mar
155	O' Connor	Carlos	Mar
302	Baez	Agnes	Mar
315	Vance	Dominic	Mar
345	Dennison	Lena	Mar
411	Keen	Elmer	Mar
431	Hinton	Adam	Mar
<i>(many rows not shown)</i>			

EmployeeID	LastName	FirstName	ManagerID
0	Staff		8
1	Killy	Jean-Claude	8
2	Miyahira	Hideharu	4
3	Street	Picabo	6
4	Heiden	Beth	1
5	Cavagnoud	Regine	4
6	Daehlie	Bjorn	1
7	Moe	Tommy	15
8	Tomba	Alberto	18
9	Jasey-Jay	Anderson	15
10	Carr	Chris	4
11	Fawcett	Mark	6
12	Mckenna	Lesley	18
13	Weinbrecht	Donna	3
14	Adam	Arno	3
15	Brassard	Jean-Luc	1
16	Boit	Philip	6
17	Bourgeat	Pierrick	15
18	Gravier	Richard	1

Figure 5.14

simple WHERE conditions, but if you want to list people twice if they bought items both in January and in March, the UNION query is easier.

As shown in Figure 5.13, create a new query using the Customer and Sale tables. Display the CustomerID, LastName, and FirstName columns. Add the SaleDate column, but uncheck the box to display the date. Add the condition to select sales only in January. If you run the query at this point, you will see a list of customers who bought items in January. To get the March customers, copy the entire statement without the semicolon. Add the word “UNION” after the existing query, then below that, paste a copy of the query. Now modify the dates in this copy to indicate March instead of January. Finally, in the first (January) SELECT statement, add a computed column to display “Jan” As SaleMonth. Do the same thing for the second SELECT statement, but display “Mar” for March. This column will identify each row to indicate the month for the sale. Run the query, and you will see a combination of rows from both queries. If you want to sort the data by Customer or by date, first you will have to save the query, then you can build a second query based on the first and sort the columns as needed.



Activity: Create Recursive Hierarchical Queries

The recursive query is somewhat strange, but it can be useful for particular types of queries. Unfortunately, Oracle relies on its own syntax for recursive or hierarchical queries, but that syntax is straightforward.

Action

- Look at the Employee table—the ManagerID column.
- Create the recursive query to show the reporting relationships.
- Check the results against the data.

```

*Killy
**Heiden
***Miyahira
***Cavagnoud
***Carr
**Daehlie
***Street
****Weinbrecht
****Adam
***Boit
**Brassard
***Moe
***Jasey-Jay
***Bourgeat
**Gravier
***Tomba
****Staff
***Mckenna
****Fawcett

```

Figure 5.15

Figure 5.14 shows the important columns in the Employee table. Notice the ManagerID column shows the ID number for the manager of each employee. First, note that Killy (EmployeeID = 1) does not have a manager, so he is at the highest level. Now check the entry for EmployeeID = 4, Beth Heiden. Her manager is EmployeeID = 1, or Killy. If you look through the list, you can see that Beth is manager to a couple of other employees. The goal is to write a query that can track through the list, starting with Killy (ManagerID Is Null), and listing each person who reports to him, and then each person who reports to that person, and so on.

This question would be difficult or impossible to answer with traditional SQL, so vendors have created special syntax to handle the problem. The Oracle version introduces two key terms: “Start With” and “Connect by Prior.” With an extra twist, the query is:

```

SELECT lpad( '*' , level, '*' ) || LastName
FROM EMPLOYEE
Start with ManagerID Is null
Connect by prior EmployeeID=ManagerID;

```

Figure 5.15 shows the results. The asterisks were generated with the lpad function which adds the number of asterisks based on the “level,” which is a pseudo-column that is part of the recursive query. The level is simply an integer specifying the depth of the recursive tree. It makes it easier to see that all of the employees with two stars (Heiden, Daehlie, Brassard, and Gravier) report to the level-1 employee (Killy).

The structure of the query is straightforward. Begin with a SELECT statement that identifies the columns to be displayed. The “level” column is helpful either by itself or using the lpad function shown here. Use the FROM statement to specify the table—the recursive query is somewhat picky about using a single table, but search the Web and you can find workarounds to handle multiple tables. The “Start With” phrase specifies the top level—typically you know the specific value or the top level has some special value such as null or a negative number. Use the “Connect by Prior” phrase to specify the condition that joins the ManagerID to the EmployeeID or lookup column.

```
CREATE TABLE Contacts
(
  ContactID          INTEGER,
  ManufacturerID     INTEGER,
  LastName           NVARCHAR2(25),
  FirstName          NVARCHAR2(25),
  Phone             NVARCHAR2 (15),
  Email             NVARCHAR2 (120),
  CONSTRAINT pk_Contacts PRIMARY KEY (ContactID),
  CONSTRAINT fk_ContactsManufacturer FOREIGN KEY (ManufacturerID)
    REFERENCES Manufacturer(ManufacturerID)
    ON DELETE CASCADE
);
```

Figure 5.16

SQL Data Definition and Data Manipulation



Activity: Create Tables

Although it is possible to create and delete tables in Oracle using the enterprise manager, you will often have to create a table using the data definition language (DDL) `CREATE TABLE` command. For example, after working with the database for a while, you realize that it would be useful to have a separate table that lists salespeople and other contacts at the manufacturers. Each person has a direct phone number and an e-mail address. To practice building tables, Figure 5.16 shows the `CREATE TABLE` command for the new `Contacts` table. Essentially, you list each desired column along with its data type. Since Oracle supports the ANSI standard data types, it is often more convenient to specify `INTEGER` for numeric data types instead of `NUMBER`; but either version will work.

Enter the SQL code in either SQL Developer and run the query. You should receive the “Table created” message. If not, check your typing carefully. You should create the primary key constraint to indicate the `ContactID` is the sole primary key column. For other tables, if you need multiple columns, simply create a comma-separated list. The foreign key constraint is similar, but you must also specify the table and column that is referenced by the foreign key. Be sure to specify the `ON DELETE CASCADE` option for the foreign key. If rows are deleted in the master

Action

Create a new query.

Enter the `CREATE TABLE` command.

Run the query.

Figure 5.17

```
CREATE TABLE MyTemp
(
  ID      INTEGER,
  LName  NVARCHAR2(25),
  FName  NVARCHAR2(25),
  CONSTRAINT pk_MyTemp PRIMARY KEY (ID)
);
```



```
INSERT INTO Customer (CustomerID, LastName, FirstName, City, Gender)
VALUES (4000,'Jones', 'Jack', 'Nowhere', 'Male');
```

Figure 5.18

table (Manufacturer), then any contacts in this table associated with that manufacturer will also be deleted automatically.

Generally, with Oracle it is easier to create tables with SQL. It is particularly useful to create a text file that contains several CREATE statements that will generate the database automatically. First, you want to test each statement individually and make sure it contains the correct statement. Then cut and paste the command into a separate text file. This file can be given to others to create the database on a different system. The CREATE TABLE command is also useful for creating temporary tables. Figure 5.17 shows the table that you need to create.

The SQL ALTER TABLE command can also be used to add new columns to an existing table. However, you rarely need this command if you work from a good design. You can also use the Enterprise Manager console to add columns to a table—and it will show you the full syntax of the SQL command. For example, to add a TempCost column to the ItemModel table, the command would be ALTER TABLE ItemModel ADD (TempCost NUMBER(38,4)).



Activity: Insert, Update, and Delete Data

SQL also provides data manipulation language (DML) commands to insert, update, and delete rows of data. Consider the INSERT command first. The simple version of the command shown in Figure 5.18 inserts a single row into one table. Notice that you specify the table columns in the first list and the corresponding values in the second list. By listing the column names, you choose to enter the data in any order and to skip columns. Of course, you will rarely enter data this way, but occasionally it comes in handy. More importantly, the SQL statement can be generated using programming code with complex routines to extract data from one source, clean it up and transfer it to the desired table. Notice that you must include the CustomerID column at this point. Chapter 7 will explain how to create a sequence number so this value can be generated automatically.

Action
 Create a new query.
 Type the INSERT command: INSERT INTO Customer (CustomerID, LastName, FirstName, City, Gender) VALUES (4000, 'Jones', 'Jack', 'Nowhere', 'Male');
 Run the query.

A second version of the INSERT command is more useful because of its power. You use it to transfer large blocks of data from one table into a second table. Note that the second table must already exist. The example in Figure 5.19 copies some data from the Customer table and transfers it to the temporary MyTemp table you created in the previous section. Again, you list the columns for the new table that

Figure 5.19

```
INSERT INTO MyTemp (ID, LName, FName)
SELECT CustomerID, LastName, FirstName
FROM Customer
WHERE City='Sacramento';
```

Question	Increase the cost of snow boards by four percent for 2010.
SQL	<pre> SELECT Category, ModelYear, Round(Cost*1.04,2) FROM ItemModel WHERE Category='Board' AND ModelYear=2006 ; -- run the top query first, then edit it to make the actual changes UPDATE ItemModel SET Cost = Round(Cost*1.04,2) WHERE Category='Board' AND ModelYear=2006; </pre>
88 rows updated.	

Figure 5.20

will hold the data, then write a SELECT statement that retrieves matching data for those columns. Be sure to issue a COMMIT command after any INSERT command to ensure changes are saved to the table.

You should keep in mind that the SELECT statement can be as complex as you wish. It can include calculations, multiple tables, complex WHERE conditions, and subqueries. For complex queries, you should first build the SELECT statement on its own and test it to ensure that it retrieves exactly the data you want. Then switch to the SQL view and add the INSERT INTO line at the top. The ability to perform calculations has another benefit. You can add a constant to the SELECT statement that will be inserted as data into the second table. For example, you might write SELECT ID, Name, “West” to insert a region name into a new table. The INSERT INTO command is useful when you need to expand a database or add new tables. You can quickly copy selected rows and columns of data into a new table.

The UPDATE command is used to change individual values for specified rows. It is a powerful command that affects many rows. You must always be cautious when using this command because it can quickly change thousands of rows of data. To illustrate the power of the command, consider that the manufacturers have announced that costs will increase by 4 percent for the 2010 snowboards. The ItemModel table contains an estimate of the Cost for each model, so you need to increase this number by 4 percent, but only for the boards.

To be safe, begin by creating a query that displays the Cost data for the 2004 boards. You should run the query to ensure that it returns exactly the data that you want to update. Next, as shown in Figure 5.20, edit the query so that it uses the UPDATE command instead of SELECT. The Round function is used to ensure that the final Cost value is rounded off to cents instead of extended fractions. Be sure you run the SELECT query first to ensure the correct rows are selected by the WHERE clause. Then edit the query by adding the UPDATE statement. In practice, do not try to run both queries at the same time. They are shown here only so you can compare the two. After you run an UPDATE query, you should issue a COMMIT command to make sure the changes are recorded to the table.

Action

Create a new query.

Columns: Category, ModelYear, and Round(Cost*1.04,2).

Table: ItemModel.

Criteria: Category='Board' And ModelYear=2010.

Run the query.

Change the first two lines to:

```
UPDATE ItemModel
```

```
SET Cost = Round(Cost*1.04,2).
```

Run the query.

Question	Delete sample row from the MyTemp table.
SQL	<pre>-- SELECT * DELETE FROM MyTemp WHERE ID > 100; Commit;</pre>
	<pre> ID LNAME FNAME ----- 1184 Cherry Louis 1 row deleted. Commit; Committed</pre>

Figure 5.21

Notice that the SQL statement is straightforward. It is also easy to change multiple columns at one time. Just separate the column assignments with commas. For example: SET Cost = Round(Cost * 1.04,2), ModelYear = 2010.

The DELETE command is similar to the INSERT and UPDATE commands, but it is more dangerous. It is designed to delete many rows of data at a time. Keep in mind that because of the relationships, when you delete a row from one table, it can trigger cascade deletes on additional tables. For the most part, these deletes are permanent. If you are not careful, you could wipe out a large chunk of your data with one DELETE command. To minimize the impact of these problems, you should always make backup copies of your database—particularly before you attempt major delete operations. If your system has an Oracle Management server installed, you can use the backup wizard in the Enterprise Manager console to make a backup copy of the schema.

To be particularly safe, this example is just going to delete data from the temporary table that was created in the previous section. Create a new query using the MyTemp table. As shown in Figure 5.21, to see the rows you are going to delete, display the ID, LName, and FName columns and set a condition to show only rows with an ID > 100. Run the query to verify that it returns only one row. Now, edit the query and replace the entire SELECT row with the DELETE command. Run the query. If only one row is deleted, issue the COMMIT command to make the deletions permanent. Your other option is to issue a ROLLBACK command to restore everything to the last point where you executed a commit.

In practice, it is best to stick with simple WHERE clauses when possible. However, it can be complex and can include subqueries. Particularly in the complex cases, you should first build a SELECT statement using the same WHERE clause to ensure that you are deleting exactly the rows you want to delete. Then convert the query into a Delete Query, or delete the SELECT statement and replace it with the DELETE command.

Action

```
Create a new query: SELECT * FROM
MyTemp WHERE ID > 100;
Test the query.
Change the SELECT row to DELETE.
Run the query.
Run a commit; command.
```

Figure 5.22

```
DROP TABLE MyTemp;
```

The DROP TABLE command is even more dangerous. It removes the entire table and all of its data. Generally, you should only use it for temporary tables. As shown in Figure 5.22, the syntax is straightforward, just make sure you enter the correct table name. Again, it would be wise to make a backup copy of your database before removing tables.

The main aspect to remember about these commands is that they operate on sets of rows that you control with the WHERE clause. The WHERE clause can be complex and include subqueries with detailed SELECT commands. All of the power of the SELECT command is available to you to control inserting, updating, and deleting rows of data.



Activity: Create Parameter Queries

Parameter queries are useful when you need to create a complex query that a manager runs on a regular basis but needs to change some of the constraints. For instance, you often use parameters to set starting and ending dates so the manager can easily select a range of data without having to know anything about building queries. The

example in Figure 5.23 shows a query that displays the total rental income by Category for a specified range of dates. This query has fixed dates for the first quarter. The objective is to replace those fixed dates with parameters that can be entered quickly by the manager—preferably without having to see or edit the query.

In Oracle, parameterized SELECT queries are somewhat complicated. They require the use of variables, which means you need to create a small package to hold the variable definitions and the procedure that executes the query. These topics are explored in more detail in Chapter 7, but it is worth typing in this example to see how they work.

To create the package and procedure, enter the commands in Figure 5.24 by typing them in. The commands are also stored as a text file on the student CD so you can cut and paste them to save some typing. The package defines a record and a cursor that are used to return the selected values. It also contains the definition of

Action

Create a new query.

Columns: Category, Sum(RentFee).

Tables: Rental, RentItem, Inventory, and ItemModel.

GROUP BY Category.

Test the query.

Figure 5.23

Question	Show rental totals by category for a specified time period.
SQL	<pre>SELECT Category, Sum(RentFee) AS SumOfRentFee FROM Rental INNER JOIN RentItem ON Rental.RentID=RentItem.RentID INNER JOIN Inventory ON RentItem.SKU=Inventory.SKU INNER JOIN ItemModel ON Inventory.ModelID=ItemModel.ModelID WHERE RentDate Between '01-Jan-2006' And '31-Mar-2006' GROUP BY Category;</pre>
CATEGORY	SUMOFRENTFEE
-----	-----
Board	18660
Boots	14370
Electronic	1350
Poles	790
Rack	420
Ski	37900

the procedure. The package body contains the actual procedure. Notice that it is passed a starting and ending date and returns the matching rows of data. The heart of the procedure is the query that you already created. The only difference is that the two dates are specified as parameters. The package and procedure only have to be created one time.

Once the package is defined, the manager only needs to issue a couple of simple commands to enter new dates and obtain the total rental fees by category. Figure 5.25 shows the commands. Some of them are the common formatting commands. You would probably want to save these commands in a file so the manager can cut and paste them, and then edit the two dates to simplify the process.

You can build complex queries and insert parameters to request specific data from the person running the query. Although it requires several steps, query parameters are a useful method to quickly build queries that users can control without having to alter the query.

Action

Create a new query.

Copy or paste the code to create the package and procedure.

Run the package creation code.

Enter the five commands to execute the parameter query.

Run the query.

Change the dates to Oct-Dec.

Run the query.

Figure 5.24

```
CREATE PACKAGE pckCategoryFees AS
  TYPE typeCategoryFees IS RECORD
    ( Category NVARCHAR2(15),
      SumOfRentFees NUMBER(8,2)
    );
  TYPE typeCursorFees IS REF CURSOR RETURN typeCategoryFees;
  PROCEDURE GetCategoryFees
    ( dateStart IN DATE,
      dateEnd   IN DATE,
      cvFees    IN OUT typeCursorFees
    );
END;
/
CREATE PACKAGE BODY pckCategoryFees AS
  PROCEDURE GetCategoryFees
    ( dateStart IN DATE,
      dateEnd   IN DATE,
      cvFees    IN OUT typeCursorFees
    ) IS
    BEGIN
      OPEN cvFees FOR
        SELECT Category, Sum(RentFee) AS SumOfRentFee
        FROM Rental INNER JOIN RentItem INNER JOIN Inventory
              INNER JOIN ItemModel
        ON Inventory.ModelID=ItemModel.ModelID
        ON RentItem.SKU=Inventory.SKU
        ON Rental.RentID=RentItem.RentID
        WHERE RentDate Between dateStart And dateEnd
        GROUP BY Category;
    END;
END;
/
```

Question	Use the parameterized procedure to display rental totals for specified dates.
SQL	SET AUTOPRINT ON VARIABLE cv REFCURSOR EXECUTE pckCategoryFees.GetCategoryFees('01-Jan-2006', '31-Mar-2006', :cv);
CATEGORY	SUMOFRENTFEE
-----	-----
Board	\$18,660.00
Boots	\$14,370.00
Electronic	\$1,350.00
Poles	\$790.00
Rack	\$420.00
Ski	\$37,900.00

Figure 5.25

Note, if you just need to reuse a single query in one session, you can use substitution variables instead of building a packaged procedure. Simply write the SQL statement as usual, but place a variable using an ampersand, instead of a literal. For example: WHERE RentDate BETWEEN &DateStart and &DateEnd. When the query runs, the user will be prompted to enter the DateStart and DateEnd values which will be substituted into the query.

Exercises



Many Charms

You will need to create some additional sample data for each table. Madison and Samantha know that they will want certain information on a weekly basis, but they will not be able to build complex queries to retrieve the data. You will have to build a few queries for them that they can run when they want to see the results or need to change prices. Some of the queries should be parameter queries so they can easily select the values they need to control the results. Note: You will have to modify the queries slightly to match the data that you have entered.

1. Which of the customers who ordered bracelets have not ordered necklaces?
2. Which customers bought more gold charms than silver ones?
3. Which categories generated the most profit over a parameterized time period?
4. Are expensive charms more profitable than mid-priced or low-priced charms? Hint: Create categories based on the prices.
5. Create a parameterized query to enable Samantha to increase prices of a certain category of charms by a given percentage.
6. Create a new table with SQL and copy into it all of the customers who have not purchased items within the last three months.
7. Delete customers from the new table in the prior exercise who have spent more than \$100 in the past year.



Standup Foods

You will need to create some additional sample data for each table. Laura knows she will want certain information on a weekly basis, but she will not be able to build complex queries to retrieve the data. You will have to build a few queries for

her that can be run to display results or change prices. Some of the queries should be parameter queries so Laura can easily select the values she needs to control the results. Note: you will have to modify the queries slightly to match the data that you have entered.

1. Identify the employees who have below average overall job evaluations.
2. Identify the main menu items that have not been served to a particular director or other celebrity (pick one from your list who wants something different).
3. Which customers have not yet referred her business to other clients?
4. Create a category table to segment the employee ratings (excellent, good, average, weak). Use the table to identify the employees with excellent evaluations as both server and dishwasher.
5. Create a temporary table and copy into it information about employees who have worked as drivers but have not driven within the last month.
6. Delete from the temporary table in the previous question the drivers whose average evaluations are less than 6 (on the 10-point scale).
7. Write a parameterized query that enables Laura to increase the base wage rate of employees by specifying a category, a minimum overall average evaluation, and the percentage increase.



EnviroSpeed

You will need to create some additional sample data for each table. Brennan and Tyler know that they will want certain information on a weekly basis, but they will not be able to build complex queries to retrieve the data. You will have to build a few queries for them that they can run when they want to see the results or need to change prices. Some of the queries should be parameter queries so they can easily select the values they need to control the results. Note: You will have to modify the queries slightly to match the data that you have entered.

1. List the experts who have worked with two or more crews in the same month.
2. Which experts have not contributed any documents within the last three months?
3. List the crews that are more than 25 percent larger than the average crew.
4. Create a table to categorize the expensiveness of cleanups. For example, spills that cost more than \$1 million to clean up are expensive; spills that cost \$500,000 to \$1 million are merely costly; and so on. Create a query to apply these categories to the actual spills.
5. Write a query that retrieves documents based on a list of keywords entered by a user. The keywords might appear anywhere in the document, and the final query should sort the list based on the number of matches.
6. Write a parameterized query to update a severity value for an incident by allowing the user to enter a chemical name and a point-wise increase in severity.

7. Write a query to copy the data on experts to a new table who have participated in a total of at least three incidents in the last year.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks. You will have to create sample data for each of the tables.

1. Identify and create at least two parameter queries that would be useful to managers. Share the business question (not the query) with other students and solve their queries.
2. Identify a business question to list items greater (or less) than average. Write the query to return the results.
3. Create a temporary table and write a query to copy some rows of data from one table into the new table.
4. Write a delete query to remove a few rows of data from the temporary table.
5. Write an update query using parameters to change the value of one of the numeric columns in a table based on a percentage and conditions entered by the user.

Forms and Reports

Chapter Outline

Forms and Reports, 100

Model-View-Controller, 101

Case: All Powder Board and Ski Shop, 102

Lab Exercise, 103

All Powder Board and Ski Shop Forms, 103

All Powder Basic Reports, 131

Exercises, 136

Final Project, 137

Objectives

- Create forms that make it easy for users to enter data.
- Create three types of forms (main, grid, subform) to understand the purpose of each.
- Create reports to display and summarize data.

Forms and Reports

The main purpose of the DBMS is to store data efficiently and provide queries to retrieve data to answer business questions. But from the perspective of businesses, the true value of the DBMS lies in the applications that can be built on top of the database. Oracle provides tools to help you build forms and reports. The tools help you create the basic forms and reports quickly; however, you will still need to edit the designs to clean them up to make them easier to read. One of first things you will see with Oracle 11g has many different tools to handle the same tasks. Oracle developed and purchased several technologies and 11g “Fusion” is designed to support multiple technologies. This flexibility can be useful, but it is also overwhelming. A given project should try to stick with a relatively small set of technologies. This chapter focuses on a few of the newer Web-based technologies (such as ADF Faces) so students can learn one fundamental technology first. Other options can be added to projects later if specific tools are needed.

Forms are used to make it easier for users to enter data. You would never want users to enter data directly into the tables. For example, look again at the Sale table. It contains mostly ID numbers, and you cannot expect workers to memorize thousands of ID numbers. Instead, you build forms to match the processes and styles of the business. Likewise, you rarely ask managers to build queries themselves. Instead, you create reports that display details and subtotals within a layout that is easy to read. You can even include charts to make it easy to compare values or examine trends over time.

With Oracle 11g, all forms are designed to run as Web pages and the forms run in relatively standard browsers using Javascript and interactive AJAX-based components. Web-based forms and reports have several benefits over older methods. The most important is that users can reach the data from anywhere in the world with a variety of devices including low-cost laptops and even cell phones. Also, all of the forms and data are centrally located making it easier to upgrade the application and monitor usage and security controls. Web forms still have some limitations in terms of interactivity, but current browsers are relatively flexible and client-side scripting with AJAX tools generally provide acceptable options for business applications.

Forms and reports can be used for internal applications as well as public Web sites for use by customers and vendors. The primary differences are in terms of security and usability. The examples in this chapter are geared toward developing internal applications, as opposed to public Web sites. Keep in mind that even public Web sites need backend tools to handle administrative tasks. The forms and reports you create here will work well for those administrative tasks. Public-based Web sites are often simpler where each page handles a single concept and the focus lies on connecting the pages. So, if you can build the three main types of forms in this chapter, you can apply the same concepts to building public Web sites.

This chapter uses the JDeveloper tool and Oracle’s Application Development Framework (ADF). JDeveloper should be installed on your workstation computer. The applications will be built and tested within JDeveloper. In a production environment, you would then deploy the application to a standalone application server and configure security options to control access to the application. But, first you have to build the application, and it is easier to understand the process if these extra steps are ignored for a while.

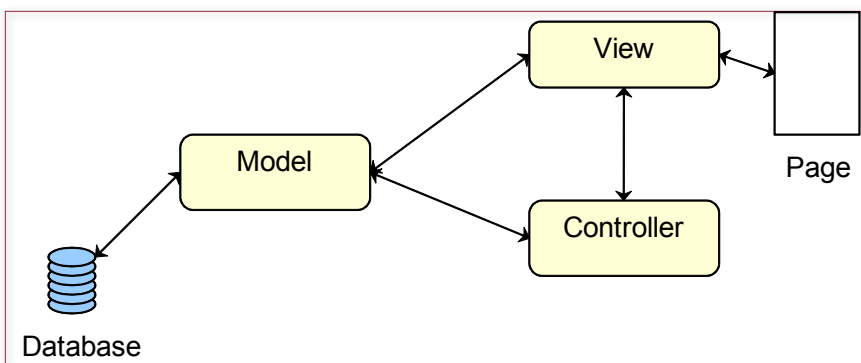
Model-View-Controller

The Oracle application development process generally follows an architecture known as Model-View-Controller (MVC). It is an approach that has proved useful for building complex applications—particularly Web applications that interact with users. Shown in Figure 6.1, each of the three items represents component or objects within the application. Each component is dedicated to a specific set of tasks. By separating these tasks, it is easier to build and maintain the application. In terms of building applications within Oracle, it also means that Oracle developers have created generic tools that work within each segment—saving you huge amounts of time and energy. Typically, these components are built as software objects within the application. By default, JDeveloper stores the model and database components in a separate project named Model. The pages and control code are stored in a separate project called ViewController. It is possible to change the names of the projects and the pages and objects can actually be stored anywhere. However, it is good practice to stick with the default choices so that other developers will understand the goals.

Model components are software objects that hold data from the database that will be needed for the page. The pre-built Oracle ADF model components know how to retrieve data and write changes back to the database. For the most part, you simply need to select tables in the database and the wizards build the matching model components—including the code that handles the data transfers. Technically, JDeveloper also allows you to design in the other direction: build Model components and generate the necessary tables. However, most applications are easier to construct if you design the database first.

View components essentially consist of the page design and layout. Typically you place panels on the page to define the page structure, such as the number of columns or the existence of header and footer panels. Then you place display objects within the panels. Oracle ADF includes objects that support simple text, text input boxes that connect to the data model, and even tables that support the display and editing of data in the form of rows and columns. Standard HTML-type components are also supported, including objects to display image files, radio buttons, check boxes, and drop down lists. Each of these can be standalone, or they can bind to the data in the model. Data-bound controls automatically include code to retrieve data from the model, display it, and store changes back to the model structures.

Figure 6.1



Controller objects are related to the view objects—and Oracle often places them into the same category. But controller objects represent logic and code—not the display of data and pages. The logic controllers are used to determine page navigation and respond to user input and choices. The goal is to keep the code separate from the display of the page. That way the page can be designed and altered without tripping over the code and each component can be built and edited separately. Of course, code that depends on objects on a page would have to be rewritten if the page objects are deleted or substantially modified.

As you build forms in JDeveloper, you will encounter these three types of objects, so it is important that you understand the purpose of each object type.

Action

Start JDeveloper with the default role.

Use menu: Application/New.

Name: AllPowder06, Package prefix: AP06.

Pick Fusion Web (ADF) template.

Create default project Model with ADF objects.

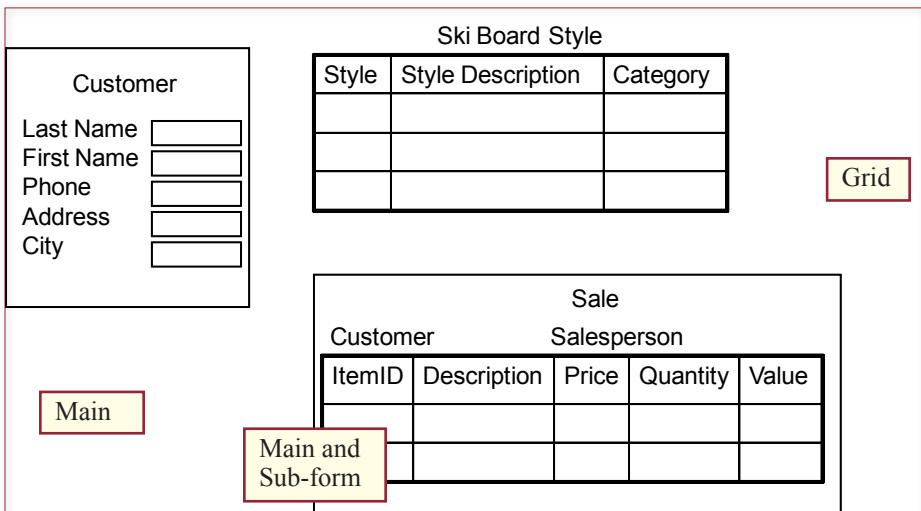
Create default project ViewController with ADF Faces.

Finish with defaults.

Case: All Powder Board and Ski Shop

The primary application at All Powder Board and Ski Shop is the need to track sales and rentals. Of course, these applications also require you to build forms and reports for inventory items and customers as well. Eventually, you will have forms that store data into each of the tables in the relationship diagram. However, before you leap to the forms wizard, make sure you understand the three major form types shown in Figure 6.2: main form, grid form, and main with subform. A main form shows one row of data at a time, such as a form to edit basic information about one customer. A tabular or grid form appears similar to the table view in that it shows several rows at one time. Main and subforms combine the two: the main form shows one row of data from one table, and the grid subform shows matching rows from a related table. The classic business example is the Sale form and SaleItem grid, where the main form shows data from one sale, and the grid shows the repeating items purchased and stored in the SaleItem table. At

Figure 6.2



this point, your responsibility is to examine the business operations and determine the best type of form to handle each operation.

Lab Exercise

All Powder Board and Ski Shop Forms

Many of the forms in an application are straightforward main forms. Users want to see data for one row—such as one customer or one employee. You generally create main forms when you need detailed control over the layout. Fortunately, the Oracle wizards make it relatively easy to create main forms. One of the useful, but sometimes tricky, aspects of Oracle forms is that they run in a disconnected mode. Data modified on the client form is not written to the database until the user tells the system to save all of the changes.



Activity: Create Basic Main Forms

Figure 6.3 shows a simple version of the form to edit customer data. It is similar to the form you built in Chapter 1. In its simplest layout, the main form contains labels and text boxes for each column in the table. You can enter any text into the label to help tell the user what data is to be entered into each text box. The data on the form is bound to the database table. Changes made to the data in the text boxes are automatically written to the database table. However, these changes are written only at certain times—when the user clicks the Submit button to save the data. The importance of the main form is that you have considerable control over the layout and presentation of the items. You can change the image of the form by setting the properties for the form or the controls to control descriptors such as size, position, and color. You can add new controls to display images or include buttons to delete or find records. Notice the use of the list box to choose the gender values. Oracle has tools to define a list of values (LOV) which can draw data from a fixed list or from another table. It is useful when you need to restrict data

Figure 6.3

The screenshot shows a web browser window displaying an Oracle Forms application titled "Edit Customer". The form contains several text boxes with labels: Customerid (1), Lastname (Jones), Firstname (Jack), Phone (111-222-3333), Email (JonesJ202@msn.com), Address (123 Main), City (Sacramento), State (CA), and Zip (95838). There is a Gender dropdown menu with "Male" selected, and a Dateofbirth dropdown menu with "Female" selected. At the bottom, there are buttons for "First", "Previous", "Next", "Last", and "Submit". Annotations with arrows point to the "Text box with label" (top right), "Gender Combo Box/LOV" (middle right), and "Record navigation" (bottom center).

entry to a specific set of items. Oracle forms also automatically include a date picker which provides a popup calendar that users can use to enter dates.

Creating any form takes several steps. JDeveloper creates a list that you can follow if you forget the basic steps—but the list has many options that are not required. Once you create the project and define a database connection, creating a main form requires only two or three steps. (1) Add a data model that defines the data tables needed. (2) Create a list of values if needed. (3) Create a display page and drag the data onto the page. Oracle provides many other options to improve the layout and add security. For now, concentrate on the main steps needed to create the form. Later, you can edit and improve the forms.

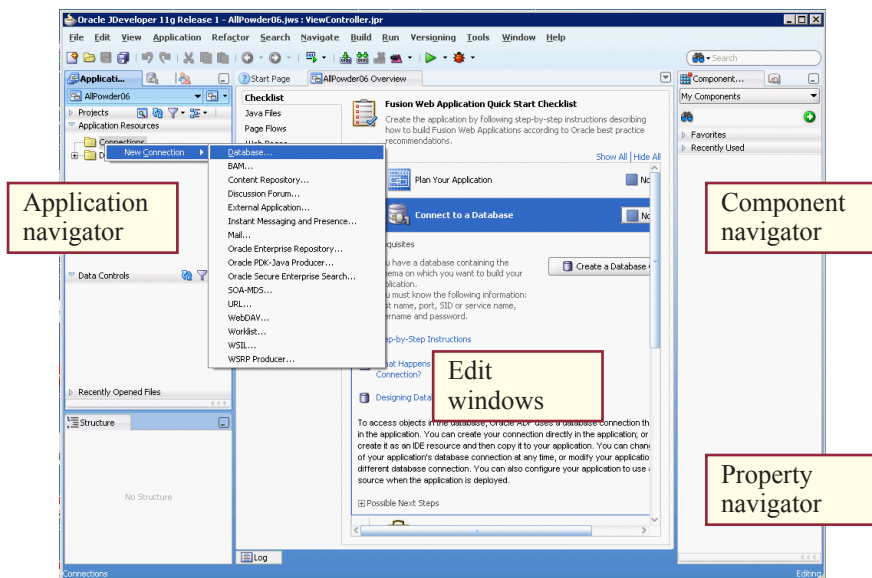
The project needs a connection to the database. Figure 6.4 shows the main JDeveloper screen with the two methods of creating a database connection. First, note the four major sections in JDeveloper: (1) Application Navigator that shows projects, resources, and eventually page lists. (2) Component Navigator that shows lists of components that can be added to pages. (3) Edit Windows that enable you to edit pages and components. (4) Property Navigator (not in the picture) that shows properties or attributes for components selected in the active edit window.

To create a database connection, you can use Step 2 in the checklist or open the Resources section in the Application Navigator, right-click the Connections entry and choose New Connection/Database. At that point, the database connection window opens and asks for the standard connection information. Give the connection a name you will recognize later, such as AllPowder06. Enter the username and password, change the server SID if necessary, and click the button to test the connection. The connection information was created by the database administrator

Action

- Use Checklist Step 2, or Right-click Connections, New Connection/Database.
- Name it (AllPowder).
- Enter standard login information (Username/Password).
- Set the server and SID if needed.
- Test Connection!

Figure 6.4



when the account was created. If you created the account yourself when you installed Oracle, go back and look at your notes. Note that you should leave the default option checked to “Save the password.” This option means that the forms will run without asking for a password. Security options can be changed later, and it is easier to develop forms and experiment if you do not have to log in every time you test a form.

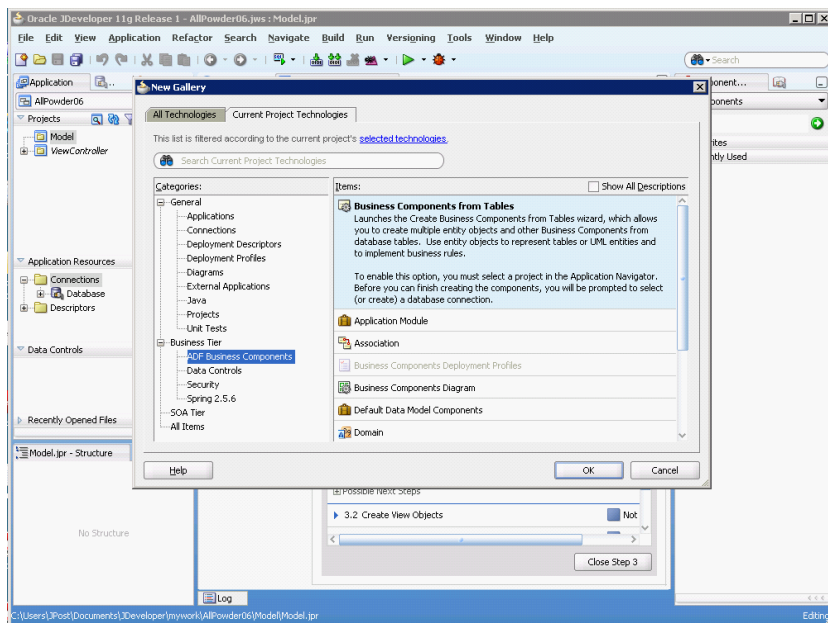
One of the most important steps in building a form is to create a data model. The data model is a buffer between the database table and the objects on the display page. You define it much the way you define tables and relationships in the actual database. For this reason, JDeveloper has a tool that makes it easy to create a data model based on selected tables in the database. You simply select the database tables and JDeveloper uses the database information to create matching objects in your project model. The model objects have built-in tools that know how to retrieve data and write changes back to the database tables—using SQL statements. The process for a main table like Customer is straightforward because it requires the use of only a single table.

You can use Checklist Step 3.1 by expanding Step 3, clicking the substeps button, and clicking the link for Step 3.1. Or, you can expand the Projects section in the Application Navigator, right-click the Model entry and choose New. As shown in Figure 6.5, expand the Business Tier in the list on the left, pick ADF Business Components, and select Business Components from Table. The first time you build a data component in a project, you will be asked to initialize the project con-

Action

Use Checklist Step 3.1 or, Expand Projects, right-click Model, New. Pick Business Tier/ADF Business Components. Click Query button. Select Customer table to the right side. Updatable View: Select Customer table. Finish with defaults.

Figure 6.5



nection. If necessary, select the connection and keep the default SQL flavor and types.

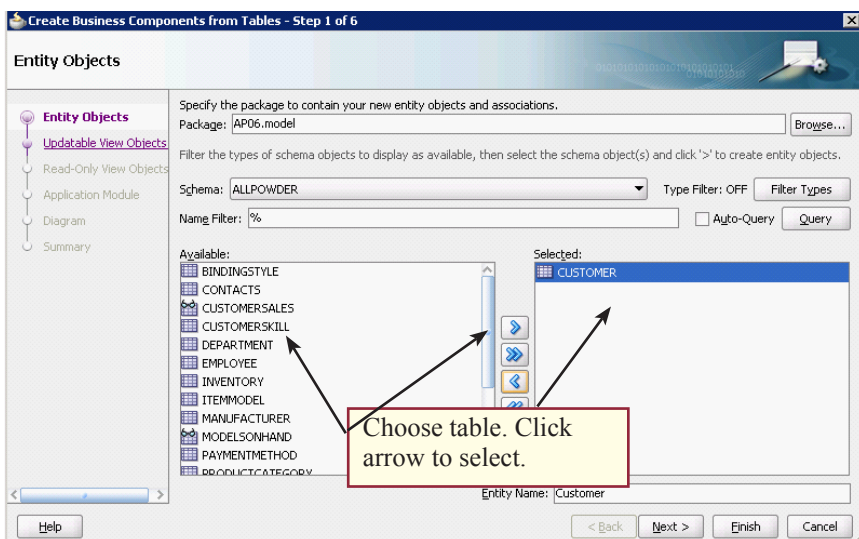
The main step is to choose the entity objects (tables) needed. Figure 6.6 shows the main process. First click the Query button to see a list of tables in the database. If the list is huge, you can enter a value in the filter column and then click the Query button to see only the matching values. For instance, you could enter Cust% to see only the tables that begin with letters “Cust”. When you find the desired table (Customer), click it to select it in the list on the left. Click the arrow (>) button to move it to the select list on the right. Only the Customer table is needed for this simple form, so you can move to the Next page of the wizard. More complex forms might require additional tables which can be selected using the same steps.

The next two steps are similar. You are asked to choose which tables will be updatable and which are read only. Because you want users to be able to edit the data in the Customer table, again select it in the list on the left and move it to the Selected list on the right—for the Updatable View Objects screen. The next screen asks you to choose read only tables and this form has none. So keep the default values and work your way through the end of the wizard.

Making forms easy to use is often a difficult task. One key feature is to give users lists when you want them to enter certain values. For the most part, never trust users to enter data—they will find ways to misspell, abbreviate, skip, or mangle the data entry. If you need data to be consistent, create a list and let them choose from the list. Options can also be displayed on the screen using radio buttons (single selection) or check boxes (multiple selections). Oracle has several controls to create and display a list of values (LOV). The simplest version is the default Choice List and it will work well for the Gender column in the Customer table. The objective is to display a list of choices (Female, Male, Unidentified) to the user who will pick one.

Creating a list of values requires a new view object that contains the list of items to be displayed. In the Application Navigator, expand the Projects section, and then the Model project. Right-click the AP06.model. (It will have a different

Figure 6.6



prefix if you used a different package name). Choose the option to Create a New View Object. Name it GenderLOVView to remind you that it holds the list of values for gender. As shown in Figure 6.7, choose the option to “Select Rows populated at design time (Static List). Gender requires only three choices and they are unlikely to change so they can be static. You will now be asked to define a table with columns and then add rows of data choices to that table. But this pseudo-table is stored only within the data model and not on the database. The main step is to add attributes, so click the plus sign and add a new attribute (Gender). On the properties form for this attribute, check the box to indicate that it is the

“Key Attribute.” For gender, the problem only needs this single column—unless eventually you want to translate gender to multiple languages. Then you would need to alter the Gender column in the Customer table to hold a number, and build the list of values with two columns: A number and a name—where the name can later be translated to multiple languages, but each name will result in the proper number being stored in the table. In situations of this sort, you would add two columns to Gender table: GenderID and GenderName, and make the GenderID column the key attribute. Anyway, to keep it easy for now, just use the single Gender column.

On the next form, you will enter the gender values to be displayed. Click the plus sign to add a row. Click on the row and type Female as the value. Repeat the process until you have entered the three rows of choices. Finish the steps of the

Action

Right-click Projects/Model/AP06.model.

Create a New View Object.

Name: GenderLOVView.

Select Rows populated at design time (Static List).

Add Attributes with the New button.

Name: Gender, Key Attribute (check box).

Static List: Click the Plus sign.

Add three rows: Female, Male, Unidentified.

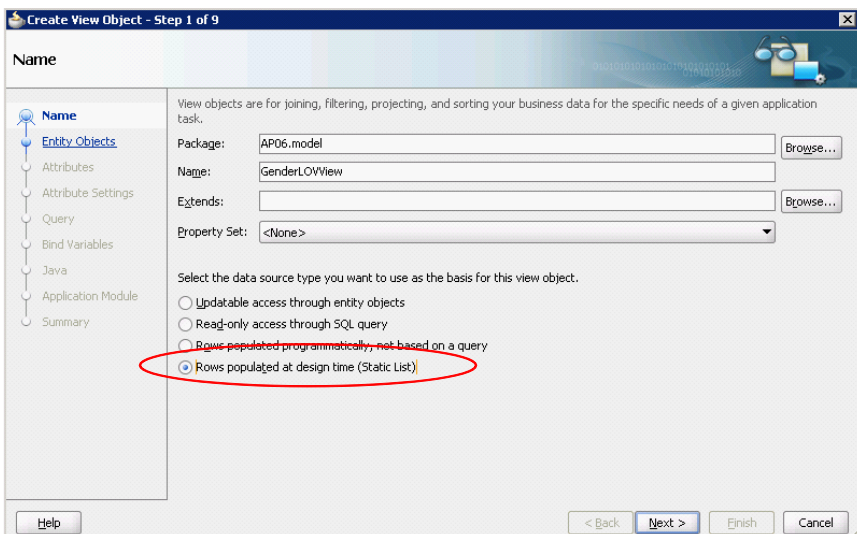
Set check box for Application Module.

Finish with defaults.

Choose Combo Box with List of Values.

Move Gender column to Selected side.

Figure 6.7



wizard using the defaults, but be sure to check the box to assign the Application Module on the next-to-last page—accepting the default module names.

The resulting GenderLOVView also lets you specify the type of control that should be used to display the data. As shown in Figure 6.8, select the List UI Hints tab on the left side of the edit window. Accept the default display of Choice List, but take a look at the number of possible options. You also need to indicate which column(s) will be displayed when the list is opened. With only one column, the choice is easy: move the Gender column to the Selected

Action

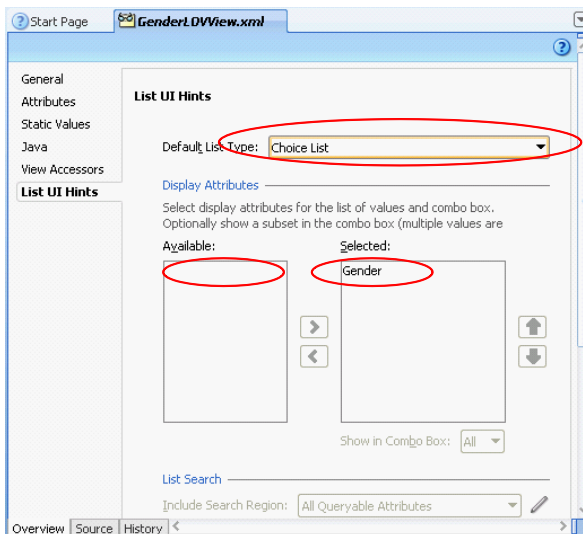
Open Customer table object in model.
View Accessors tab, click plus sign.
Move GenderLOVView to right side.
Attributes tab, select Gender column.
Double-click to edit Gender attribute.
Select control Hints tab.
Set Control Type to Combo Box with List of Values.
Select Validation tab.
Set Rule Type to List.
Choose GenderLOVView1/Gender.
Failure Handling tab, enter message (Female, Male, Unidentified).

box. If your list contains an ID number and a matching value, you would normally choose only the matching value and leave the ID number hidden. The list of values object has now been created, so you can save it and close the edit window.

To use the list of values, you need to assign it to a column (Gender) in the Customer table. Actually, you can do two things: (1) Declare the list as a validation rule so users can never enter anything else, and (2) Assign the LOV so the form builder automatically creates the choice list box.

The validation rules are best declared in the underlying Customer table model—where they will automatically be assigned to the corresponding Customer-View object. Find the Customer object in the Model section of the Application Navigator and double-click it to open it in the edit window. Select the View Accessors tab and click the plus sign to create a new accessor. As shown in Figure 6.9, find the GenderLOVView use the arrow button to move it to the right-side selection list. Click OK to close the box.

Figure 6.8



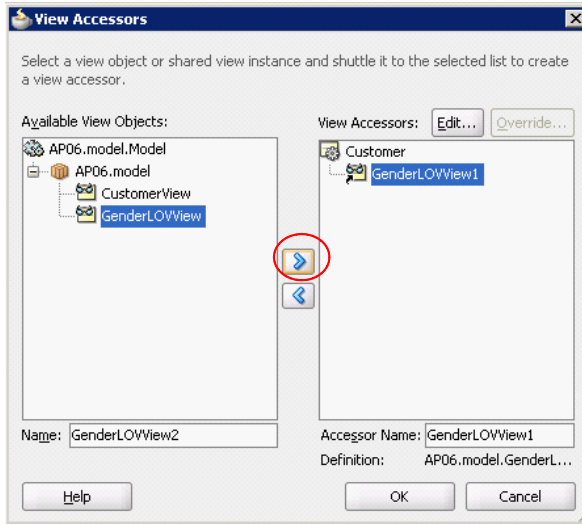
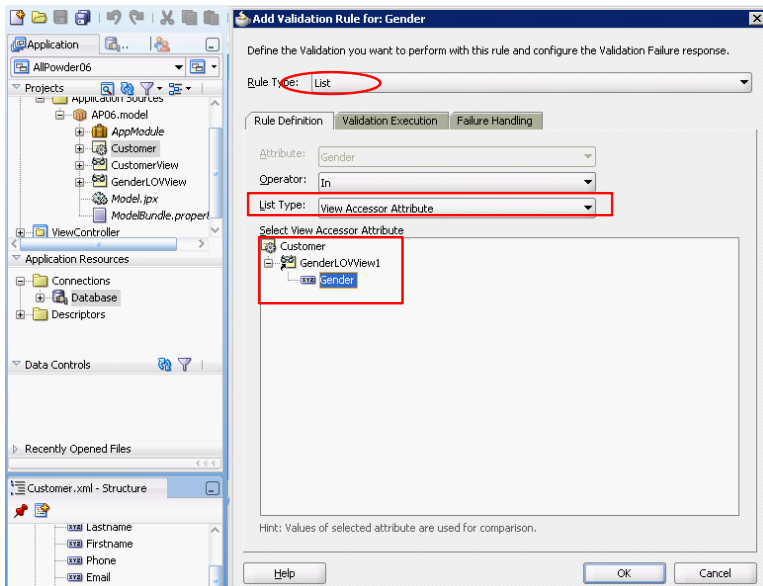


Figure 6.9

Select the Attributes tab and double-click the Gender column to edit it. Select the Validation tab in the Gender edit window and click the New button to add a new validation rule. As shown in Figure 6.10, choose List as the rule type and View Accessor Attribute as the list type. Then select the Gender GenderLOView1 view and the Gender column in the tree. Select the third tab in the edit window labeled Failure Handling. Enter a message to be displayed to the users such as: Please select from the list: Female, Male, or Unidentified. Click OK, save, and close the Customer edit window. Any forms built with this view will now accept only the values in the gender list.

Figure 6.10



You are almost finished. The next step is to assign the LOV to the Gender column so that the choice list will be displayed automatically. This assignment takes place in the CustomerView object (not the table). In the Model section of the Application Navigator, double-click the CustomerView object to edit it. Select the Gender column but instead of trying to edit it, scroll the edit window down to see the List of Values: Gender section below the column list. Click the plus sign to create a new list of values.

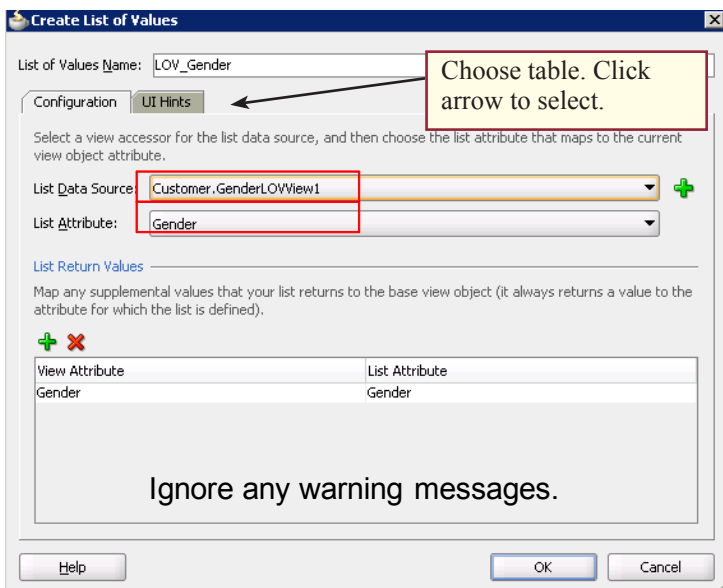
Action
Open Customer View in model.
Select Gender column.
Scroll to bottom.
List of Values: Gender, click Plus sign.
List Data Source: GenderLOVView1.
List Attribute: Gender.
Ignore warning messages.

Figure 6.11 shows the basic steps of assigning the gender LOV to the Gender column in the CustomerView object. For the List Data Source, choose the GenderLOVView1 object that you built a minute ago. For the List Attribute, choose the Gender column. If your list has multiple columns, this is where you need to pick the Key column which contains the value in the list that will be inserted into the Customer table. You can also use the tab to override the default type of display, but the default Choice List is best for the gender data.

Finally, you are ready to build the display page. With the data and list of values created and assigned the process of building the page is relatively easy—leaving you free to concentrate on the layout and design for the page. You can use Step 5.2 in the checklist, or create a new page from the Application Navigator. Because the checklist is easy to lose, you should learn to create objects in the navigator. Expand the ViewController project in the navigator, right-click the Web Content entry. In the object tree of the pop-up box, expand the Web Tier and choose the JSF entry. Pick the JSF Page option from the list of JSF items.

Figure 6.12 shows the basic options for a JSF Page. First enter a name (Edit-Customer.jspx). The name cannot include spaces and must end with the jsp file

Figure 6.11

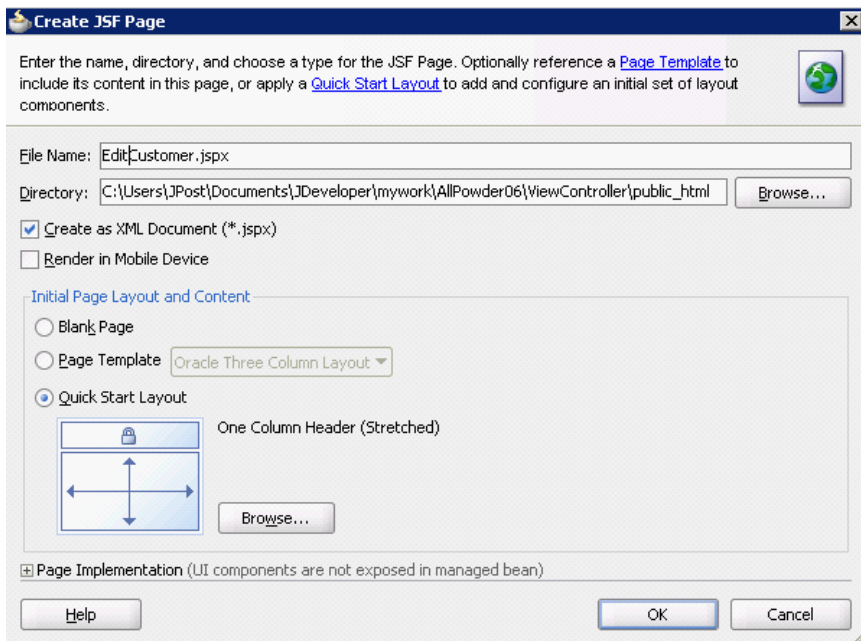


type. Leave most of the default values, but click the Browse button to see the Quick Start Layout choices. For this simple form, the simple One Column Header (Stretched) layout is fine. Eventually, you will want to choose more complex layouts for your Web site so you can add page navigation elements. You might want to hire a graphics designer to help determine the best design and structure for the entire Web site. You can even build a custom template which can be used as the foundation for all of your pages—a useful approach for large projects. A set of templates ensures all developers build similar pages. For now, just get familiar with the standard options so you have some idea of the basic choices.

Once the page is created, add a title at the top by typing Edit Customer into the top panel of the page design. Each page is essentially an HTML page, so select/highlight the new title and choose the Heading 1 style from the second drop-down list. You can use cascading style sheets to assign consistent styles to every page. In the Component Navigator at the top right of JDeveloper, change the ADF Faces selection box to the CSS options. Drag the built-in JDeveloper style sheet onto the main edit page. A pop-up box lets you rename the file to: pow-

Action
 Checklist Step 5.2 or
 ViewController project, right-click Web Content, New JSF Page.
 File Name: EditCustomer.jspx.
 Choose Quick Start Layout (One Column Header).
 Type title: Edit Customer.
 Set it as Heading 1.
 In Component navigator, pick CSS from dropdown instead of ADF Faces.
 Drag JDeveloper.css onto the form and rename as Powder.css.
 In Application navigator, refresh Data Controls section.
 Drag CustomerView1 object to form page (bottom).
 Check two boxes: Navigation and Submit buttons.
 Verify Gender column is set to ADF Select One Choice.

Figure 6.12



der.css. Later, you can edit the styles in this file to match your preferences. For now, simply observe that the title has a nicer appearance.

Now for the big step. Refresh the Data Controls section of the Application Navigator and expand its subtree. As shown in Figure 6.13, drag the CustomerView1 object and drop it onto the main section of the form design. In the pop-up menu select the Form and ADF Form options. In the resulting Edit Form Fields screen, you can use the up/down arrows to change the initial order of the fields. You can also verify the type of display box for the Gender and Date columns. More importantly, check the two boxes to Include Navigation Controls and Include Submit Button so the user can see and edit the Customer data.

That is it. You are now ready to test the form. Click the Save All button to be safe. Right-click the form and choose the run option. You might have to wait a minute or so for the application to build and the internal Web server to start. Figure 6.14 shows the form. You can use the navigation buttons (Next, Last, and so on) to scroll through the customers. The one glaring problem is that the Email textbox is too wide to fit on most displays. It is a good idea to fix this problem first, so close the browser window.

Action

Save everything (button).

Right-click the form and choose Run.

Wait.

Use red square stop button to stop the form.

Select the Email box.

Expand Properties/Appearance.

Change Columns from bindings.Email... to bindings.Lastname...

Select Gender combo box.

Change columns to bindings.Lastname...

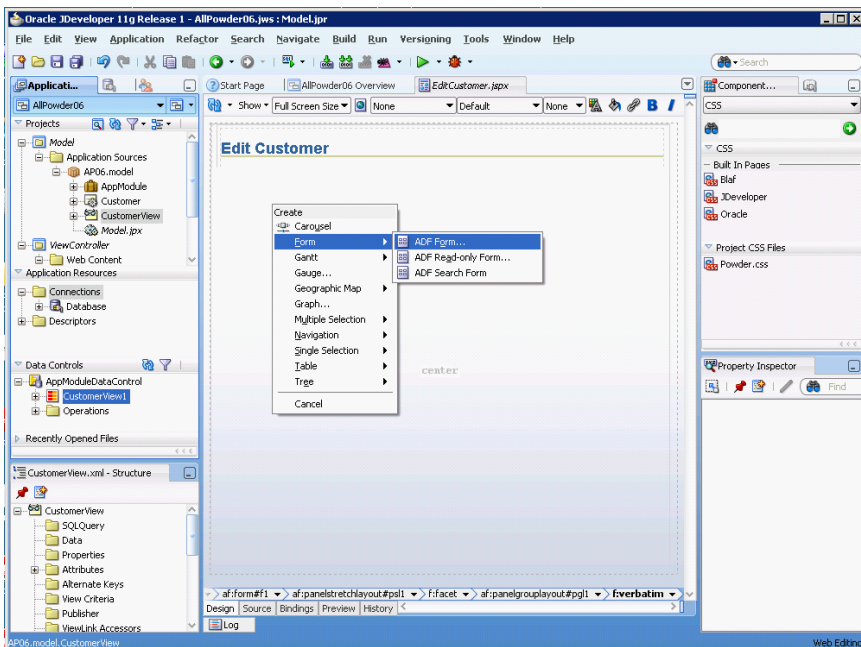
Run form.

Test gender combo box.

Test date picker.

Stop form and close all design pages.

Figure 6.13



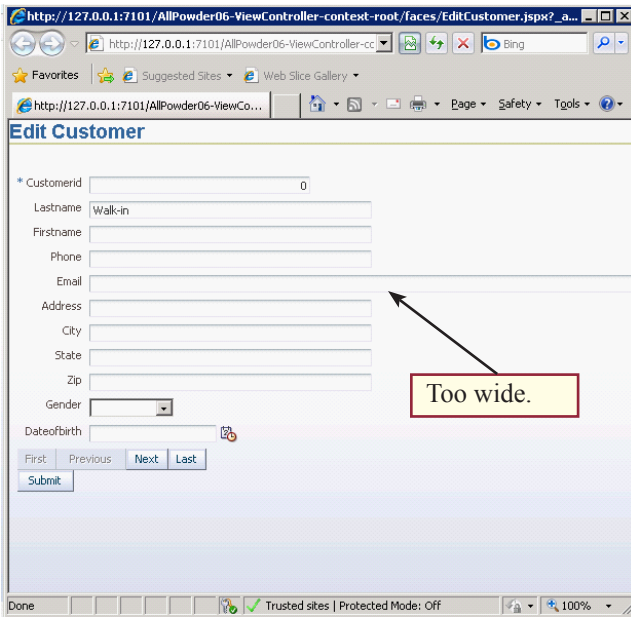
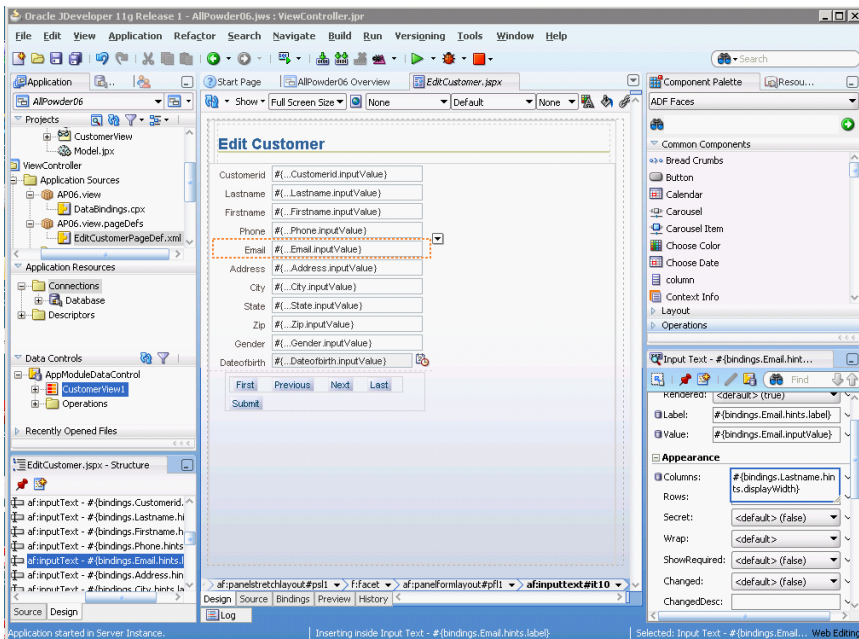


Figure 6.14

As shown in Figure 6.15, select the Email textbox control in the edit window. In the Properties navigator on the bottom right, expand the Appearance section. Notice the Columns property contains an entry that sets the size based on the display width of the Email column: `#{bindings.Email.hints.displayWidth}`. You could try to find where this value is stored and change it, but an easier solution is to realize that it is best to keep all of the widths the same, so edit the entry to use the width

Figure 6.15



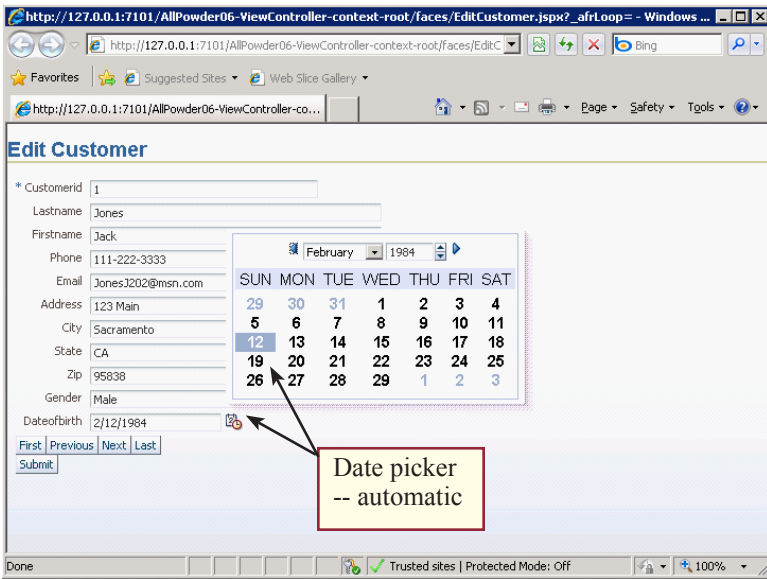


Figure 6.16

of the Lastname column instead: `#{bindings.Lastname.hints.displayWidth}` by changing Email to Lastname.

Now you can run the form, scroll through the entries, modify any of the data, and click the Submit button to save your changes to the database. Figure 6.3 at the start of this chapter shows the LOV choice list used to edit the gender value. Here, Figure 6.16 shows the use of the automatic date picker to display a calendar. You should also test the form by trying to enter invalid data. If you find problems, you could add more validation rules and LOV selection boxes to the form. The one catch is that these changes are easiest to make to the underlying table and view in the Model section. If you make changes to the view, they will not automatically be added to the page. Instead, you generally have to delete the existing controls on the page, edit the underlying view, and then re-drag the view onto the page to create new controls. So, you should figure out what controls and validation are needed before you get too involved in playing with the design and layout of the page.



Activity: Create Grid Forms

Grid forms are another simple type of form. They are used when a table has a limited number of columns and rows. The columns should all fit on one screen—users find it difficult to edit data if they have to scroll horizontally. The number of rows should be limited because the grid form has few methods for searching, and users should not be forced to scroll through thousands of rows to change one piece of data. Figure 6.17 shows a grid form for the SkiBoardStyle table. Notice that the data in this table is generally used only to provide consistent values to other tables. This form will generally be used only by an administrator once in a while to

Action

Application Navigator: Model/New.

Business Components: Business Components from Tables.

Click Query button, select SkiBoardStyle table to right.

Updatable View, select SkiBoardStyle table.

Read-only, none, default.

Finish with defaults.

modify or add a style. The data all fit on one screen, making it easy to find the items to be altered and to compare the various entries across the rows. In practice, you will use grid forms for similar tasks aimed at administration. Think hard before you use one of these forms for general users. Although you have some control over the form design, your options are limited, so users need to know what they are doing.

You create a grid form in much the same way as a main form. You need a data model to hold the data for the form. The SkiBoardStyle table also links to the ProductCategory table by asking users to assign a Category such as Ski or Board to each style. To ensure consistency and referential integrity, this data should be entered through a list of values that is filled from the ProductCategory table—so you will have to create a view object for that list as well.

In the Application Navigator, right-click the Model project and choose New. As with the main form, select the Business Components items and pick the Business Components from Tables option. Click the Query button to see the list of possible tables and select the SkiBoardStyle table by moving it to the right-side panel. As before, this table needs to be in the selected list of updatable views. The form does not need any other read-only tables, so accept the default options and finish the wizard.

You also need to create a view object for the list of values from the ProductCategory table. In the Projects section, expand the Model entry and right-click the

Action

Projects/Model/Ap06.model: New View Object.

Name: ProductCategoryLOVView.

Read-only access through SQL query.

Write SQL.

Accept most defaults, but set Category as Key Attribute.

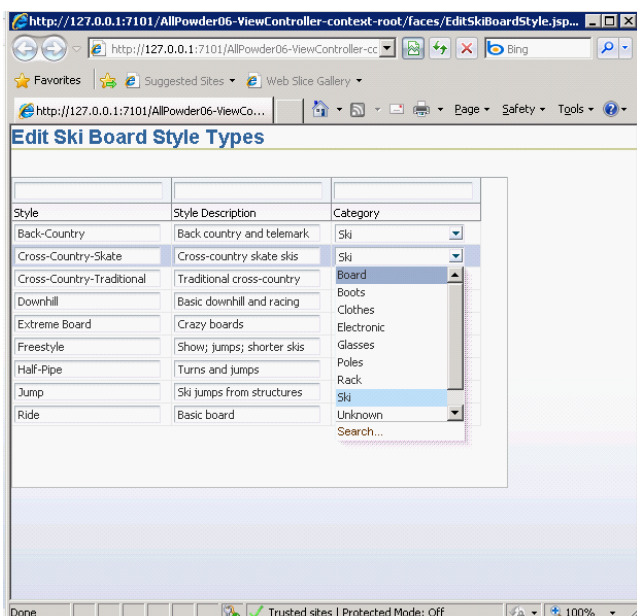
Set Checkbox for Application Module.

Use defaults to finish.

List UI Hints tab: Keep default: Choice List

Move Category column to Selected box.

Figure 6.17



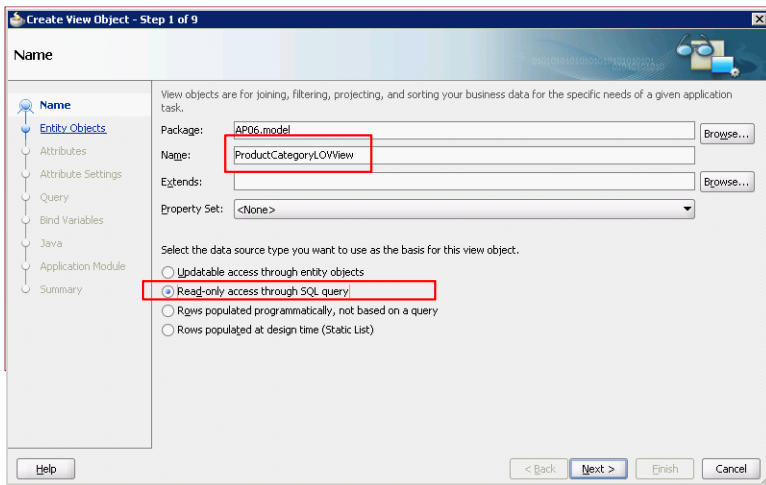


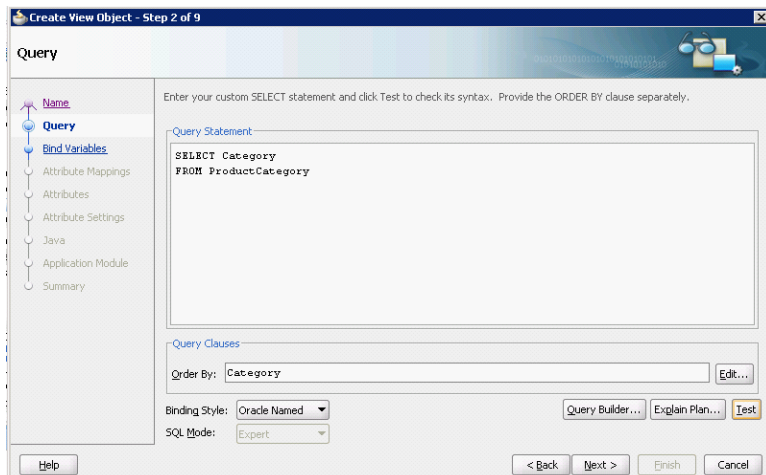
Figure 6.18

AP06.model entry. Choose New View Object from the menu. Figure 6.18 shows the choices that must be made. Name it something like ProductCategoryLOV-View so you recognize later that it is a list of values object. Choose the Read-only access through SQL query option. On the next page, write an SQL query to retrieve the list of categories in the ProductCategory table:

```
SELECT Category FROM ProductCategory ORDER BY Category
```

Figure 6.19 shows the basic form. Notice there is a button to help build queries, but the process might not be useful. You do need to click the Test button to ensure the query is valid. If you need help building the query, use SQL Developer, or open the Database Navigator window in JDeveloper—the tab is just to the right of the Application Navigator tab at the top-left of the window. When you are certain the query retrieves the data you need, copy the SQL and paste it into list of values window. On the next screen, mark the Category column as a key attribute. On the next-to-last screen, check the option to save the object in the default application module and finish the wizard.

Figure 6.19



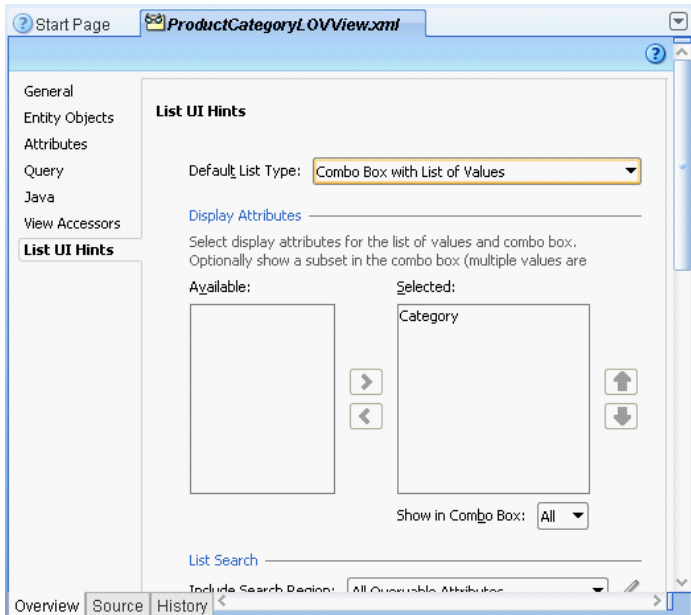
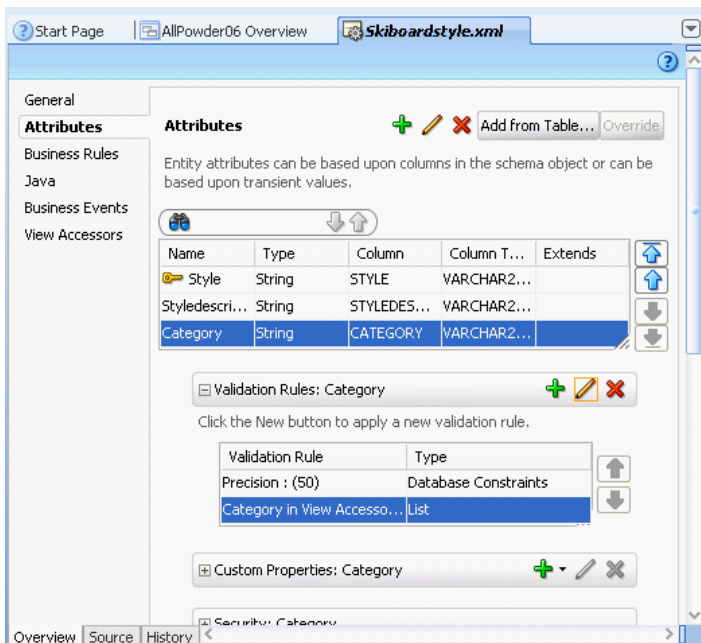


Figure 6.20

As shown in Figure 6.20, select the List UI Hints tab. This time, change the default from Choice List to: Combo Box with List of Values. Be sure to move the Category column to the Selected box on the right side. The combo box approach has some options that are not available on the standard choice list control. The problem with LOV controls is that the Web browser has to load the entire list of options to display the page. If the list is huge, it can slow down the time it takes to load and display the page—even if the user does not need to use that list.

Figure 6.21



The List of Values option chosen here avoids that problem by initially loading only a portion of the data. If the user needs to see more, he or she can click the “Search” option and enter a pattern to download only the items that are needed. This process might take a couple of extra steps, but it is substantially more efficient for displaying Web pages.

You still need to add the list of values to the SkiBoardStyle table. Because the interface will use a combo box, users might try to type in random values, so you need to add the validation rule as well as the LOV display.

First, open the SkiBoardStyle table in the Model section. Select the View Accessors tab and click the Add button. Move the new ProductCategoryLOVView to the right panel. In the Attributes tab of the SkiBoardStyle table, select the Category table. Click the plus sign in the Validation rule to add a new one. Choose List for the rule type and View Accessor Attribute for the rule type. Choose the new ProductCategoryLOVView1 and then the Category column. Switch to the Failure Handling tab and enter a message of the form: Please select category from the list. Figure 6.21 summarizes the changes for the SkiBoardStyle table. Save and close the table edit windows.

Action
Model/SkiBoardStyle table: Edit/double click.
View Accessors tab, click plus sign to add.
Move ProductCategoryLOVView to right panel.
Attributes tab, select Category.
Validation rule, click plus sign to add.
Rule Type: List.
List Type: View Accessor Attribute
Choose ProductCategoryLOVView1 and Category.
Failure Handling message: Please select category from list.

Figure 6.22

The screenshot shows the 'Create List of Values' dialog box with the following configuration:

- List of Values Name: LOV_Category
- Configuration tab selected
- Select a view accessor for the list data source, and then choose the list attribute that maps to the current view object attribute.
- List Data Source: SkiBoardStyle.ProductCategoryLOVView1
- List Attribute: Category
- List Return Values section:

View Attribute	List Attribute
Category	Category

Buttons at the bottom: Help, OK, Cancel.

The next step is to add the LOV to the view, so open the SkiBoardStyle View in the Model project. Choose the Attributes tab to see the list of columns. Select the Category column, and scroll down the edit window if necessary. At the List of Values: Category entry, click the plus sign to add a new LOV. For the List Data Source, choose `Skiboardstyle.ProductCategoryLOVView1`, and select `Category` for the List Attribute. Ignore any warning messages that might be displayed. Figure 6.22 shows the assignment choices.

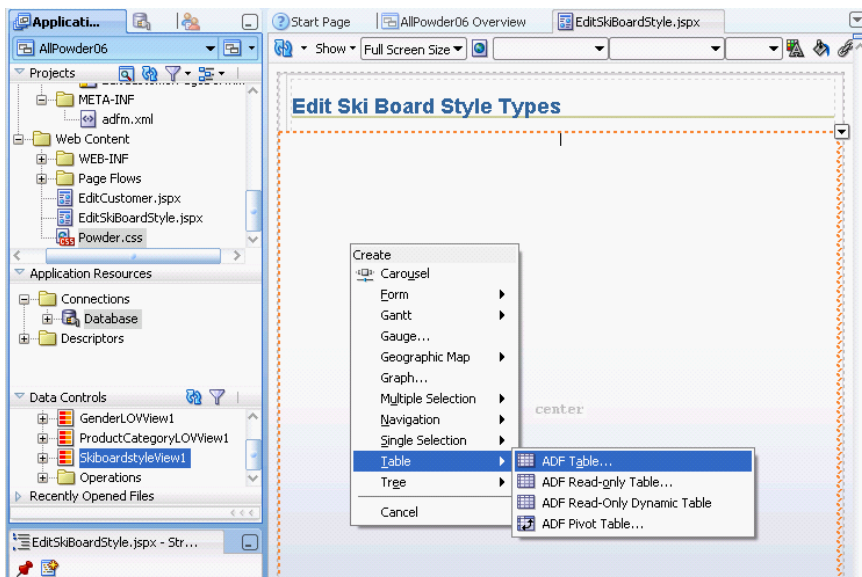
Action
Model/SkiBoardStyle View: Edit/double click.
Attributes tab, select Category column.
List of Values: Category, click plus sign to add.
List Data Source: Skiboardstyle.ProductCategoryLOVView1.
List Attribute: Category.
Ignore error messages.

With the data sources defined and the validation rule and LOV assigned, creating the page is straightforward. First create a blank JSF page. In the Application Navigator, expand the ViewController project, right-click the Web Content entry and pick the New option. In the module-selection form, choose the JSF entry under the Web Tier and select the JSF Page option in the right-side panel.

In creating the new page, enter the name (`EditSkiBoardStyle.jspx`) and select the same layout as you did for the last page. Most of your Web pages will have the same layout—to ensure a consistent style. Just as you did with the main form, type a title onto the top of the form, such as `Edit Ski Board Style Types`. Assign it a Heading 1 style and drag the `Powder.css` style from the Component Navigator onto the page design. Later, if you want to change the Web site appearance, you simply edit the `Powder.css` file and all pages will reflect the changes.

As shown in Figure 6.23, the real difference with the Grid form lies in adding the data view to the page. Refresh the Data Controls section so it has the current information. Find the new `SkiboardstyleView1` and drag it onto the form. This time, instead of choosing the Form option, select `Table` and `ADF table` from the

Figure 6.23



pop-up menu. The following page will list the columns and enable you to change the order if desired.

Figure 6.24 shows the more important options at the top of the form. You should check all three boxes for: Row Selection, Filtering, and Sorting. The last two are particularly important if the form has many rows of data—they enable the user to reduce the display list and sort it to work with a smaller set of data. You will get the opportunity to work with these options when you run the form.

Right-click the form and choose the option to run it. The first pass design is not bad, but it is likely that the column widths are too narrow and you cannot see the entire text for each column entry. Also, the text label “Styledescription” is annoying. Stop the browser and edit the page design. If you can hold the cursor on the dividers between the columns, you can drag them to make the wider—although the last column is difficult. It is usually easier to select a column and edit its properties. Expand the Appearance section in the Property Navigator and set the widths to the columns as: 150, 150, and 140. Be sure the panel holding the columns is wide enough to hold the new widths (at least 440 units).

To change the Styledescription label, you can check the Header Text property to see that it is bound to hints.Label property of the original table. Instead of overwriting the binding value in the properties, it is better to go back and edit the source. That way it will be fixed for every other form that might want to use it and be easier for another developer to find if it needs to be changed in the future. In the Application Navigator, open the SkiBoardStyle table (not the view). Select the Attributes tab and double-click the StyleDescription entry to edit it. For Label

Action

Right-click the form and run it. Wait.

Notice the column widths.

Stop the browser.

Adjust the column widths by dragging the borders in the design.

Fix the title/label for Style Description.

Model: Edit Skiboardstyle View.

Attributes tab: Edit Styledescription

Set Label Text: Style Description.

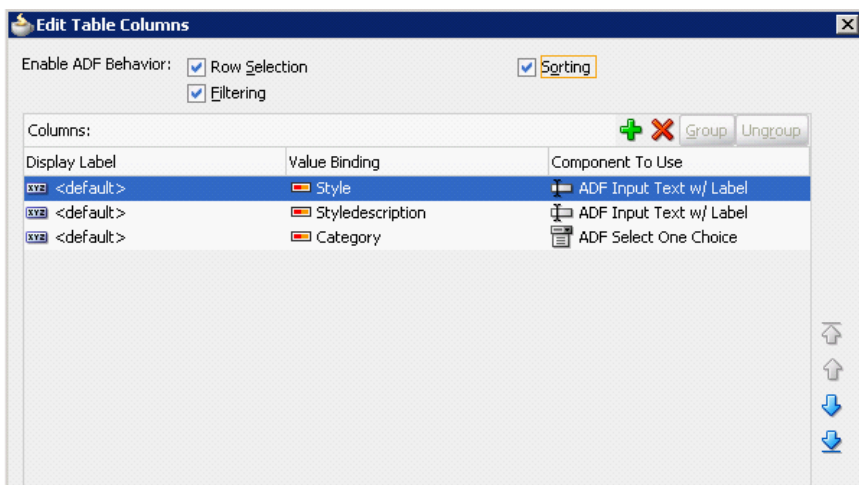
Save everything and run the form.

Test the LOV for Category.

Test the sorting by clicking the column headings.

Test the filter. Enter Board in the box above the Category heading and press <Enter>.

Figure 6.24



The screenshot shows a window titled "Search and Select: Category". At the top, there is a search bar with a dropdown arrow and the text "Search". Below the search bar is a text input field labeled "Category" containing the text "B%". Below the search bar and input field is a list box containing three items: "Category", "Board", and "Boots". The "Category" item is currently selected and highlighted.

Figure 6.25

Text, enter: Style Description, with the space and proper capitalization. Close the edit window.

Run the form again and verify that it works. Use the combo box to select one of the entries for Ski or Board. Click the “Search” option at the bottom of the list. Figure 6.25 shows a portion of the search form that pops up. Enter B% as the category and click the Search button. The Search button is probably hidden way to the right and you have to scroll to see it. Later, you will want to reduce the size of the Category box to try and make this form smaller. Anyway, the search narrows the selection list to just the categories that begin with the letter “B.” For lists of value that contain a large number of entries, this approach is much better than trying to cram the entire list into one display. This approach reduces the amount of data to download on the page, and it provides a smaller list that is easier to read and search. But, it requires more steps, and might require training the users to understand how it works. One recommendation is to use the combo box/list of values approach whenever the list of items exceeds 100 rows.



Activity: Create Main Forms and Subforms

Now that you understand the main forms and grid forms, it is time to combine them into a main form and subform. Remember where this process began: with business forms—particularly the Sale form. A typical business sale form has data for the sale (SaleID, SaleDate) and customer (name, address, and so on). It also has a section of repeating data to hold the specific items being purchased by the customer. This repeating section was split into the SaleItem table, with some elements placed in the Inven-

Action

- Add the table objects to the project: Sale, SaleItem, Inventory.
- Do NOT create views for them. You can add all tables for use later.
- Right-click AP06.model, New Association: Sale + Customer.
- Choose Customer/CustomerID as the source (left) and Sale/CustomerID as the destination (right).
- Click the Add button to create the link.
- Accept defaults except check the box to add to the Application Module.

tory and ItemModel tables. The purpose of the main and subform is to recombine these tables. Keep in mind that each form can be associated directly with only one table. In this case, the Sale form will be based on the Sale table, and the subform will be based on the SaleItem table. Additional data from the other tables can be displayed on the forms, but only the primary keys from those two tables will be used.

Figure 6.26 shows the two parts of the basic Sales form. Creating this form takes many steps—most are similar to those used earlier, but a few new twists are being added to this description. First, the data objects for the main form and subform must be created separately. Notice that each section uses data from multiple tables. The main form is based on the Sale table, but it includes some data from the Customer table. Creating the data for the main form (and the subform) requires building a View object based on a query. Second, these view objects must be linked through an association—which is based on the foreign key (SaleItem.SaleID connects to Sale.SaleID). This link restricts the rows displayed in the subform to those that match the SaleID in the main form. Third, the subform contains a computed expression (LineTotal) to obtain price times quantity. Fourth, the main form displays the total value of the subform, computed as the sum of the LineTotal. Fourth, the main form contains a couple lists of values (for selecting the customer and setting PaymentMethod). To illustrate the process, these are going to be added after the first-pass form has been created. With all of these complications, it is not going to be possible to display all of the screen shots. Instead, the text outlines the steps—you can check the PowerPoint slides for this chapter for additional screen shots if you need more details.

The basic process involves seven major steps:

1. Create a data view joining the Sale and Customer tables.
2. Create a data view joining the SaleItem and Inventory tables.
3. Create a link view to establish a join between these two views.
4. Build the initial form to test it.
5. Add lists of values to the main form.
6. Clean up the form display.
7. Add a calculation to display the Sale Total.

Figure 6.26

Sku	Sale Price	Quantitysold	Modelid	Itemsize	Line Total
600017	\$32.00	1	FCU-154	1200	\$32.00
600046	\$15.00	1	OUU-35	1300	\$15.00
800115	\$425.00	1	05B-498	150	\$425.00
800126	\$352.00	1	ZPR-23	240	\$352.00

View objects are needed instead of tables because both forms contain multiple tables that need to be connected. But, the first step is to add the underlying tables to the project so they can be referenced in the joins. Use the standard right-click menu on Model, New: Business Tier, ADF Business Components, Business Components from Tables. Choose the new tables: Sale, SaleItem, Inventory, and ItemModel. Do not create views for the tables. Skip the updatable and read-only options. In a major project, you would probably just add all of the tables at the start—knowing you are going to use them later. But, adding dozens of tables and views makes the project cluttered and harder to focus on the tables needed at the moment.

Because the Customer table was added before the Sale table, JDeveloper probably did not create the foreign key association between Customer and Sale; so you have to create it by hand. Look at the list of view objects and you should see FK...associations for the Sale/SaleItem table but not for Customer/Sale. Right-click AP06.model to build a New Association. Name it SaleCustomerAssociation. In the left panel (Source), expand the Customer table and select the CustomerID. In the right panel (Destination), expand the Sale table and choose the CustomerID as the matching value.

Now you can create the view that combines columns from the Sale and Customer tables. Right-click AP06.model, New View Object. Name it SaleCustomerView and accept the default option: Updatable access through entity objects. Choose the Sale and Customer tables, leaving the default options. Figure 6.27 shows that

Action

Right-click AP06.model and choose New View Object.

Name: SaleCustomerView.

Defaults: Updatable access through entity objects.

Select Sale and Customer tables, Sale will be updatable, Customer will not.

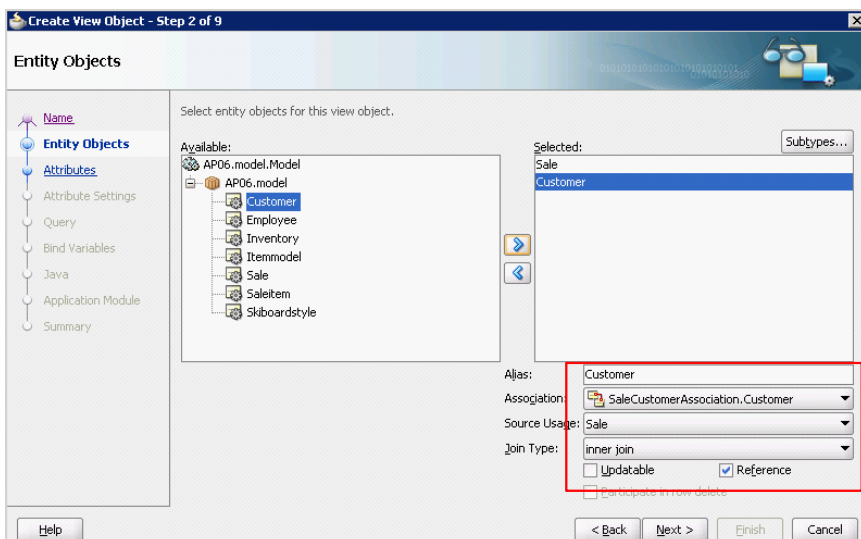
Verify the link between Sale and Customer.

Select all columns from the Sale table and LastName, FirstName, Phone, Email from Customer.

Add Order By Sale.SaleID.

Build a second view for SaleItem and Inventory using all columns from SaleItem and ItemSize and ModelID from Inventory.

Figure 6.27



the Customer table should be set as Reference, not Updatable, and the association that links them should be selected in the drop-down list. If the association is not available, stop and go back a step to build that link.

Select the attributes to be displayed on the main form. Start by selecting all of the columns from the Sale table. You can use the arrow keys to rearrange the order. Then add the FirstName, LastName, Phone, and Email columns from the Customer table.

The wizard will automatically include the Customer.CustomerID column. Bring it along for now—it can be removed from the display form later. Advance to the query shown in Figure 6.28 that is built to connect the Sale and Customer tables. Notice that it uses the old syntax to join the two tables instead of the new standard INNER JOIN. Add Sale.SaleID to the Order By clause—or use Sale.SaleDate if it will be easier to scroll through the form by date. Ignore the Bind Variables and Java options and accept the default options to finish.

Repeat the process for the SaleItem and Inventory tables. Choose the SaleItem table first and the foreign key association should be available in the drop-down list. Choose all columns from the SaleItem table along with ModelID and Item-Size from the Inventory table. It would also be possible to add a third table (Item-Model) to bring along the description and other attributes—but it will be difficult to get them to fit on the display page, so stick with just these few columns for now.

On the attribute summary page, click the New button to create a new computed attributed. Name it: LineTotal with a Type of BigDecimal and Value Type of Expression. For the expression value, enter `Quantitysold * Saleprice`; which should not be updatable (by the user). Click the Edit button and move Quantitysold and Saleprice to the right panel to indicate the total should be recomputed when ei-

Action

In the SaleItemInventory View, at Attribute summary, click the Add button to create a new transient attribute.

Name: LineTotal

Type: BigDecimal

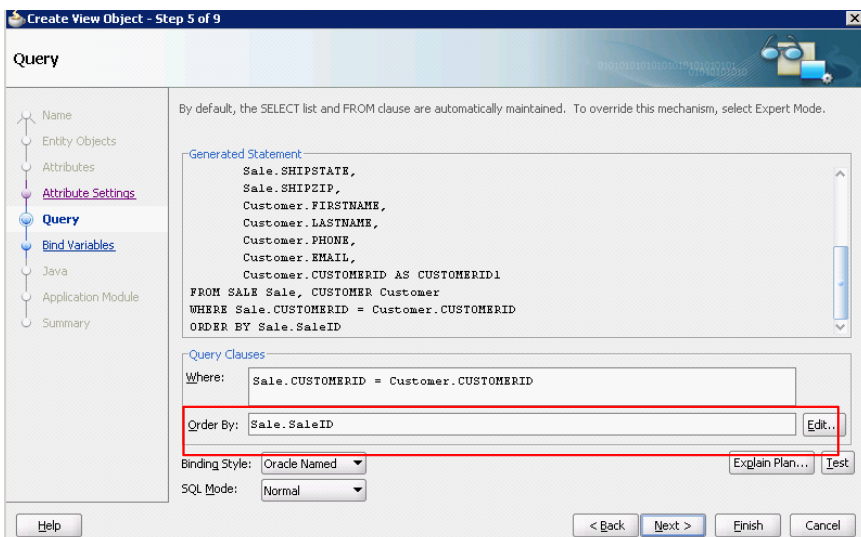
Value Type: Expression

Value: `Quantitysold*Saleprice`

Click the Edit button.

Select Quantitysold and Saleprice to trigger recalculations.

Figure 6.28



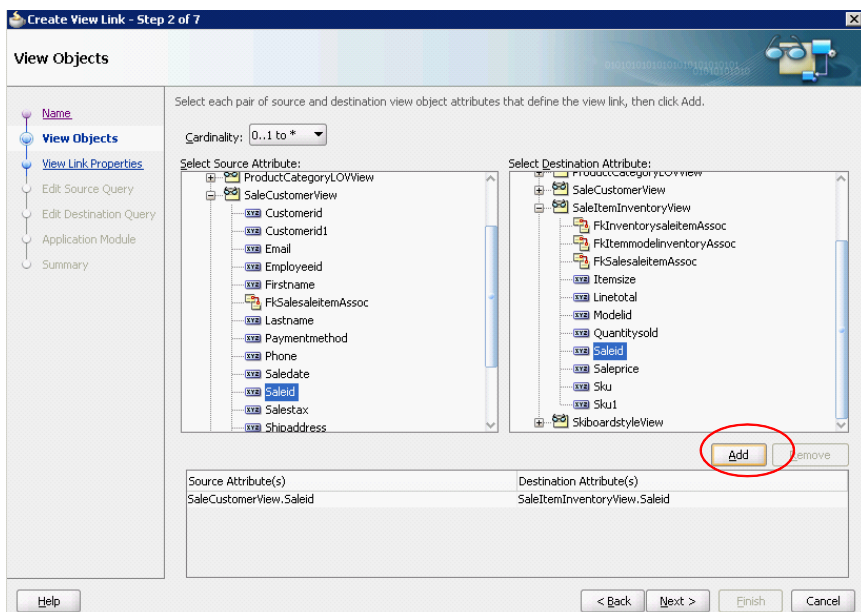
ther of these values changes. Close the edit window and finish the View wizard. Double-check the query and add a sort clause such as Saleitem.SKU. Accept the defaults to finish the wizard.

The next step (#3) is to create an association that links these two views together. Right-click AP06.model and choose New View Link. Name it: SaleToSaleItemViewLink. In the left panel (Source), choose the SaleCustomerView created for the main form and pick the Saleid attribute. In the right panel (Destination), expand the SaleItemInventoryView and choose its Saleid attribute to match the source value. Figure 6.29 indicates that you need to remember to click the Add button to actually create the link—which is then displayed in the list at the bottom of the wizard.

With the two view objects created and connected through a link view, it is now possible to create a first-pass version of the form to test the design (Step #4). The first step is to create a new blank form. In the Application Navigator, expand the ViewController project, right-click the Web Content entry and choose New. Expand the Web Tier and pick the JSF entry, followed by the JSF Page type. Name it Sale.jspx and use the standard single-column template used earlier. Add a title at the top (Sale), assign it the Heading 1 style and drag the Powder.css style onto the page—so this new page matches the style of the other pages. So far, this process matches the steps for creating any form.

Figure 6.30 shows the real trick to creating the main form/subform. Refresh the Data Controls. Scroll through the list to find SaleCustomerView2. Expand it to ensure that SaleItemInventoryView2 lies beneath it. These views with the hierarchical relationship were created when the link between them was established. They

Figure 6.29



Action

Right-click AP06.model, New View Link.
 Name: SaleToSaleItemViewLink
 Source: SaleCustomerView.Saleid
 Destination: SaleItemInventoryView.Saleid
 Click the Add button.
 Accept the defaults except set the Application Module.

are different from the standalone versions of the original views because only this set is linked. Drag the SaleCustomerView2 object onto the form and choose the Form/ADF Form option. Check the options to create the navigation and submit buttons.

After the main form controls are created, drag the nested SaleItemInventoryView2 onto the form beneath the controls. Choose the Table/ADF Table option. Check the options for Row Selection and Sorting. Do not select the Filtering option—unless you anticipate the Sale form to hold dozens of lines that will need to be searched. The goal is to keep the form as simple as possible, and filtering adds options that need to be explained to users.

Figure 6.31 shows the basic form. Observe that the layout is going to need work. For example, the detail subform is way too narrow. However, for now, check that the subform only displays rows that match the SaleID in the main form. Also check that the customer information is updated as you scroll through other sales.

One critical problem exists with all forms created by the wizard: The Submit button does not actually do what you think it does. Basically, the Submit button simply calls the server and refreshes the page. Any data changes are stored only in the data objects created in the application—they are not automatically written to the database tables. To actually store the data, you need to either add a separate Commit button, or add the Commit functionality to the Submit button. In terms of

Action

ViewController, Web Content, New page.
Web Tier/JSF/JSF Page.

Name: Sale.jsp, single-column template.
Add title, set as Heading 1, drag Powder.css onto page.

Refresh the Data Controls.

Find SaleCustomerView2 and expand it.

Verify that SaleItemInventoryView2 is beneath it.

Drag SaleCustomerView2 onto the form:
Form/ADF Form.

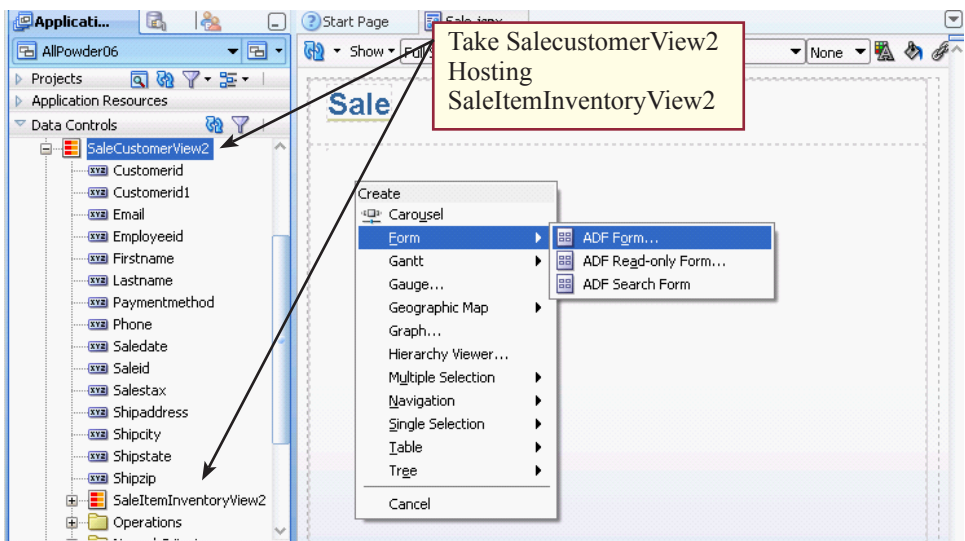
Select Navigation and Submit buttons.

Drag nested SaleItemInventoryView2 onto the bottom of the form: Table/ADF Table.

Check the Row Selection and Sorting boxes (not filtering).

Run the form to test it.

Figure 6.30



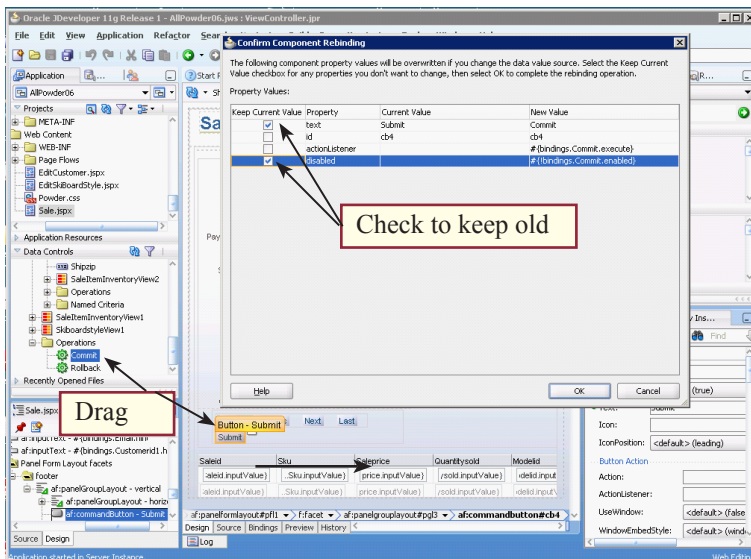
Saleid	Sku	Saleprice
1002	600017	3
1002	600046	1
1002	800115	42
1002	800126	35

Figure 6.31

usability, most application should just add the Commit functionality to the Submit button.

Figure 6.32 shows the basic process. In Data Controls, find the SaleCustomerView2 entry and scroll down to expand the Operations node. Drag the Commit function and drop it onto the Submit button. The key is to ensure that you drag the Commit operation from the correct location—underneath the SaleCustomerView2 node. Check the two boxes indicated to keep the original values: Name=Submit and Disabled=blank (or false which means the button is enabled and can be used). The Submit button will now function properly and changes made on the form will be saved to the database.

Figure 6.32



The next step (#5) is to add the lists of values to select the Customer and the PaymentMethod. The process of creating the LOVs is similar to that used before. Create new View Objects to retrieve the desired data. Edit the SaleCustomerView object to add these new accessors and assign them as LOV objects to the matching attributes. The big change is that in the earlier forms, the LOVs were built before the view was dragged onto the form, so the form build knew which type of control to create. In this new case, the form has already been built. There is no easy way to edit the existing text boxes on the form, so the trick is to drag a new control onto the form, set it as an LOV control and then delete the original text box.

By now, you should be able to create an LOV on your own. However, the CustomerLOVView adds one trick. The LOV needs to display the person's name (first and last) and transfer the CustomerID into the Sale table. The trick is that the display value needs to be a single column, so you have to create a new column (Fullname) concatenated from the other columns. Start the LOV as a New View Object using read-only SQL and the Customer table. When you get to the query screen shown in Figure 6.33, enter a query that retrieves the CustomerID and concatenates the three columns into a single value called Fullname:

```
SELECT CustomerID, Lastname || ' , ' || Firstname || ' ' || Phone AS Fullname
FROM Customer
ORDER BY Fullname
```

As you finish the wizard be sure you set the Key Attribute for the CustomerID column and assign the result to the default Application Module. On the List UI Hints tab, for the List Type, choose Comobo Box with List of Values, and set Fullname in the Display panel.

The second step in creating an LOV is to open the underlying SaleCustomerView object so that you can assign the LOV to the CustomerID column. First

Action

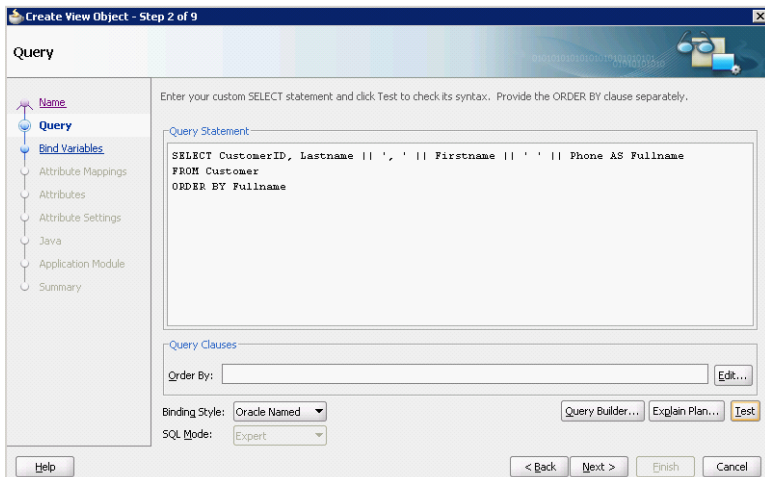
Close the browser running the Sale form.

In Data Controls, under the SaleCustomerView2 entry, expand the Operations node.

Drag the Commit function and drop it onto the Submit button.

Check the boxes to keep the original values for the name (Submit) and disabled (empty).

Figure 6.33



select the View Accessors tab and add the CustomerLOVView to this view object. Switch to the Attribute tab and select the CustomerID column. Be sure to select the CustomerID from the Sale table and not from the Customer table. Scroll to the LOV section and add a new LOV. Select the new LOV as the list data source and choose CustomerID as the list attribute. These steps create the LOV and assign it to the CustomerID in the Sale table. The LOV is not yet installed on the page, but you can do that step after you finish the second LOV.

Action

Create a New View Object.

Name: CustomerLOVView.

Read-only access through SQL Query.

Query includes CustomerID and a LastName+FirstName+Phone display column.

List UI Hints: Combo Box with List of Values.

Edit SaleCustomerView.

Add new LOV as a view accessor.

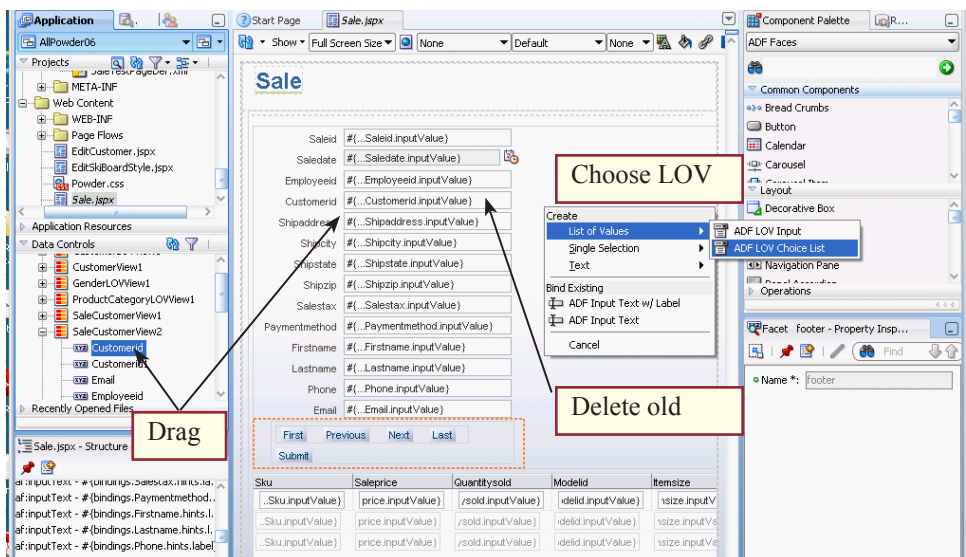
Link to Customerid in the Sale table.

Create a PaymentMethodLOV using the default Choice List.

Repeat the basic process to create the PaymentMethod LOV. Use a read-only SQL query, and select the single column. Set the UI Hint to the default Choice List and pick the PaymentMethod as the display column. Open the SaleCustomerView again and add the new LOV to the list of View Accessors. Switch to the Attribute list, select PaymentMethod and choose the new PaymentMethodLOVView, again choosing its only column.

Finally you can put the LOVs on the Sale form. Close the view object windows if they are open and open the Sale.aspx page in design mode. Refresh the Data Controls and expand the SaleCustomerView2 node. Delete the existing text controls for CustomerID and PaymentMethod. Figure 6.34 shows the next step. Drag the CustomerID data onto the Sale form and drop it carefully between two existing text controls—look for the horizontal dividing line as a target—do not drop it onto an existing text control. Also, be careful to drag the CustomerID column and not CustomerID1 (which is in from the Customer table). In the pop-up menu, choose

Figure 6.34



Once the table is wider, delete the SaleID column—now that you know the subform works properly, there is no reason to display the SaleID on each row. Next, set the width of each column to 70 so they fit on the screen. Then delete the SKU1 column at the end.

It would also be nice to format the SalePrice and LineTotal columns as currency. The easiest approach is to edit the underlying SaleItemInventoryView object. In the Attributes tab, select the SalePrice column and choose the Control Hints tab. Set the Label text to Sale Price and the Format type to Currency. Do the same thing for the LineTotal column. Save and close the view. The changes will be picked up automatically when the Sale page runs.

If you work with the form for a couple of minutes, you will see that you can scroll through the existing orders, but there is no way to add new ones. The solution is to add a button to the main form that enables the user to create a new sale. Fortunately, this function is one of the preprogrammed options within the Data Controls. Expand the SaleCustomerView1 node and scroll down to find and expand the Operations node. Drag the CreateInsert function and drop it onto the main form next to the Submit button. Change its text property to New Sale. Because of the panel (HTML) layout method, the new button actually appears below the Submit button. You might want to add a horizontal panel grouping to place the two buttons side-by-side. The easiest way to add this grouping is to edit the source code. Select the new button and click the Source tab at the bottom of the form. Scroll until you see the highlighted button text.

Figure 6.35 shows the tag and end-tag lines that need to be added to create the grouping. Immediately before the Submit button tag, enter the `<af:panelGroupLayout layout="horizontal" id="pgl4">` tag. On the line following the New Sale button, add the closing tag.

This new button creates a blank Sale entry. Ultimately, you will need to add a sequence to the database so that a new SaleID is created automatically. That process is explained in Chapter 7. More importantly, you cannot use the same process to add new rows to the SaleItem table—so there is no easy way to add new

Action

- Open the Sale.aspx page in design mode.
- Refresh the Data Controls.
- Expand the SaleCustomerView2 node.
- Delete the CustomerID text control.
- Drag Customerid (not Customerid1) and drop it between two text controls.
- Choose List of Values/ADF LOV Choice List.
- Repeat the steps for PaymentMethod.

Action

- Clean up the subform.
- Set Table width to 500.
- Delete SaleID column (in subform).
- Reduce all column widths to 70.
- Delete the SKU1 column (at end).
- Go back to SaleItemInventoryView and set formats for SalePrice and LineTotal.

Figure 6.35

```
<af:panelGroupLayout layout="horizontal" id="pgl4">
  <af:commandButton text="Submit" id="cb4"
    actionListener="#{bindings.Commit.execute}"/>
  <af:commandButton actionListener="#{bindings.CreateInsert.execute}"
    text="New Sale" disabled="#{!bindings.CreateInsert.enabled}"
    id="cb6"/>
</af:panelGroupLayout>
```


items to be sold. You could add a link that opens a new form just to add a new row to the SaleItem table. Another option is to add a text control and button at the bottom of the form. The user would enter a new SKU and click the Add button to insert it into the SaleItem table. However, this process requires additional coding. So, for now, the form is best used for examining and editing existing sales.

Finally, in Step #7, the form needs to compute the total value of the items sold and display that value on the main form. The process is straightforward, but is computed through a somewhat unexpected process. The subtotal is actually computed in the underlying SaleCustomerView—not on the form itself.

Open the SaleCustomerView object and select the Attributes tab. Click the plus sign to add a new transient attribute. Name it: SaleTotal with a data type of BigDecimal. Select the Expression option and carefully enter: SaleItemInventoryView.sum(“LineTotal”)

The expression uses a function form “Groovy” which is an underlying language used in ADF for expressions and calculations that need to interact with the form. The first part (SaleItemInventoryView) is the view accessor that was created when the link was defined between the SaleCustomer and SaleItemInventory views. If you need to create a similar expression in the future, go back and check the name from the Link View. The sum function is one of the basic aggregator functions, and the column total must be enclosed in double quotes. That is the hard part. Everything else is easy. Use the Control Hints to assign the label: Sale Total and Format Type Currency. Save and close the edit window.

The last step is to place the new control on the page. Return to the Sale page and refresh the Data Controls. Drag the new SaleTotal entry onto the top part of the Sale form and choose the Text/ADF Output Formatted w/Label option. Save everything and run the form. You should now see the subtotal for the items displayed in the subform.

Action

In Data Controls expand the SaleCustomerView1 node.

Scroll down and expand the Operations node.

Drag the CreateInsert function and drop it next to the Submit button on the main form.

Select the Button option.

Edit the Source page to add a horizontal layout.

Action

Open the underlying SaleCustomerView.

In the Attributes tab click the plus sign.

Name: SaleTotal

Type: BigDecimal

Expression value:

SaleItemInventoryView.sum(“LineTotal”)

Control Hints: Currency

Dependencies: Saleid

Sale.jspx, drag-and-drop SaleTotal as Text/ADF Output Formatted w/Label

All Powder Basic Reports



Activity: Create Interactive Reports with Subtotals

Most managers want reports so they can evaluate the progress of the business. Increasingly, there is little difference between forms and reports. Originally, reports were designed to be printed and report writers had detailed layout controls for positioning text on the page along with page breaks, headers, and footers. With a Web-based system, reports can be more interactive—providing managers the ability to search for data and drill down to see details. Ultimately, business in-

telligence tools might be more useful to managers than traditional reports. These tools are examined in Chapter 9. Still, it is useful to understand the basic structure of reports.

The challenge with Oracle 11g is that there is no single technology for creating traditional reports. The ADF system examined in the start of this chapter has some tools—which will be used to show how to build interactive data analysis. The Application Express (APEX) system has some more traditional features for building reports, but it requires a separate installation. If you truly need traditional

reports, it is probably best to use the Reports system in the 10g package. Again, that system requires a separate installation (along with management and cost). Another option is Microsoft's Web-based Reporting Systems (see the Microsoft Workbook). Because it is Web based, you could easily include links on your ADF forms to open a browser window that calls that system. One option that some bloggers have suggested is Jasper—a open-source, Java-based reporting system available free from Source Forge: <http://jasperforge.org/projects/jasperreports>.

The goal of this Workbook is to focus on ADF, so the example in this section uses the existing tools to build a basic Customer Sales report. The nice thing is that you will use the same tools to build the report as you do for forms, so the process should be straightforward by now.

Action

- Create a View object
CustomerSalesReportMainView that retrieves basic Customer data.
- Create a View object
SalesTotalsReportView that computes the total value for each Sale.
- Create a Link View object to join the two new views.
- Add a subtotal column to the Sales Main view.
- Create a new JSF page that uses the Master/Detail tools to add two tables.
- Clean up the report.
- Run it and save it.

Figure 6.36

The screenshot shows a web browser window with the URL <http://127.0.0.1:7101/AllPowder06-ViewController-context-root/faces/CustomSalesR...>. The page title is "Customer Sales".

Customer

Customerid	Lastname	Firstname	Email	State	Gender	Sale Total
1307	Lyons	Chester	LyonsC45@msn.c...	CA	Male	\$3,569.00
1439	Hines	Arlene	HinesA230@msn.c...	GA	Female	\$2,815.00
1274	Dixon	Carol	DixonC804@msn.c...	NJ	Female	\$2,789.00
983	Gillespie	Audrey	GillespieA85@msn...	NY	Female	\$2,703.00
155	O'Connor	Carlos	O'ConnorCS24@m...	AL	Male	\$2,674.00
1864	Ford	Manuel	FordM554@msn.com	LA	Male	\$2,661.00
1583	Nash	Joseph	NashJ726@msn.com	NY	Male	\$2,600.00
1057	Rush	Bonita	RushB580@msn.com	IN	Female	\$2,485.00
998	Warden	Jewell	WardenJ602@msn...	AL	Female	\$2,406.00
1891	Turner	Guy	TurnerG252@msn...	MI	Male	\$2,358.00
227	Harvey	Simon	HarveyS524@msn...	IN	Male	\$2,314.00

Sales

Saleid	Saledate	Saletotal	Salecount
1114	1/8/2010	1614	4
1436	12/8/2010	1483	5
1435	12/28/2010	472	1

The first issue in building a report is to identify the level of detail that will be needed. You can always use queries and expressions to compute subtotals across groups, but you need to ensure that your query retrieves the level of detail desired by the managers. As an example, consider a basic sales report by customer. Managers want to list each customer, followed by the sales placed by that customer. If they also want to include the individual items purchased on each sale, that level of detail is different than if they simply want to see the total value of the sale. For now, assume they need to see only the Customer data (main group) and the value of each Sale (detail group). Figure 6.36 shows an example of the report. Of course, the Sale total needs to use a query to compute the sum of price times quantity by sale.

The report has two main sections, so you need to create a new View object for each section. You will also need to create a View Link to build the connection between the views. Adding a subtotal for each customer is straightforward using the technique shown in the previous section. At that point, the page can be built using the ADF Master/Detail tools, and all you have to do is clean up the display and layout.

Begin by creating a new View object named: CustomerSalesReportMainView. Define it as Read-only access through SQL query. Build a query that retrieves basic Customer data, such as CustomerID, Lastname, and State:

```
SELECT
    CUSTOMER.CUSTOMERID CUSTOMERID,
    CUSTOMER.LASTNAME LASTNAME,
    CUSTOMER.FIRSTNAME FIRSTNAME,
    CUSTOMER.EMAIL EMAIL,
    CUSTOMER.STATE STATE,
    CUSTOMER.GENDER GENDER
FROM
    CUSTOMER
ORDER BY CUSTOMER.LASTNAME, CUSTOMER.FIRSTNAME
```

Accept most of the wizard's default values, except assign the Key Attribute to the CustomerID column, and select the default Application Module near the end of the wizard.

Create the second view for the detail object named: SalesTotalsReportView. Again, build it as a read-only SQL query. This query is more complex because it needs to return the SaleDate, CustomerID, and the total of price times quantity for each matching row in the SaleItem table:

```
SELECT
    SALE.SALEID SALEID,
    SALE.SALEDATE SALEDATE,
    SALE.CUSTOMERID CUSTOMERID,
    SUM(SALEITEM.QUANTITYSOLD*SALEITEM.SALEPRICE)
SaleTotal,
    COUNT(SALE.SALEID) SaleCount
FROM
    SALE INNER JOIN SALEITEM
        ON SALE.SALEID=SALEITEM.SALEID
GROUP BY SALE.SALEID, SALE.SALEDATE, SALE.CUSTOMERID
ORDER BY SALE.SALEDATE
```

Be sure to include the join condition. ADF accepts both the newer INNER JOIN form and the older method with the join condition placed in the WHERE clause. You should probably test both queries in SQL Developer to ensure they return the correct values. Again, accept the wizard's defaults, except set SaleID as the Key Attribute and set the Application Module.

Figure 6.37 shows the main step in creating the new View Link object. Choose the new CustomerSales view as the source its CustomerID to link to CustomerID in the SalesTotals view. Remember to click the Add sign to generate the connection link. Finish the wizard by accepting the defaults, except remember to add it to the Application Module at the end.

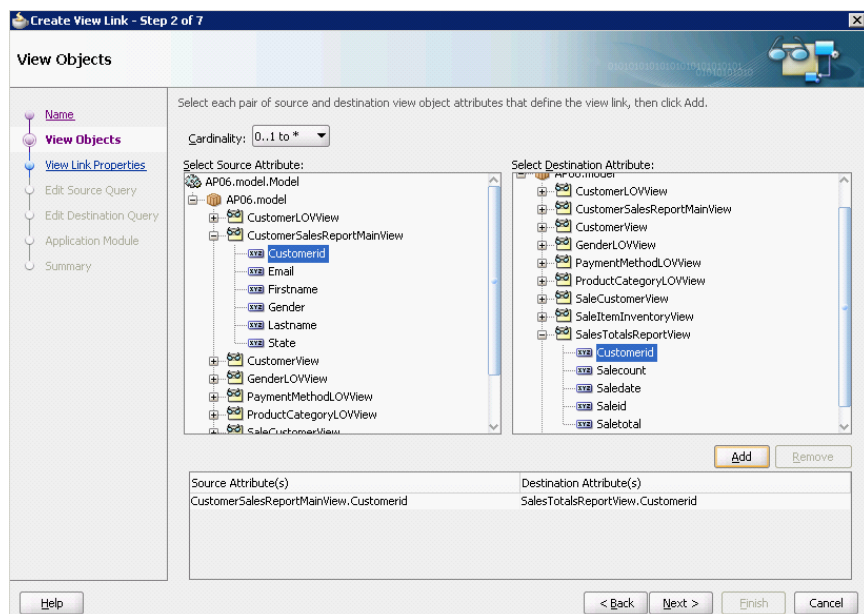
It is important to compute the total of the SalesTotal for each customer and display that value in the report. Managers will probably want to sort the customer list in terms of sales, and then drill down to see the individual sales. Remember the trick to computing subtotals in ADF is to write the expression in the underlying data View. Open the CustomerSalesReportMainView and select the Attributes tab. Click the plus sign to add an attribute. Name it: SalesTotal with a Type of BigDecimal. For the expression value, enter:

```
SalesTotalsReportView.sum("Saletotal")
```

The first part (SalesTotalsReportView object) is determined by the name used in the Link View object that ties the two views together. The Saletotal column must be enclosed in quotes and it is case sensitive. If you re-examine the SalesTotalsReportView, you will see that the column returned is capitalized only on the first letter. After entering the formula, click the Edit button and add CustomerID as a dependency column, then close that edit window. Click the Control Hints tab and enter Sale Total as the label text and set its format to Currency. Save and close the view editor.

With the data objects successfully created and linked, you can now build the JSF display page. Create a new page using the standard one-column layout named:

Figure 6.37



CustomerSalesReport.aspx. Add a “Customer Sales” title at the top and remember to drag the Powder.css style sheet onto the page.

Refresh the Data Controls and expand the CustomerSalesReportMainView2 entry. Figure 6.38 shows the hierarchical relationship that you need to see—which was created by the Link View. Now, the trick is to drag the detail view (SalesTotalsReportView2) object onto the form—not the parent node. Then select the Master Detail/ADF Master Table, Detail Table option to create two tables—one for the Customer data in the master table and one for the Sales totals in the details table. Save and run the report. As you select different rows in the main Customer section, the details Sales section should update. Note that many of the customers do not yet have sales, so the details section will remain blank in several cases.

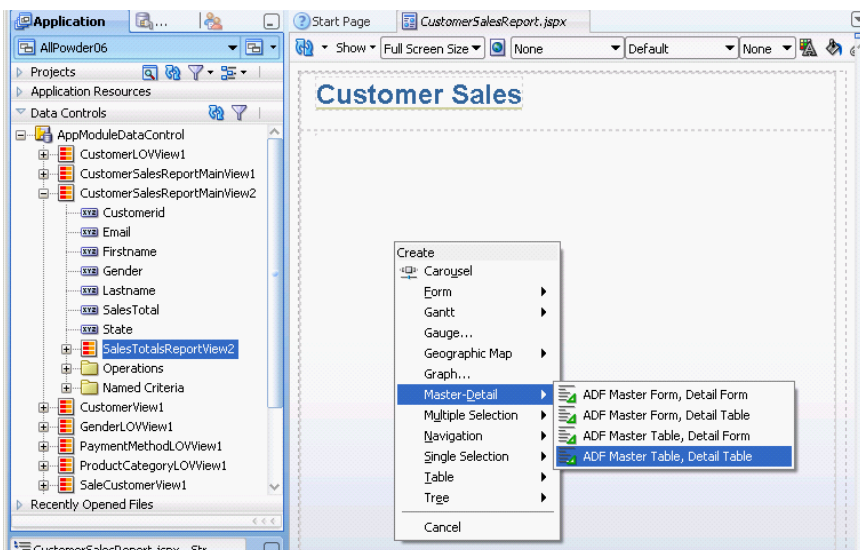
The bigger issue is that the report layout needs work. Both tables are too narrow and the columns need to be resized. Some of the basic fixes include setting the main table width to 550 and the width of the detail table to 500. Reduce most column widths to 70, although a couple can be even smaller. For example, State could be reduced to 30. Remove the CustomerID column from the Detail table. Also reduce the heights of the two tables to save some vertical space.

The subtitles can be changed by editing the properties for the two panelHeader objects. These can be selected using the Structure Navigator at the bottom left corner of JDeveloper, or even by editing the XML source directly. You can also set the columns in the main Customer table to be sortable—simply select each one and change its Sortable property to true. However, note that the sorting is somewhat slow when the report runs.

As you work with redesigning the report re-run it to evaluate the effects of your changes. Report design is often an iterative process as you experiment to see exactly what data will fit on a standard page. You will have to make some assumptions about the screen size and browsers used by the managers, but eventually, your report should resemble Figure 6.36.

Ultimately, the report could probably use search filters. For example, to search for customers or select a range of sale dates. Some of these features can be added to this report design. In other cases, it might be easier to build a different page

Figure 6.38



and manually placing the two tables instead of relying on the master/detail tools. However, you will probably have to edit the PageDef objects to ensure that the main form correctly triggers updates to the details form. You can use the master/detail report design for comparison and even copy XML code from the PageDef files.

You can also add charts with the built-in chart object. However, in the end, avoid trying to cram every possible feature into one report. It is often better to create many simpler reports and tie them together later through hyperlinks. Remember that there is a finite amount of screen space available. Also, more complex interactions, subtotals, crosstabs, and drill downs are easier to create using the business intelligence tools available.

Exercises



Crystal Tigers

The Crystal Tigers club is mostly interested in tracking members and events. The officers who will use the system do not know much about computers, but they can enter data into forms. They are also interested in a few key reports. For instance, they want to be able to get totals for the number of hours members devoted to charity events. They also want monthly summaries of the amount of money raised. The vice president also wants to be able to print a simple listing of the officers, their phone numbers and e-mail addresses. Sometimes, she also wants a similar list for members who have participated in the initial steps of an event. She wants to be able to carry the list with her when the event starts so she knows who to contact if problems arise.

1. Create the basic forms needed to enter data into the database.
2. Build a form similar to the one defined in Chapter 2.
3. Create the main reports needed by the organization.



Capitol Artists

Job tracking is the most important aspect of the application needed by Capitol Artists. In particular, the employees need to be able to quickly select a job and enter the time and expenses for the task performed. This data is then used to create a monthly billing report for the client. Consequently, you need to focus on creating the forms to capture this data. You need to make sure they are fast and easy to use. The managers also want weekly reports showing the hours and money generated by each employee so they can use the data in personnel evaluations.

1. Create the basic forms needed to enter data into the database.
2. Build a form similar to the one defined in Chapter 2.
3. Create the main reports needed by the managers.



Offshore Speed

Special orders have always been a complex problem for the Offshore Speed managers. Customers come to the shop because it is one of the few that can obtain the custom parts they want. But the company has always had problems training employees to collect all of the order data and, keep track of getting the orders placed and delivered in a timely manner. Some of these orders include contracts with

other local firms to perform customization and finish work on the boats. Although these firms do excellent work, most are terrible at keeping records. Consequently, the managers want to use the system to generate reports on individual boats for each contract shop that can be used to remind the other owners of the details. The company also needs reports on the inventory status of the specialized parts. They are having trouble keeping some items in stock, and other items seem to sit on the shelves forever; but they have no good way of keeping track at the moment.

1. Create the basic forms needed to enter data into the database.
2. Build a form similar to the one defined in Chapter 2.
3. Create the main reports needed by the managers.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks.

1. Create the main forms needed for the database, including forms that will be used by administrators.
2. Build the forms similar to the ones used to define the project. That is, build database forms that match the existing user forms.
3. Create the main reports needed. Think about the analysis that managers will want to do and provide reports that help them. Consider adding charts to compare data.

Database Integrity and Transactions

Chapter Outline

Program Code in Oracle, 139

Case: All Powder Board and Ski Shop, 144

Lab Exercise, 144

All Powder Board and Ski Data, 144

Database Cursors, Keys, and Locks, 168

Exercises, 172

Final Project, 174

Objectives

- Define customized functions.
- Improve forms by responding to form events.
- Execute customized SQL statements from code.
- Define transactions.
- Create new rows and use the generated key value.
- Write cursor-based programs that compare data across rows.
- Set up and handle optimistic locking conditions.

Program Code in Oracle

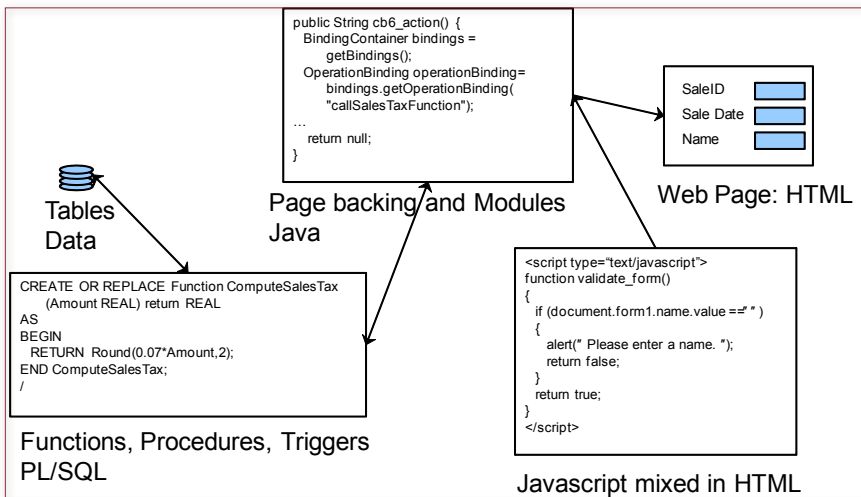
The Oracle DBMS is large, complex, and constantly changing. At this point, you should be able to design and create data tables. You should also know how to write relatively complex SQL queries to retrieve data as well as insert, update, and delete rows of data. The last chapter introduced the basic concepts of building forms and reports. Forms and reports are critical because they are used to create applications that focus on the user's tasks. As a developer, you can never expect users to know anything about the database or SQL. Simple forms and reports can be created using the basic concepts of SQL. However, many forms and applications require more complex interactions between the user and the database. These interactions often have to be programmed with procedural code that evaluates conditions, searches for items, or performs repetitive steps automatically. So, you have to learn how to write program code in the Oracle environment.

An important aspect of the current version of Oracle is that users interact with your application using a Web browser. This approach actually simplifies several difficult problems. For example, the database, application, and code all reside on a central server—making it relatively easy to deploy and upgrade files as well as monitor security access. And the effect is minimal when client computers crash, because users can simply start a browser on a new computer. However, programming Web applications can become complicated.

Figure 7.1 shows the major components involved in writing programming code in Oracle Web applications. First, you can create functions, procedures, and triggers that are stored in the DBMS along with the tables. These functions are written in PL/SQL which consists of SQL statements along with procedural commands such as IF statements and loops. Code that relies heavily on the data and consists mostly of SQL commands should be stored as these functions and procedures. They will be available to any applications that need to access the data.

The second level of programming relies on the Java language and runs as part of the Web server. You can create independent Application Modules, which are code files that reside on the server and are used to perform specific tasks. Generally, business rules and processes are written in Application Modules—often tied

Figure 7.1



into PL/SQL procedures stored in the DBMS. Application Modules can be used and shared by many applications on the same server. Every Web form or report that you create can also have a “backing page,” which consists of a code file that is designed to interact with the objects on the specific page. This code runs on the server and executed either when the page is first built or when some page action (such as a button click) calls back to the server to run a specific function.

The third level of programming exists because of HTML and Web browsers. It consists of Javascript code that is embedded (or downloaded) on the page itself. This code runs on the client computer within the browser. It is used to provide more immediate responses to the user—typically without the need to return to the server. For instance, Javascript is commonly used to perform initial validation of input data on the form. If a user forgets to enter a critical item, the Javascript code can quickly spot the problem and raise a warning message. However, you must always remember that because Javascript runs on the client computer, it can be prevented from running, circumvented, or even altered by users. So any tests made using Javascript must be repeated using either Java or PL/SQL on the server. Then why use Javascript? Basically, to provide more immediate feedback to users. Javascript can be used to provide a richer, more responsive environment to the user, including partial page refresh and drag-and-drop capabilities. Asynchronous Javascript and XML (AJAX) technologies are used to have the browser send requests to the server behind the page, and provide more subtle page changes without the need to rebuild the entire page.

Are you worried yet? To build applications in Oracle, ultimately you need to learn three programming techniques: PL/SQL, Java, and Javascript. And yes, Javascript is different from Java. On top of the three languages, each environment has its own framework, or objects available. PL/SQL uses standard SQL commands, so you should be familiar with those; but you eventually need to learn to use the built-in functions. Java on the Web server typically relies on ADF Faces—a large set of tools for building and interacting with Web pages. Javascript on the browser relies on the document object model (DOM) of a browser page. You need to learn how to identify individual components on a page, such as the document, form, input boxes, and tables.

The languages of Java and Javascript are nice, but they are not explained in this book. Fortunately, you can find several tutorials and tons of examples on the Web. Instead, this chapter focuses on showing you how to handle some common tasks. You should be able to build all of the examples by copying the sample code. Ultimately, in large companies, on big projects, the various tasks are assigned to different people or different groups. A database expert might be in charge of designing the tables and the PL/SQL. An application developer might focus on writing server code in Java, and a page designer might be hired just to make the pages work well for the user. One of the purposes of Oracle’s JDeveloper is to divide the application into these three sets of tasks so that different people can work on their assigned components at the same time with minimal interference with other workers.

The key is to understand when you need to use each type of code. PL/SQL code is used for business processes that need to be centralized and depend heavily on the data. Javascript on the browser is used to improve usability and appearance on specific pages. ADF Java server code is used for two purposes: (1) To control data interaction on individual pages (backing page), and (2) To perform generalized business functions (application module).

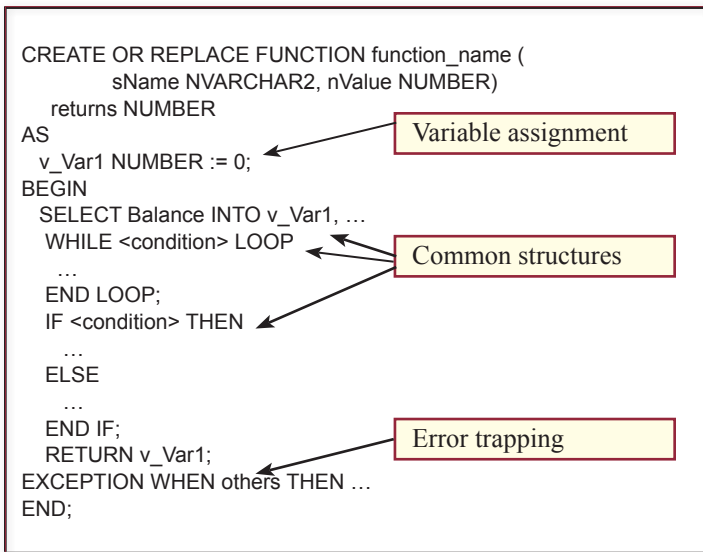
PL/SQL Functions, Procedures, and Triggers

First, the only difference between procedures and functions is that functions return a value and procedures do not. Both can have input parameters and the internal structure and code is very similar. Generally, functions are more useful because even if you do not need to return a result, it is nice to return an indicator to let the calling process know if the code succeeded or failed. So, the examples here focus on functions, but you can use a similar process to create procedures if you really need one.

Second, the difference between functions and triggers is that functions have to be called specifically to run. Triggers consist of code that is run when some database event occurs. For example, triggers can be assigned to run when a data row is inserted into a table, when a specific data item is updated, or when rows are deleted. You write the code and assign it to a specific database event. The challenge with triggers is that they might cascade—a change in one table fires a trigger that alters a second table which fires a new trigger, which alters a third table, and so on. Writing triggers that affect other triggers is risky. Trying to debug cascading triggers is a much bigger problem. Whenever you write a trigger be absolutely certain that you need it and that you document your work so that other people will be able to understand the purpose of the trigger.

Figure 7.2 shows the basic structure of a PL/SQL function. You define a unique name, specify the input parameters and their data types, and identify the data type of the value being returned by the function. Within the function (or procedure or trigger), you can declare local variables. You can perform calculations with these variables, but be sure to use the colon-equals notation (`v_Var1 := 0;`) or the code will not compile. You can also use the `SELECT INTO` statement to retrieve single values from a table and store them into a variable. If you need to retrieve multiple rows from a table, you need to define a cursor variable that will track through the returned dataset row-by-row. To track through rows, you need to create a loop with either a `FOR` statement or a `WHILE` condition. Conditional (`IF`) statements are also commonly used in writing code. Error trapping is handled by adding an

Figure 7.2



EXCEPTION section to your code and then writing WHEN <condition> THEN statements.

The basic format of the Oracle cursor code is:

```
CURSOR myCursor IS
  SELECT columnA, columnB, ... FROM table ...
BEGIN
  FOR oneRow IN myCursor LOOP
    Code: oneRow.columnA ...
  END LOOP;
END;
```

You are always better off using SQL and avoiding cursor loops as much as possible. With the newer LAG and LEAD analytic functions, it is much easier to use SQL even for relatively complex problems.

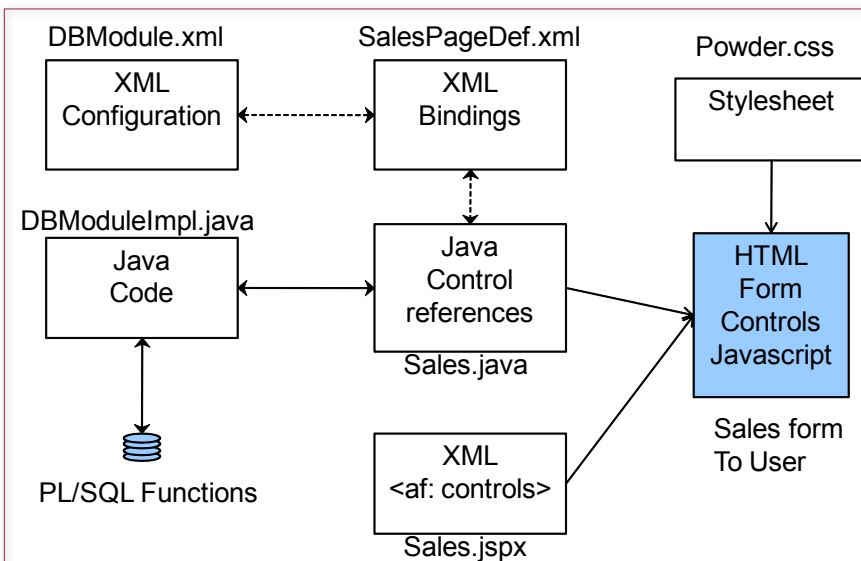
Javascript and the Browser

Javascript is used very little in this chapter—largely because it is a relatively complex topic. Also, you can find a large amount of documentation for standard Javascript problems on the Web. Finally, JDeveloper has some tools to automatically write Javascript code for you to handle common tasks. Specifically, you can create field validators on the form simply by specifying properties and formats or dragging pre-written validators onto the form. Many of the common tasks for Javascript are already handled within JDeveloper.

The key to writing Javascript code is to learn the document object model (DOM). Warning: The DOM is going to change when the next version of HTML (HTML5) is released—probably in late 2010 or early 2011. The main structures should remain, but several new elements are being added.

The main DOM elements are the Window and Document objects. The standard HTML elements reside within the Document—typically in a hierarchical reference. For example, Javascript code might refer to the value in a specific text box as: document.form1.Lastname.value; Warning: Javascript is case-sensitive and it

Figure 7.3



is easy to make typing mistakes. These errors are hard to debug and typically show up only when the page runs and the page debugger generates a hard-to-understand error message.

ADF Faces and Java Server Code

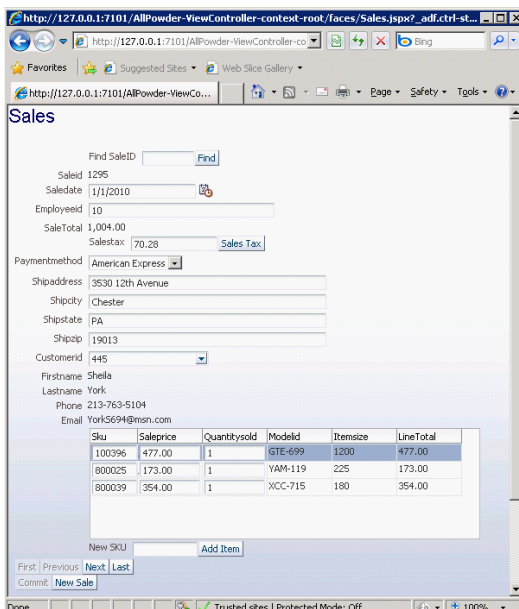
ADF Faces is an Oracle proprietary framework designed to make it easier to build Web pages with Java. Oracle donated an earlier version of the framework to open-source (Java Faces). It is possible to use Java Faces instead of ADF, but you cannot mix the two, and ADF is the default and is more useful with JDeveloper. You used ADF Faces in the last chapter to build the forms and reports.

Java on the server is commonly used in two places: (1) On a backing page to handle data interactions with the specific page, and (2) In separate application modules to automate business procedures. In many cases, you will need code in both places—the application module to retrieve and process data from the database and backing-page code to deliver the results to the page. Figure 7.3 shows the primary files that are needed to integrate code into an application module and a display page. As you will see in the lab sections below, the individual files are created by the JDeveloper wizards. JDeveloper also has some tools to automatically create code and references in the files. However, you sometimes need to edit the files to add your own code and references.

To understand the process, begin with the Sales.jspx page that you have already seen. It is an XML file that contains the descriptions of the page layout and the ADF controls on that page. The Powder.css page contains standard Web style definitions—it is passed directly to the browser. The Web server reads the Sales.jspx page and generates HTML and Javascript code that is sent to the browser.

The Sales.java file is a backing page for the Sales form. Initially, it is largely empty—containing simple references to the controls on the jsp page. You can define events on the browser page that will trigger code to run on the Sales.java page. So creating code requires two steps: (1) Identifying the page event, and (2)

Figure 7.4



Writing the Java code to do what you want. For instance, you could place a button on a form so that when it is clicked (event) your code computes sales tax due based on the total amount of the sale and stores the result in the sales tax control on the form.

The system quickly becomes more complex when you want to write code to access the database or perform other centralized functions. In this case, you need to create a new Application Module—which creates several files. The two most important files are `DBModule.xml` and `DBModuleImpl.java` (assuming the module is named `DBModule`). The java file contains the actual code—including calls to PL/SQL functions to interact with the database. The `DBModule.xml` file contains definitions that enable your code to be accessed from other pages (and modules). To access the module functions from a page, you also need to add definitions in the `SalesPageDef.xml` file. JDeveloper has tools to help you create these definitions in the `DBModule.xml` and `SalesPageDef.xml` files, but it helps if you know how the files interact and how to edit them.

Keep Figure 7.3 as a reference—it will help you remember which files are needed when you write code. Why does Oracle use so many different files? The goal is to separate projects into three major levels: (1) Database, (2) Business components, and (3) Page display. Page display is actually split further into data and layout. These separations make it easier to modify one segment without affecting the others. For example, a graphics designer can redesign the pages without affecting the data content. Keep in mind that other configuration files (XML) are also used to define pages and controls. If you need to delete pages and start over, you often have to manually edit these additional files: `AppModule.xml` for views and `DataBindings.cpx` for pages. However, when you manually edit any of the XML files, you should first make a backup copy in case you accidentally delete critical references.

Case: All Powder Board and Ski Shop

With this background, you are now ready to improve the Sales form. Figure 7.4 shows the basic objective: (1) A button to compute sales tax, (2) A button (at the bottom) to add a new Sale, (3) The ability to add a new item (SKU) purchased, and (4) A simple search that lets the user jump to a specific SaleID. Notice that the form does not contain a button to delete either Sales or SaleItems. These options are straightforward to add, but they carry dangers. Letting anyone delete any sale or any item can lead to accidents or deliberate destruction of data. A more complex form could include delete options that appear only for certain users, along with security controls on the underlying data. These elements can be added later.

Lab Exercise

All Powder Board and Ski Data

To be able to add a new Sale, the system needs to generate SaleID values automatically. You cannot expect users to create unique ID values, so Oracle has a system for creating them. Like everything else in Oracle, it is not completely automatic, so you have to write some PL/SQL code. For the projects in this chapter, you might be able to continue using the base forms you created in Chapter 6. However, it might be easier to start over with a new project. In addition to starting with clean project files, it will be good practice to improve your speed at building base forms.



Activity: Generate and Use Keys

Many tables require generated keys—to ensure that unique ID values are created automatically. Oracle uses sequences to generate unique key values. Sequences are objects that stand alone to generate sequential values. They have some nice properties that make them highly efficient for multiple users and heavy loads. However, the act of generating a new number does not automatically store it in the table. You need to add a trigger to the desired table, which provides a means of automating the number generation when you need it the most—inserting a new row. Figure 7.5 shows the PL/SQL code to generate a sequence of numbers that will be used for the Sale table. Notice that the command contains no overt indication that it is for the Sale table. Only the name that you provide gives a clue. The point is that sequences are technically independent from a table. Your code, either through a trigger or through other INSERT code is what ties a sequence to a table. Notice that several options are available for generating sequences of numbers. This example starts at 10000 to avoid collisions with the existing data. You might have to DROP any existing sequence that was defined by the database design system, since it will start at 1, which is too low. You could also use an ALTER SEQUENCE command to set the MINVALUE if you prefer not to drop the existing sequence.

One of the easiest ways to use a sequence is to automatically generate a new value whenever a row is inserted into the table. Figure 7.6 shows the trigger that will generate sequenced key values for the Sale table. Each time a row is added to the Sale table, this trigger is fired and it generates a unique key value by selecting it off the sequence list. The trigger must be fired before the row is inserted, because you cannot insert a row without a primary key. There are some potential drawbacks to this approach. In particular, what happens if someone wants to insert a new row without using the generated key? This situation arises when you need to import existing data that already has key values that should not be changed. If this situation is relatively rare, or at least controllable, you can always disable the trigger (ALTER TRIGGER GenKeyForSale DISABLE), load the data, and then reenabling the trigger. If there is a regular need to insert rows

Action

Create the new sequence definition for the Sale table.

Create a trigger for the Sale table that generates a new sequence value and uses it for the SaleID.

Test the process by inserting a row into the Sale table without using a SaleID.

Action

Create the new sequence definition for the Sale table.

Create a trigger for the Sale table that generates a new sequence value and uses it for the SaleID.

Test the process by inserting a row into the Sale table without using a SaleID.

Figure 7.5

```
--DROP SEQUENCE sq_Sale;
CREATE SEQUENCE sq_Sale
INCREMENT BY 1
START WITH 10000
NOMAXVALUE
NOCYCLE
CACHE 10;
```

```

CREATE OR REPLACE TRIGGER GenKeyForSale
  BEFORE INSERT ON Sale
  FOR EACH ROW
BEGIN
  SELECT sq_Sale.NEXTVAL INTO :NEW.SaleID FROM dual;
END;
/

```

Figure 7.6

with a known SaleID, the trigger code can be modified to add a conditional statement:

```

IF :NEW.SaleID Is Null OR :New.SaleID < 0 THEN
  SELECT ...
END IF

```

You should test all sequences and triggers using SQL. Figure 7.7 shows the statements needed to test this particular trigger. The first line inserts a row into the Sale table. Notice that it does not specify a value for the SaleID. If you did specify a value, the command would be accepted, but the SaleID value would be discarded. The second line retrieves the value that was generated by the sequence. You will use this value (probably 10000) in the third statement to retrieve the Sale values to ensure the row was created correctly. Note that you can use the sq_Sale.CURRVAL function to obtain the number that was most recently generated. This function will be useful on the Sales form because it will be used to refresh the display page to move to the new Sale.

At this point the sequence has been defined and assigned to the table. Ultimately, you need to create sequences and triggers for all of the tables that require generated keys: Sale, Rental, Customer, Manufacturer, and Employee. You should copy your SQL statements for the sequences and triggers and store them in your lab notebook. You can rerun these scripts if you ever need to rebuild the database.

Action

Define sequences and INSERT triggers for tables: Sale, Customer, Manufacturer, Employee, and Rental.
 Create a new application: AllPowder with package prefix: AP.
 Define the database connection.
 Create entities for all tables, do not create view objects.
 Assign the default Application Module.
 Select the Business Components diagram.

Figure 7.7

```

INSERT INTO Sale (CustomerID, EmployeeID) VALUES (582, 5);
SELECT sq_Sale.CURRVAL FROM dual;
SELECT SaleID, CustomerID FROM Sale WHERE SaleID=10000;

CURRVAL
10000

SELECT SaleID, CustomerID FROM Sale WHERE SaleID=10000;

SALEID CUSTOMERID
10000      582

```


So far all of the work has been done within the database. Now you need to rebuild the JDeveloper project. It is actually easiest to start over from scratch. Create a new project—named AllPowder with a package prefix of AP. Define the database connection. Add all of the tables as new entity objects. Right-click the Model node and choose New, then Business Tier/ADF Business Components: Business Components from Tables. Select all of the tables but not stored views. For now, do not create any view objects (either updatable or read only). Assign the default Application Module and select the Business Components Diagram.

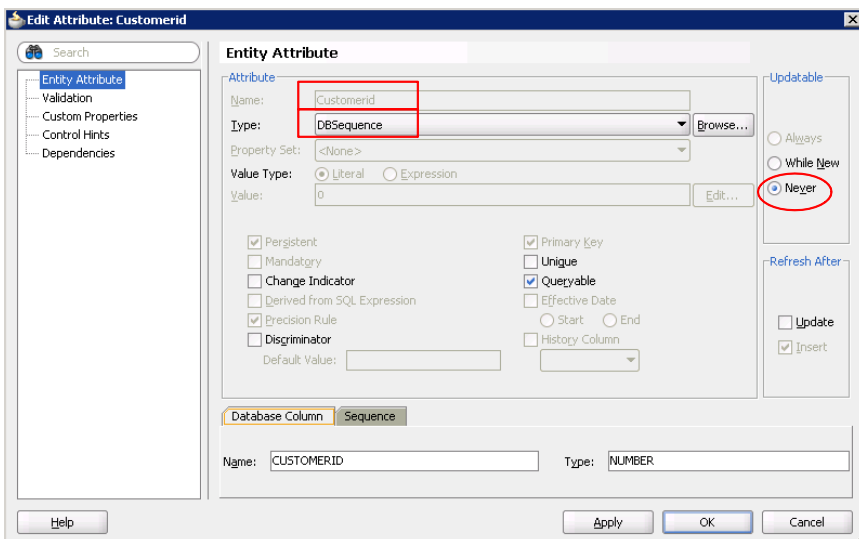
Action

Edit the Customer table object.
Click the Attributes tab.
Edit the CustomerID.
Change its type to: DBSequence.
Set Update to Never.
Re-create the CustomerView.
Re-create the Gender LOV and assign it to the CustomerView.
Re-create the Edit Customer form.
Drag the Commit operation and drop it onto the Submit button.
Add a CreateInsert button to the form.

Before creating forms, you need to tell JDeveloper that some of the tables use sequences. Figure 7.8 shows the basic process for the Customer entity. Open the entity and switch to the Attributes tab. Double-click the CustomerID attribute to edit it. Change the Type to DBSequence and be sure to set the Updates to Never on the right-hand side. If you look at the Sequences tab near the bottom, you can see this form has the ability to define sequences as well. This tab option is only used if you choose the option to have JDeveloper create the underlying tables on the database. You need to repeat the process of setting the DBSequence type for all five of the tables that use sequences.

Now you can use the standard process to build the Customer form. You can review the detailed steps in the previous chapter. Basically, create a View Object in the Model project that uses all of the attributes from the Customer object. You should also re-create the Gender LOV with the static list of choices (Female, Male, and Unidentified). Set the UI Hints and then add the GenderLOVView as an accessor in the CustomerView and assign the Gender column to the LOV.

Figure 7.8



The screenshot shows a web browser window with the URL `http://127.0.0.1:7101/AllPowder...`. The page title is "Edit Customer". The form contains the following fields and values:

- Customerid: -2 (highlighted with a red box)
- Lastname: Test
- Firstname: Beta
- Phone: 111-2222
- Email: test@test.com
- Address: (empty)
- City: (empty)
- State: (empty)
- Zip: (empty)
- Gender: Male (dropdown menu)
- Dateofbirth: (empty)

At the bottom of the form, there are navigation buttons: "First", "Previous", "Next", and "Last". Below these are action buttons: "Submit", "New Customer", and "Commit".

Figure 7.9

Finally, you can create a new JSF Page in the ViewController project. Drag the new CustomerView onto the page as a Form/ADF Form. Use an OutputText object to hold the title and add or create the style sheet Powder.css. Fix the Submit button by dragging the Commit operation from the data control and dropping it onto the Submit button. Add a CreateInsert button to the form so you can use it to add new customers.

Figure 7.9 shows the Customer form after clicking the New Customer (insert) button. Notice that the Web server creates a temporary CustomerID that is negative. Remember that the server holds a temporary set of data in the entity objects. When this data is submitted (committed) to the database, the SQL trigger will automatically create a new CustomerID value from the sequence.

Eventually, you should add a search form to the top of the page and bind the main retrieval query to the search conditions. Not only will it make it easier for users to find specific customers, but the retrieval query is more efficient—pulling only a few selected rows from the database at one time. JDeveloper has two query tools that you can use, but a few more coding items have to be covered first.



Activity: Create Sales Tax Function

Figure 7.10 shows the Sale form developed in the last chapter. Notice that it has a box to enter the sales tax. If you look at the underlying Sale table, you will see that it contains a column to hold the sales tax amount for each sale.

You could argue that the sales tax does not have to be stored, since it can always be computed from the other sales data. But what happens if the tax rate changes? Or, what if the round-off computation is modified? Then the company's sales tax records will no longer exactly match the data filed with the state and local governments. It is safer to store the actual tax amount collected to ensure consistency. However, now you need a method to compute the sales tax on each sale; you certainly cannot expect clerks to compute the amount, or even look it up correctly

Action

- Use SQL to create the ComputeSalesTax function within a new Taxes package.
- Test the function with an SQL statement.

Sale

* Saleid

Saledate

Employeeid

Customerid

Paymentmethod

Salestax Sales Tax

Sale Total \$824.00

Shipaddress

Shipcity

Shipstate

Shipzip

Firstname

Lastname

Phone

Email

First Previous Next Last

Submit New Sale

SKU	Sale Price	Quantitysold	Modelid	Itemsize	Line Total
600017	\$32.00	1	FCU-154	1200	\$32.00
600046	\$15.00	1	OOU-35	1300	\$15.00
800115	\$425.00	1	DSB-498	150	\$425.00
800126	\$352.00	1	ZPR-23	240	\$352.00

Done Trusted sites Protected Mode: Off

Figure 7.10

in a table. Instead, you need to write a function that will compute the sales tax correctly and transfer it to the form and the database. Sales taxes can be highly complex. Some items might be taxable, while others are not. Since each state and local district is different (and there are several thousand tax districts in the United States alone), this presentation is simplified and assumes a single tax rate that is applied to all sales and to rental items.

The first question you must answer when creating custom code is to determine where it belongs: (1) In the database as PL/SQL, (2) In an application module as Java, or (3) On the Sale page as Javascript. Putting computations on individual pages is usually a bad idea—it makes them hard to find and change later, and they can be used only on that page. Options (1) and (2) are better and roughly similar; however, option (1) writing the code as a database function is probably the best answer in this case. Eventually, the tax computation will need more complex features that will rely on information in the data tables, such as year, geographic

Figure 7.11

```

CREATE OR REPLACE PACKAGE Taxes AS
  Function ComputeSalesTax(Amount REAL) return REAL;
END Taxes;
/
CREATE PACKAGE BODY Taxes AS
  FUNCTION ComputeSalesTax(Amount REAL) return REAL IS
    taxRate REAL := 0.07;
  BEGIN
    RETURN (Round(taxRate*Amount,2));
  END ComputeSalesTax;
END Taxes;
/

```

```
SELECT Taxes.ComputeSalesTax(500) FROM dual;
TAXES.COMPUTESALETAX(500)
35
```

Figure 7.12

region, and type of product. Placing the code in a database package will make it available to any form, query, or report within the application.

In this case, you will create a package named Taxes that will eventually hold other tax-related procedures. When the auditor asks to see all of the tax-related calculations, you can quickly find them in this one package. Figure 7.11 shows the PL/SQL code used to create the package header and the package body. If you run the two commands together, make certain they are separated by a slash (/) or it will not accept them. Right now, the package body contains only the simple function to compute sales taxes.

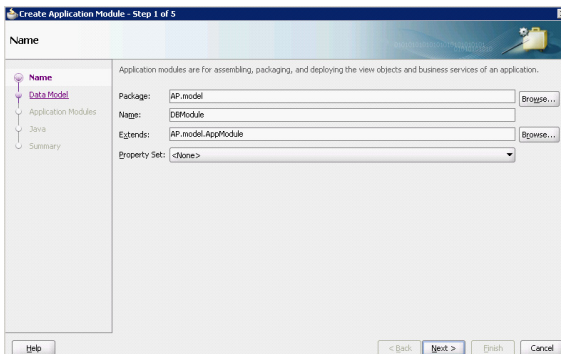
The tax calculation function is deliberately simple to highlight the process instead of the accounting rules. Be sure to use a variable for the tax rate, since it makes the code easier to understand, which reduces errors when someone tries to modify it later. Also, make sure you use the Round function to truncate the tax due at two decimal places. Run the commands to create the package and the function. You can now use this function in queries and forms just as you would use any other function. You can test the function in SQL using the special system table called dual. This tiny table contains one column and one row of data, making it useful for testing functions and calculations because it returns only a single value. Figure 7.12 shows the command and the correct result. Since the function is stored within a package, you include the name of the package when you call the function Taxes.ComputeSalesTax.



Activity: Create an Application Module

Writing the sales tax function in the database makes it accessible to operations within the database. You could use that function in a query that computes the total amount for each sale and then computes the sales tax on that total. This total could then be passed to the page form as simply another attribute. The drawback to this approach is that it re-computes the sales tax every time the page is opened—potentially changing the value of the tax if the tax rate changes. In most cases, you do not want to alter tax computations that were made in prior years. Ultimately, you want the Sales page to decide when to compute the tax—possibly whenever

Figure 7.13



new items are added or by relying on a clerk to click a Tax button.

Enabling a display page to call a function in the database requires adding an intermediate step: Create an Application Module that knows how to connect to the database. You create an application module by right-clicking the Model section in the navigator and choosing the New Application Module option. Figure 7.13 shows the two important steps: (1) Name: DBModule, and (2) Extends AP.model.AppModule. The name could be almost anything and you can create different modules to

Action

Create a new Application Module that extends the existing AppModule.

Check box to Generate Application Class.

Open DBModuleImpl.java and add helper code from Oracle.

Add function: callSalesTaxFunction

Open DBModule.xml, Overview tab, Java tab, Client Interface.

Move new function to Selected side.

Figure 7.14: From Oracle Fusion Documentation

```
// Helper method from Oracle Fusion Documentation Chapter 37.5
// Some constants
protected static int NUMBER = Types.NUMERIC;
protected static int DATE = Types.DATE;
protected static int VARCHAR2 = Types.VARCHAR;

protected Object callStoredFunction(int sqlReturnType, String stmt,
    Object[] bindVars) {
    CallableStatement st = null;
    try {
        // 1. Create a JDBC CallableStatement
        st = getDBTransaction().createCallableStatement(
            "begin ? := "+stmt+";end;",0);
        // 2. Register the first bind variable for the return value
        st.registerOutParameter(1, sqlReturnType);
        if (bindVars != null) {
            // 3. Loop over values for the bind variables passed in, if any
            for (int z = 0; z < bindVars.length; z++) {
                // 4. Set the value of user-supplied bind vars in the stmt
                st.setObject(z + 2, bindVars[z]);
            }
        }
        // 5. Set the value of user-supplied bind vars in the stmt
        st.executeUpdate();
        // 6. Return the value of the first bind variable
        return st.getObject(1);
    }
    catch (SQLException e) {
        throw new JboException(e);
    }
    finally {
        if (st != null) {
            try {
                // 7. Close the statement
                st.close();
            }
            catch (SQLException e) {}
        }
    }
}
```

handle different sets of tasks. Eventually the module will hold several functions or procedures, so related processes should be grouped into a single module with name that reflects their purpose. The second step is critical and not set by default. Notice that the value depends on the specific names chosen within your project. Use the Browse button to select the Application Module that already exists in your project. When the wizard finishes you should have a new DBModule section in the Model project. It should contain four new files: DBModule.xml, DBModuleClient.java, DBModule.java, and DBModuleImpl.java. You need to work with only the first and last of these files.

The main module code belongs in the “implementation” file, so open DBModuleImpl.java to edit it. The first thing you need is a special helper code from the Oracle documentation that makes it relatively painless to call database functions. The code can be found in chapter 37.5 of the Fusion Developer’s Guide for Oracle Application Development Framework, document B31974-05, which is available online. Figure 7.14 shows the callStoredFunction code and you should be able to copy-and-paste the entire set of code into the DBModuleImpl.java file. Place the code immediately above the bottom closing brace “}”. This generic function can be called with the name and parameters of the sales tax function. It reformats the parameters, calls the function in the database, and returns the result to your calling function.

The next step is to write a Java function in the same DBModuleImpl.java file that uses the helper function to call the sales tax computation. Figure 7.15 shows the code—it basically consists of a single long line that calls the helper function. This new function creates an interface that enables Java code on any page to call this function which passes the parameter (value or sales total) to the function in the database, computes the sales tax and returns the value as a string. The call to the helper function includes the type of data being returned, the name of the function with question marks used for parameters, and an object array that contains the values to pass to the function. An output type of String was chosen because the result is ultimately displayed on a page in a text box.

Both sets of java code will require the editor to import various library definitions. The editor should load them automatically, but it might prompt you to use Alt-Enter to select the appropriate library. Ultimately, the libraries needed are:

```
import java.sql.CallableStatement;
import java.sql.SQLException;
import java.sql.Types;
import oracle.jbo.JboException;
import oracle.jbo.server.ApplicationModuleImpl;
```

You can save and close the java file, but another critical step is required before the function can be seen by your page. You need to tell the DBModule to publish its functions so they are available to other code modules. First, notice that the

Figure 7.15

```
public String callSalesTaxFunction(Number value) {
    String result=(String)callStoredFunction(
        VARCHAR2, " Taxes.ComputeSalesTax(?)",
        new Object[]{value});
    return result;
}
```

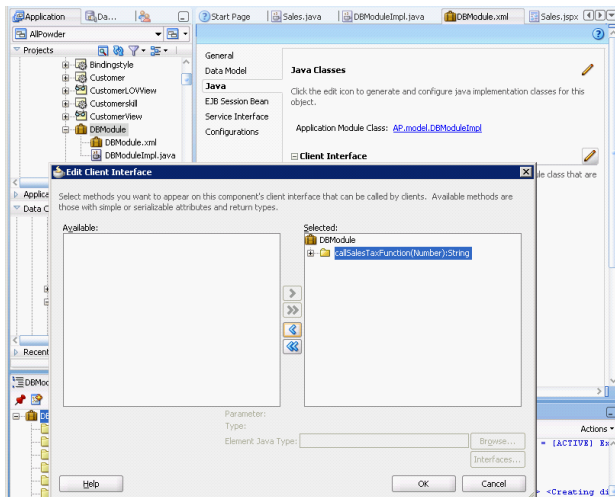


Figure 7.16

salesTax function was declared with the “public” notation. Second, as shown in Figure 7.16, open the DBModule.xml file. Select the Overview tab on the bottom and then the Java tab at the top left of the page. Click the edit button (pencil) for the Client Interface section. Move the new sales tax function to the Selected side. Save and close the XML file. Your function should now be visible to other modules—eventually you might have to refresh the Data Controls to see the options.



Activity: Create a New Sales Page to Compute Sales Tax

You are now ready to create the new Sales page. Technically, it is possible to modify the page you created in the last chapter, but you need to add a backing page to it to handle Java code and that would probably have to be done manually. Starting from scratch, JDeveloper has an option to add all of the components for you. The basic process is to create (1) the Sale + Customer View, (2) the CustomerLOV and PaymentMethodLOV views to use for lookups, (3) the SaleItem + Inventory view, (4) the View Link between the main form and the subform data, (5) define the new variables LineTotal and SaleTotal, and (6) create the JSPX page by dragging on the main form and subform data.

Begin by creating the SaleCustomerView that pulls all of the attributes from the Sale entity and a few customer attributes such as Lastname, Firstname, Email, and Phone. Because the Sale and Customer objects are already in JDeveloper, it should automatically know to build the relationship association between the two tables.

Action

- Create Sale + Customer View.
- Create CustomerLOVView as a Combo Box/LOV.
- Assign it to the SaleCustomerView.
- Create PaymentMethodLOVView
- Assign it to the SaleCustomerView.
- Create SaleItem + Inventory View.
- Add LineTotal=Saleprice*Quantitysold attribute.
- Create new View Link
- SaleToSaleItemViewLink to join the SaleCustomerView to the SaleItemInventoryView.
- In SaleCustomerView, create
- SaleTotal=SaleItemInventoryView.sum(“LineTotal”)
- Set formats for money items to Number: #,000.00 not to Currency.

Create a view using a read-only query to build the list of values for Customer so users can look up customers by name or phone. Use the UI Hints to assign it to a Combo Box List of Values. Create a similar view for PaymentMethod but use the default list box as the UI. Remember to open the SaleCustomerView and add both LOV views as accessors. Then assign the CustomerLOV to the CustomerID attribute in the Sale table and the PaymentMethodLOV to the PaymentMethod attribute.

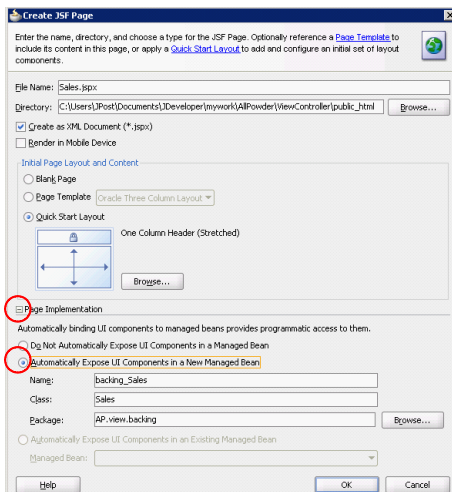
Create a new View for the subform that uses all attributes from the SaleItem table and the ModelID and ItemSize as read-only attributes from the Inventory table. Add a calculated attribute: $\text{LineTotal} = \text{Saleprice} * \text{Quantitysold}$.

Create a View Link between the master query SaleCustomerView and the details query SaleItemInventoryView that joins them on the matching SaleID entries. With that link created, re-open the SaleCustomerView and create a new attribute that sums the LineTotal: $\text{SaleTotal} = \text{SaleItemInventoryView.sum}(\text{"LineTotal"})$. If you assign formats to the monetary attributes, it is better to choose the Number format ($\#,##0.00$) instead of the Currency format. To compute the sales tax, the code needs to retrieve the SaleTotal value and it is more difficult to parse if it includes a currency (\$) sign. It is even more complicated if the currency might change in an international environment.

It seems like a lot of steps to define objects before you even get to look at the page. As you learned in the last chapter, it is critical to first define the data views for the master and detail sections along with the Link View that connects them. But, now it is relatively easy to create the basic page.

Creating the sales page requires one critical new step, so be careful when you first start the wizard. Right-click the Web Content section of the View Controller and choose the New option. Under Web Tier, choose JSF, and JSF Page in the list of items. Figure 7.17 shows the main setup page. Enter Sales.jspx as the name of the page and stick with the default One Column Header layout. The big change is to expand the Page Implementation section and choose the option to “Automatically Expose UI Components in a New Managed Bean.” This is the option that generates the Sales.java page and automatically builds the links to the jsp page.

Figure 7.17



Once the page has been created, you can work with the `jspx` page just as before. Add a title using an Output-Text control, entering a label (Sales) and assigning it a style (PageHeader). To ensure consistent styles across pages, drag or create the `Powder.css` style sheet onto the page. Edit this file and add a definition for `PageTitle` that sets the font size and color of the title.

Now you can drag the data elements on to the main page. Remember to refresh the Data Controls and find the `SaleCustomerView2` object that serves as the parent for the `SaleItemInventoryView2`. Drag the `SaleCustomerView2`

onto the page and select the `Form/ADF Form` option. Be sure to check the boxes to include navigation and submit buttons. Drag the `SaleItemInventoryView2` onto the bottom of the page and select the `Table/ADF Table` control. Choose the `Select and Sort` options but not the `Query` one.

Clean up the form and subform—removing unneeded columns, resizing, and reformatting to make the items fit on a reasonable page. From the Data Controls, drag the `Operation Commit` and drop it on the `Submit` button. You can keep the `Submit` name. Do not add any other buttons yet. Run the form and ensure that it works—particularly the totals.

Action

- Create a new `Sale.jsx` page.
- Expand the `Page Implementation` section.
- Check: `Automatically Expose UI Components in a New Managed Bean`.
- For the Main form, drag `SaleCustomerView` as `Form/ADF Form`.
- For the subform, drag `SaleItemInventoryView` as `Table/ADF Table`.
- Clean up the form and subform.
- Drag the `Commit` operation onto the `Submit` button.
- Do not add any buttons yet.

Figure 7.18

```

<f:view>
<af:document ... >
<af:resource type="css" source="Powder.css" />
<af:form ... >
  <af:panelStretchLayout ... >
    <f:facet name="top"> ... title ... </f:facet>
    <f:facet name="center">
      <af:panelFormLayout ... >
        ... inputText controls for main form ...
      <af:facet name="footer">
        <af:panelGroupLayout layout="vertical" ...>
          <af:panelGroupLayout layout="horizontal" ...>
            ... command buttons ...
          </af:panelGroupLayout>
        </af:panelGroupLayout>
      </af:facet>
    <af:table ...>
      ... subform columns
    </af:table>
  </af:panelFormLayout>
</f:facet>
</af:panelStretchLayout>
</af:form>
</af:document>
</f:view>

```

You should spend a few minutes and look through the Source (XML) for the jsp page. The wizard template relies on several `<af:panel...>` controls to handle formatting. You will want to manually edit some of the formatting so it helps if you understand the basic concepts. Figure 7.18 summarizes the overall structure of the layout controls. Your initial page might be slightly different—particularly in terms of placement of the subform table. Here, the table has been moved inside the main `panelStretchLayout` which improves the overall flow layout of the page. More importantly, notice the use of the `<af:panelGroupLayout layout="horizontal"...>` control. Without this control, all items added to the page are displayed in a vertically stacked list. The `panelGroupLayout` is used when you want to put multiple items side-by-side. You will need to add a couple of these to improve the display of some new buttons that will be added.

The main goal now is to add a button that can be used to compute the Sales tax using the function in the database. The basic steps are straightforward except you must remember to define the interface to the function in the `SalePageDef.xml` file.

Open the `SalePageDef.xml` file and switch to the Overview tab. In the main Bindings window click the plus sign to add a new binding definition. Choose the `methodAction` option in the pop-up window. Select the `DBModuleDataControl` in the top window of the Create Action Binding form, but do not expand it. In the Operation selection box, choose the `callSalesTaxFunction` that you created earlier. Click the OK button to add the `methodAction` entry to this `PageDef` file. You must perform this step for every page that uses a function stored in an application module. Figure 7.19 shows the `methodAction` entry that was created in case you need to edit it later.

In the ADF Faces list on the top right, find a command button and drag it onto the form near the Sales tax control. You will move it later so do not worry about exact placement. Double-click the new button and accept the defaults to create the code structure in the Java backing page. Figure 7.20 shows the code that you will

Action

- Drag a button onto the page from the ADF control list.
- Double-click the button to generate the code entry in the java file.
- Add code to handle the event that calls the Sales Tax Function.
- Open the file `SalesPageDef.xml` and select the Overview tab.
- Click the Add (plus) button in the Bindings window and select `methodAction`.
- Select the `DBModule` in the top window
- Choose `callSalesTaxFunction` in the Operation selection list.
- Run the form and test the tax button.

Figure 7.19

```
<methodAction id="callSalesTaxFunction" RequiresUpdateModel="true"
  Action="invokeMethod" MethodName="callSalesTaxFunction"
  IsViewObjectMethod="false"
  DataControl="DBModuleDataControl"
  InstanceName="DBModuleDataControl.dataProvider"
  ReturnName="DBModuleDataControl.methodResults.
callSalesTaxFunction_DBModuleDataControl_dataProvider_callSalesTaxFunction_
result">
  <NamedData NDName="value" NDValue="0"
    NDType="java.lang.Number"/>
</methodAction>
```

need. You should use copy-and-paste to transfer this code into your editor. Note that you will already have the first line and closing brace, although the name of your button might be different. You will also have to edit the code in two places. The code needs to know the names of two input boxes on your form: It13 is the ID of the text box that holds the SaleTotal value and It3 is the ID of the text box that displays the sales tax. Go back to your jsp page, select each of these two controls and note the ID values. If necessary, edit the code and enter the correct values of the controls on your form.

It would be nice to put the Tax button next to the text box for the sales tax. Edit the Source for the jsp code and place a tag above the `inputText` tag for the sales tax:

```
<af:panelGroupLayout layout="horizontal" id="pg16">
```

Place the closing tag after the closing tag for the `inputText`:

```
</af:panelGroupLayout>
```

Return to the Design view and drag the Tax button and drop it next to the Sales tax text box. If necessary, you can hand-edit the Source code to place everything correctly. Finally, save everything and test run the `Sale.jsp` page to test it. Click the new Tax button and watch for errors. You can check the calculation to ensure

Action

Drag a button onto the page from the ADF control list.

Double-click the button to generate the code entry in the java file.

Add code to handle the event that calls the Sales Tax Function.

Open the file `SalesPageDef.xml` and select the Overview tab.

Click the Add (plus) button in the Bindings window and select `methodAction`.

Select the `DBModule` in the top window

Choose `callSalesTaxFunction` in the Operation selection list.

Run the form and test the tax button.

Figure 7.20

```
public String cb8_action() {
    BindingContainer bindings =
        BindingContext.getCurrent().getCurrentBindingsEntry(); OperationBinding
    operationBinding=
        bindings.getOperationBinding("callSalesTaxFunction");
    // set parameter
    Map paramsMap = operationBinding.getParamsMap();
    RichInputText salesTotalTextBox = getIt13();
    DecimalFormatSymbols sym = new DecimalFormatSymbols();
    sym.setDecimalSeparator('.');
    sym.setGroupingSeparator(',');
    DecimalFormat form = new DecimalFormat("");
    Number v;
    try {
        v = form.parse(salesTotalTextBox.getValue().toString());
    } catch (Exception e) {
        return null; // no value
    }
    paramsMap.put("value", v);
    Object result = operationBinding.execute();
    // System.out.println("### Result Sales.java= " + result);
    RichInputText salesTaxTextBox = getIt3();
    salesTaxTextBox.setValue(result.toString());
    return null;
}
```

that the tax is seven percent of the total value—but the existing tax values are likely to be correct. You could edit the existing tax value and then click the Tax button to see if it is recomputed correctly.



Activity: Add Buttons for New Sale, New SaleItem, and Search

Now that you know how to write code and connect it to the page, you need to add a few additional features to the Sale form to make it useful. Most importantly, you need the ability to add a new Sale and to enter new items being purchased. The two processes are similar. Along the way you can add a search option to jump to Sales based on the SaleID.

Begin by writing the PL/SQL functions to insert a new sale—this function needs to return the SaleID that was created so it can be used on the page. Figure 7.21 shows the PL/SQL code for creating two functions: (1) To insert a new Sale using the current date and return the generated SaleID; and (2) To insert a new SaleItem given the SaleID and the SKU of the item being sold. The SaleItem code uses a query to look up the List Price for the item and set it as the default

<p>Action</p> <ul style="list-style-type: none"> Create PL/SQL for inserting a new Sale. Create PL/SQL for inserting a new SaleItem. Add Java functions in DBModuleImpl.java to call the PL/SQL code. Add the Client Interface. Add a button to the main page for New Sale. Write the code for the New Sale button. Add the methodAction for insertNewSale in the SalePageDef.xml file.
--

Figure 7.21

```

CREATE OR REPLACE FUNCTION InsertNewSale return NUMBER
AS
BEGIN
    INSERT INTO Sale(SaleDate)
    VALUES (SysDate);
    Commit;
    RETURN sq_Sale.CURRVAL;
END InsertNewSale;
/
CREATE OR REPLACE FUNCTION InsertNewSaleItem(
    inSaleID NUMBER, inSKU NVARCHAR2)
    return NUMBER
AS
    v_ListPrice NUMBER(10,4);
BEGIN
    SELECT ItemModel.ListPrice INTO v_ListPrice
    FROM INVENTORY
    INNER JOIN ITEMMODEL
    ON INVENTORY.MODELID=ITEMMODEL.MODELID
    WHERE INVENTORY.SKU=inSKU;
    INSERT INTO SaleItem(SaleID, SKU, QuantitySold, SalePrice)
    VALUES (inSaleID, inSKU, 1, v_ListPrice);
    Commit;
    return 1;
EXCEPTION WHEN others THEN
    RETURN 0;
END;
/

```

Sale Price in the sale. Notice that both functions require a Commit; statement to force the database updates—otherwise the new values will not be visible to other processes—including the Sale page.

Next edit the DBModuleImpl.java file to add two functions to call the PL/SQL commands. The new functions are similar to the tax function—basically build the function that calls the helper utility and returns the results. Figure

7.22 shows the two functions. The first one (InsertNewSale) is easier because it does not have any input parameters. Both functions return a numeric result. The return for NewSale is critical because it contains the SaleID generated. The value for NewSaleItem simply indicates whether the insert was successful (1) or not (0). For example, someone might accidentally try to insert the SKU of a product that has already been entered for that sale.

Remember that when you add functions to the DBModuleImpl code, you also need to add them to the Client Interface. Open the DBModule.xml file and use the Overview and Java tabs to edit the Client Interface and add the two new functions. You also need to add them to the SalesPageDef.xml file as method actions. You probably have to refresh the Data Control before you can see the new functions in the PageDef file.

With the functions defined and published, you can add a new button to the Sale form. Open Sale.jspx in Design view and drag an ADF command button onto the form, placing it next to the existing Submit button. Double-click the button to open the Java editor. Figure 7.23 shows the code for the button. It consists of three parts: (1) Call the PL/SQL function through the DBModule, (2) Requery the main SaleCustomerView to retrieve the new entry, and (3) Jump to the newly inserted row based on the SaleID value returned as a result to the first function.

Add a text box (New SKU) and a button at the bottom of the subform so users can add a new sale item. Place them in a panelGroupLayout layout="horizontal" layout panel.

Action

Add a text box and a button for a New Sale Item.

Write the code for the New Sale Item.

Add methodAction for insertNewSaleItem in SalePageDef.xml file.

Add a text box and button for SaleID search.

Write the code for the SaleIDSearch.

Figure 7.22

```
public Number InsertNewSale() {
    Number result = (Number)callStoredFunction(
        NUMBER, " InsertNewSale()",
        null);
    return result;
}

public Number InsertNewSaleItem(Number SaleID, String SKU) {
    Number result=(Number)callStoredFunction(
        NUMBER, " InsertNewSaleItem(?,?)",
        new Object[]{SaleID, SKU});
    return result;
}
```

```

public String cb9_action() {
    // Add event code here... New Sale
    BindingContainer bindings
        = BindingContext.getCurrent().getCurrentBindingsEntry();
    OperationBinding operationBinding
        = bindings.getOperationBinding("InsertNewSale");
    Object result = operationBinding.execute();
    //System.out.println("### Result new SalesID= " + result);
    // Requery the page to load the newly inserted row
    DCBindingContainer dcBindings = (DCBindingContainer)
        BindingContext.getCurrent().getCurrentBindingsEntry();
    DCIteratorBinding iterBind =
        (DCIteratorBinding)dcBindings.get("SaleCustomerView2Iterator");
    iterBind.executeQuery();
    //Find the new row
    iterBind.setCurrentRowWithKeyValue(result.toString());
    return null;
}

```

Figure 7.23

Figure 7.24 shows the code you need to add to the Sale.java page. You will need to modify the two get statements so they match the ID values of the SaleID and new SKU text boxes on your form.

Figure 7.24

```

public String cb6_action() {
    // Add event code here...Insert new SaleItem
    BindingContainer bindings
        = BindingContext.getCurrent().getCurrentBindingsEntry();
    OperationBinding operationBinding
        = bindings.getOperationBinding("InsertNewSaleItem");

    // set parameters
    RichInputText SIDTextBox = getIt1();
    RichInputText SKUTextBox = getIt20();
    Map paramsMap = operationBinding.getParamsMap();
    NumberFormat fmt = NumberFormat.getInstance();
    Number v;
    try {
        v = fmt.parse(SIDTextBox.getValue().toString());
    } catch (Exception e) {
        return null; // no value
    }
    paramsMap.put("SaleID", v);
    paramsMap.put("SKU", SKUTextBox.getValue().toString());
    Object result = operationBinding.execute();

    // Requery the subform to load the newly inserted row
    DCBindingContainer dcBindings = (DCBindingContainer)
        BindingContext.getCurrent().getCurrentBindingsEntry();
    DCIteratorBinding iterBind =
        (DCIteratorBinding)dcBindings.get("SaleItemInventoryView2Iterator");
    iterBind.executeQuery();
    return null;
}

```

```

public String cb5_action() {
    // call find row on iterator
    RichInputText findSIDTextBox = getIt17();
    try {
        DCBindingContainer dcBindings = (DCBindingContainer)
        BindingContext.getCurrent().getCurrentBindingsEntry();
        DCIteratorBinding iterBind
        = (DCIteratorBinding)dcBindings.get("SaleCustomerView2Iterator");
        String sSID = findSIDTextBox.getValue().toString();
        iterBind.setCurrentRowWithKeyValue(sSID);
    } catch (Exception e) {
        findSIDTextBox.setValue("Invalid");
    }
    return null;
}

```

Figure 7.25

Notice that when a new Sale is added, the code jumps to the new item based on the generated SaleID value. You can use this same process to add a search text box on the form. Add a new horizontal panelGroupLayout to the top of the form and insert an ADF Text Box and ADF command button. Figure 7.25 shows the code for the simple search function. Notice the similarity to the insert New Sale code. Remember to change the get function to match the ID of the text box you created. The ADF Faces system supports more complex searches on multiple columns. The Oracle documentation contains examples and other samples can be found on the Web.

You are close, but need to complete one more step—twice. Recall that the SalePageDef.xml file needs to have an entry to enable the Sale.java page to call the DBModuleImpl code. You need two entries—one for the insertNewSale and one for the insertNewSaleItem function. If you understand the exact syntax, you could add the elements by hand. Otherwise, it is easier to drag two new temporary buttons onto the Sale.jspx page—using each of the two functions in the Data Controls list. Manually edit the XML source and remove the two buttons, and edit the Sale.java page to remove the references to the two new buttons. The new elements in the SalePageDef.xml file should remain and be available for your real functions.

Save everything and run the page to test it. Try searching for sales by ID values. Try adding a new sale and adding new Sale Items. Ultimately, the page could use more features, such as delete options for both the sale and the sale items. The process is straightforward and you might even be able to use the built-in delete operation functions. However, allowing anyone to delete data can be dangerous. Now that you know how to create database functions and call them from the page, you can handle relatively complex tasks.



Activity: Update Inventory with Data Triggers

Maintaining quantity on hand statistics for inventory is one of the trickiest elements in programming business forms. Reexamine the Inventory table and notice that it contains the column QuantityOnHand. This value represents the current number in stock for a specific item. The value of the column is that clerks can quickly check the column to see if certain sizes are available. Also, managers can get a quick look at the list of items that might be under- or overstocked. Technically, this value would not have to be stored in the database—if you have a

complete list of all purchases, sales, and adjustments, you could use a query to compute the total number currently in stock. However, with thousands of items and sales, this query might take too long to run. Consequently, you need a mechanism to update this value on the fly. Whenever an item is sold, the corresponding quantity should be subtracted from the quantity on hand. In Oracle, this subtraction can be handled with triggers on the data tables. It

is best to put this code in the database because then the code is executed no matter how the tables are updated. These data triggers are simply code that is executed whenever a specified event occurs. The three events are DELETE, INSERT, and UPDATE. You can attach a trigger before or after the change is written to the database. In the inventory situation, you want to attach your code to the AFTER event so that you do not change the quantity on hand (QOH) until after the change has truly been made.

The first step is to examine the tables and understand how they are related. You need to change the Inventory table whenever changes occur to the SaleItem table. The SaleItem table specifies the SKU value that matches exactly one row in the inventory table. Also, when testing, remember that you need a matching entry in the Sale table to provide the SaleID key. You should look at some sample data in the three tables so you can enter consistent values to test. In this case, a new SaleID of 3000 to CustomerID 582 by EmployeeID 5 should work. SKU values of 500000 and 500010 both have an initial QOH of 10 units.

The next step is to think about the events that can occur and determine what they mean and how they will affect the QOH. It is easier to understand the process by considering one event at a time. Think about the first step in a sale. A row is entered into the Sale table: INSERT INTO Sale (SaleID, CustomerID, EmployeeID) VALUES (3000, 582, 5). You could enter the data for SaleDate and so on, but since this data is temporary, these three items are sufficient. Now, the next logical step that occurs in a sale is that the SKU for the item being purchased is entered into the SaleItem table: INSERT INTO SaleItem (SaleID, SKU, QuantitySold, SalePrice) VALUES (3000, 500000, 1, 100). At this point, the QuantitySold of one unit means that the system should subtract that value from the quantity on hand. To accomplish this task automatically, you need to establish an AFTER INSERT trigger on the SaleItem table. Figure 7.26 shows the SQL used to create this trig-

Action

Insert a new row into the Sale table with a SaleID of 3000, CustomerID of 582, and EmployeeID of 5.

Create the AFTER INSERT trigger for the SaleItem table.

Insert a new row into the SaleItem table (3000, 500000, 1, 100).

Check the value of QuantityOnHand in the Inventory table for SKU=500000 and verify it decreased from 10 to 9.

Figure 7.26

```
CREATE OR REPLACE TRIGGER NewSaleQOH
  AFTER INSERT ON SaleItem
  FOR EACH ROW
BEGIN
  UPDATE INVENTORY
  SET QuantityOnHand = QuantityOnHand - :NEW.QuantitySold
  WHERE SKU = :NEW.SKU;
END;
/
```


ger. First the trigger is given a unique name. The second line specifies that the trigger should be fired after a row is inserted into the SaleItem table. The third line indicates that the code body should be executed once for each row being inserted. The begin/end block holds the main code, which consists of a single SQL UPDATE statement. The UPDATE statement should look familiar, with a small twist. The twist is that

it refers to the values being inserted into the SaleItem table using the :NEW syntax to reference the data that was just added to that table. The statement simply tells the database to subtract the new quantity sold from the existing quantity on hand for the SKU value just entered into the SaleItem table. When you have successfully created the trigger, issue the INSERT statement to add the row to the SaleItem table. Now verify that the QOH was modified with the query: SELECT SKU, QuantityOnHand FROM SaleItem WHERE SKU=500000.

You could continue to issue INSERT commands for different quantities, and the quantity on hand will decrease. Everything seems to be fine. However, what happens if there is a data entry error? Try deleting the row you inserted: DELETE FROM SaleItem WHERE SaleID=3000 And SKU=500000. Check the QOH in the Inventory table and you will see that it does not change. Why is that bad? Because the delete statement implies that the item was not actually sold, and since you have already subtracted the quantity, you need to add that value back to the QOH. In other words, you need another database trigger—one that fires when a row is deleted in the SaleItem table. Figure 7.27 shows the statement to create the trigger. This code is similar to the after insert version. The only differences are that the quantity sold is added back to the quantity on hand, and the syntax uses the :OLD reference. The :OLD reference simply means to use the values that existed before the row was deleted. In this case, there are no :NEW values because the Delete command does not create anything. To test the new trigger, insert the SaleItem row again and check the quantity on hand. If you are using the SQL Plus Worksheet, you can use the Back and Forward query buttons to bring up the SELECT statement that you used before. Now, delete the SaleItem row and check the quantity on hand again. It should be restored to its value before the latest INSERT command.

The two triggers you created are powerful tools. Once they have been defined, you never need to think about them. Anytime a process inserts or deletes a row,

Action

Delete the SaleItem row (SaleID=3000 And SKU=500000).

Check the quantity on hand.

Add the AFTER DELETE trigger.

Insert the SaleItem row again.

Check the quantity on hand.

Delete the SaleItem row.

Check the quantity on hand.

Figure 7.27

```
CREATE OR REPLACE TRIGGER DelSaleQOH
  AFTER DELETE ON SaleItem
  FOR EACH ROW
BEGIN
  UPDATE INVENTORY
  SET QuantityOnHand = QuantityOnHand + :OLD.QuantitySold
  WHERE SKU = :OLD.SKU;
END;
/
```

they are activated and inventory is changed immediately. You could test these actions using the Sale form, and you should see the same results. However, there is still something missing. One of the trickiest aspects to event programming is that you need to think hard about possible actions by users,

Action
 Add the ON UPDATE trigger.
 Check the quantity on hand.
 Issue an update to change the QuantitySold in the SaleItem table.
 Check the quantity on hand.

and the consequences. In the inventory situation, what happens if a clerk goes back and changes a value? Originally, an SKU and quantity were entered, then the clerk sees an error or a customer changes his mind. Try it first with a change in quantity. Check the current value for QOH then insert the row to sell one unit. Check the QOH again to see that it was reduced by one, say from nine to eight units. Now, consider what if the customer actually purchased two units. Issue the statement to change the QuantitySold to two units: UPDATE SaleItem SET QuantitySold=2 WHERE SaleID=3000 And SKU=500000. Check the QOH and you will see that it still shows only one item was removed from inventory (eight units remaining instead of seven).

You need to add an UPDATE trigger to the SaleItem table to handle this problem. Figure 7.28 shows the code to create the trigger. Again, it uses a familiar UPDATE statement. However, check the use of the :OLD and :NEW references carefully. They contain the heart of the logic. The :OLD values are the data that was stored in the SaleItem table before the update was initiated. The :NEW values are the data in the row after it has been changed. In this case, the QuantitySold changed from one (old) to two (new). For the specified product SKU, this query adds the old value back and subtracts out the new value instead. Remember that a change in quantity means that the original subtraction was incorrect, so it is restored while the new value is subtracted. Again, you should test this trigger by checking the current QOH value, issuing an Update statement to the SaleItem table to change the quantity sold value, and then examine the new QOH to see that it holds the proper total.

If you look closely at the Update trigger code and think about the problem for a minute, you will see that one additional situation has to be handled. What happens if a clerk changes the SKU? In this case, you need to add the QuantitySold back to the original SKU item, then subtract the QuantitySold from the new SKU item. Of course, the QuantitySold might have been changed at the same time, so you need to be careful about which one you add and subtract.

Figure 7.28

```
CREATE or REPLACE TRIGGER ChangeSaleQOH
  AFTER UPDATE ON SaleItem
  FOR EACH ROW
BEGIN
  UPDATE Inventory
  SET QuantityOnHand = QuantityOnHand + :OLD.QuantitySold - :NEW.
  QuantitySold
  WHERE SKU = :OLD.SKU;
END;
/
```

```
CREATE or REPLACE TRIGGER ChangeSaleQOH
  AFTER UPDATE ON SaleItem
  FOR EACH ROW
BEGIN
  IF (:OLD.SKU = :NEW.SKU) THEN
    UPDATE Inventory
    SET QuantityOnHand = QuantityOnHand +
      :OLD.QuantitySold - :NEW.QuantitySold
    WHERE SKU = :OLD.SKU;
  ELSE
    UPDATE Inventory
    SET QuantityOnHand = QuantityOnHand + :OLD.QuantitySold
    WHERE SKU = :OLD.SKU;
    UPDATE Inventory
    SET QuantityOnHand = QuantityOnHand - :NEW.QuantitySold
    WHERE SKU = :NEW.SKU;
  END IF;
END;
/
```

Figure 7.29

Figure 7.29 shows a revised version of the AFTER UPDATE trigger. At this point, it is useful to point out the value of the CREATE or REPLACE clause in Oracle. Since the trigger already exists, you cannot simply issue another CREATE statement with the same trigger name. Normally, you would have to DROP the original trigger and then create the new one. The CREATE or REPLACE statement essentially combines these two operations to save you a step. The IF statement divides the trigger so that it handles the two cases separately. Actually, you could have created two completely separate triggers using a WHEN condition, but it is easier to see the code with all of the cases in one trigger. The IF statement is true when the SKU was not changed. This Update statement is the same as the simple update trigger. The ELSE condition handles the case where the SKU was changed (and the QuantitySold might or might not have been altered). The first UPDATE statement restores the old quantity subtracted from the original SKU value (old quantity, old SKU). The second UPDATE statement subtracts the new QuantitySold from the new SKU inventory level.

These three triggers should now handle all of the sales situations that affect the inventory quantity on hand. You should reset the QOH value and test all of the changes. In particular, in the SaleItem row change both the QuantitySold and the SKU.

Of course, if you have created purchase order and purchase item tables, you would have to add similar triggers to the purchase item table. The only difference is that a purchase adds quantity to the QOH instead of subtracting it, so you have to reverse the signs in the code.



Activity: Define Transactions

Transactions consist of multiple changes that must succeed or fail together. One of Oracle's strengths is its support to ensure that transactions are completed correctly. In particular, all changes are written to journal logs. If the system crashes in the middle of a transaction, the system can still recover the transactions that were interrupted or roll them back to the point where the changes began. The other

important aspect of transactions is the ability to prevent or handle collisions of two processes altering the same data at the same time.

Katy, the manager at All Powder, has noticed that many customers do not like being charged for damages caused to the rental equipment. Some of them believe that the equipment is

simply wearing out and failing. She also notices that there can be several complaints about a specific rental—particularly when it involves multiple items. David, the rental manager, agrees, but still wants to be able to track the cumulative charges. He has suggested that any reduction in the damage charge be recorded as a discount to that customer. That way, he can track the total damages, as well as which customers might receive the most discounts. Katy also likes the discount idea, because she wants to implement a discount program for employees who rent equipment. Since multiple discounts can be applied to a single rental, a new table is needed. Figure 7.30 shows the table keyed by both RentID and DiscountDate.

You can build a form to handle data entry for the employee discounts, but do not do that now. Eventually, you will need a Rental form that is similar to the Sale form; then you could add a button onto that form to open a discount form. However, all of the main work will be done in a function or procedure stored in the database, so this section concentrates on creating that function. The process for handling customer discounts for disagreements over damage discounts has an interesting complication. You need to create a transaction that decreases the repair charges and adds a row to the RentalDiscount table for the same amount. Both actions must be completed together.

The function needs to be written in PL/SQL. Later, you can add a connector in the DBModuleImpl.java code and then create a form that calls it. To reverse repair charges, the function needs to know the RentID and the SKU from the RentItem table. The function also needs to know the amount to be subtracted—it might be less than the total damage charge. Also, a reason for the discount should be passed to the function so it can be stored in the new RentalDiscount table. So, the input values will be: RentID, SKU, DiscountAmount, and Reason.

Action

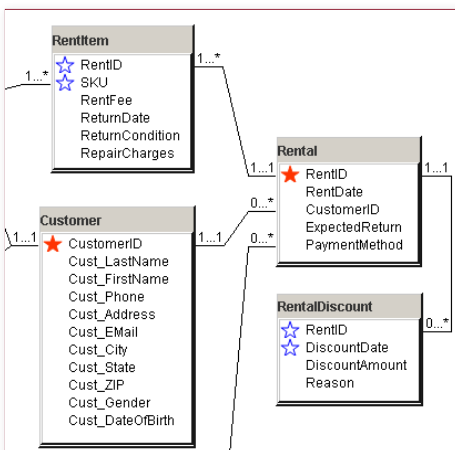
Create the full ON UPDATE trigger.

Check the quantity on hand.

Change the QuantitySold and SKU (to 500010) in the SaleItem row.

Check the quantity on hand for SKU 500000 and 500010.

Figure 7.30



Particularly for Web applications, it is important that transaction code be written in the PL/SQL database. You cannot handle transaction processing on the Web page because of potential transmission delays and the fact that a user might walk away from an open session—potentially holding a transaction open for long periods of time.

<p>Action</p> <p>Create the RentalDiscount table.</p> <p>Create the GiveCustomerRentalDiscount function.</p> <p>Test the function.</p>
--

In terms of processing, the function needs to subtract the discount amount from the existing repair charge, with a simple test to ensure the charge does not go below zero. Then the function needs to store the amount and reason in the new RentalDiscount table. The two changes should be written as a transaction so they both succeed or fail together. Because Oracle automatically has transaction-processing enable, this step is easy. However, you need to trap errors and return a negative value if something goes wrong. Later, the form submitting the request could notify the user or take additional steps if the transaction fails.

Figure 7.31 shows the code for the new function. Examine the RentItem table to see that the SKU is handled as characters instead of numbers. The first step of the function is to retrieve the existing value for RepairCharges and then subtracting the proposed discount. If the discount is larger than the repair charge, the repair charge is reduced to zero. However, the discount is unchanged—a decision that should be verified with management. The UPDATE statement writes the

Figure 7.31

```

CREATE OR REPLACE FUNCTION GiveCustomerRentalDiscount
  (oldRentID NUMBER, oldSKU NVARCHAR2,
   DiscountAmount NUMBER, Reason NVARCHAR2)
  RETURN NUMBER
AS
  OldRepairCharge NUMBER(10,4);
BEGIN
  SELECT RepairCharges INTO OldRepairCharge FROM RentItem
  WHERE RentItem.RentID=oldRentID AND RentItem.SKU=oldSKU;
  OldRepairCharge := OldRepairCharge-DiscountAmount;
  IF (OldRepairCharge < 0) THEN
    OldRepairCharge := 0;
  END IF;

  UPDATE RentItem SET RepairCharges=OldRepairCharge
  WHERE RentItem.RentID=oldRentID AND RentItem.SKU=oldSKU;

  INSERT INTO RentalDiscount(RentID, DiscountDate,
    DiscountAmount, Reason)
  VALUES (oldRentID, SYSDATE, DiscountAmount, Reason);
  Commit;

  Return OldRepairCharge;
EXCEPTION
  WHEN others THEN
    ROLLBACK;
    Return -1;
END GiveCustomerRentalDiscount;
/

```

```

set ServerOutput on
declare result NUMBER;
begin
result := GiveCustomerRentalDiscount (
    4020, '200285', 15, 'Testing');
dbms_output.put_line('result: ' || result);
end;

```

Figure 7.32

new repair charge back to the RentItem table. The INSERT statement stores the discount and reason for the change in the new RentalDiscount table. The Commit statement indicates the end of the transaction. If the transaction completes successfully, the discounted repair charge is returned to the caller. If anything goes wrong while processing the transaction, the EXCEPTION section is triggered, the changes are rolled back, and a negative value is returned to the caller to indicate a problem occurred.

If a more drastic event arises during the transaction processing, such as a computer crash, then the Oracle DBMS handles the transaction completion when it restarts. It will automatically read the log files and try to complete the transaction. The only work you, the developer, had to do was use the commit command to indicate the end of the transaction—plus handle simpler errors with the EXCEPTION section.

To test the function, start by finding a RentItem entry with a repair charge greater than zero. For example: RentID=4020, SKU='200285', RepairCharges=18. Then use a new query window in SQL Developer to call the function. Figure 7.32 shows the script for calling the function and printing the returned value. It uses two special Oracle script commands: set ServerOutput on and dbms_output.put_line(...). Run this script with the script processor in SQL Developer using the script button or F5. The result should be 3, which is the repair charge of 18 minus the 15 discount.

Database Cursors, Keys, and Locks



Activity: Read Rows of Data

Direct SQL commands are useful for DML issues where you need to change or delete rows of data. Sometimes you need program code to examine several rows of data. You need to use database cursors to handle these tasks. Consider the business question of sales by week. Katy wants to know if weekly sales increase more in the first part of the year or in the last part. In particular, she wants to know the average percent increase in weekly sales for the first weeks (1 to 15) compared to the last 15 weeks (38 to 52). Remember that SQL can perform calculations on data within the same row. SQL can also compute subtotals for groups of data. However, it is difficult to get SQL to compare data by subtracting values across two rows. Instead, it is easier to write a query that does the main computations, and then use cursor code to do the comparisons. Actually, the re-

Action
Create a new query.
Tables: Sale and SaleItem.
Create column TO_CHAR(SaleDate, 'ww') AS SaleWeek.
Create column QuantitySold*SalePrice AS Value.
Sum the Value column by week.

```

CREATE OR REPLACE VIEW WeeklySales AS
SELECT TO_CHAR(SaleDate, 'ww') AS SalesWeek,
       Sum(SalePrice*QuantitySold) AS Value
FROM Sale INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
WHERE SaleDate Is Not Null
GROUP BY TO_CHAR(SaleDate, 'ww');

```

Figure 7.33

cent Lead and Lag functions in Oracle make this problem easy to handle in SQL; but build the cursor for practice.

Begin by creating a query that computes total sales by week. Figure 7.33 shows the query. Note that you need to format the SaleDate using the TO_CHAR function with a format of 'ww' to get the number of the week. Make sure you compute the Sum of the price times quantity and that the total is computed for each week with the GROUP BY clause. A couple of entries have missing dates, so they can be removed from this query. Use the CREATE VIEW line at the top to save the query, but make sure you test the query before you add this line.

The next step is to compute the percentage change between the rows. The code for this step will be created within a function in a new SalesAnalysis package. Eventually, you can add a button and result box to a form to display the computation, but for now, it is faster to build the command and test it in PL/SQL.

The next step is to write the code that computes the average percent increase. For each pair of rows, the code needs to subtract the two values and divide by the value in the prior row to yield a percentage change. This percentage needs

Figure 7.34

```

CREATE OR REPLACE PACKAGE SalesAnalysis AS
  FUNCTION AvgPercentWeeklyChange return REAL;
END SalesAnalysis;
/
CREATE OR REPLACE PACKAGE BODY SalesAnalysis AS
  FUNCTION AvgPercentWeeklyChange return REAL IS
    CURSOR c1 IS
      SELECT SalesWeek, Value FROM WeeklySales;
    Avg1 WeeklySales.Value%TYPE;
    N Integer;
    PriorValue WeeklySales.Value%TYPE;
  BEGIN
    Avg1 := 0;
    N := 0;
    PriorValue := -1;
    FOR recSales in c1 LOOP
      IF PriorValue > 0 THEN
        Avg1 := Avg1 + (recSales.Value - PriorValue)/PriorValue;
        N := N + 1;
      END IF;
      PriorValue := recSales.Value;
    END LOOP;
    RETURN (Avg1/N);
  END AvgPercentWeeklyChange;
END SalesAnalysis;
/

```

Define the SELECT statement for the cursor to trace through

Create a variable to hold the value from the previous row with the same data type as the column in the table

Skip the first week because there is no prior value

Compute the percent change and keep a running total

Save the current row value and move to the next row

to be summed and eventually divided by the number of calculations to obtain the average percent increase. Figure 7.34 shows the main code. The SQL statement is opened as a cursor, which retrieves one row of data at a time using the loop. The AvgI variable keeps the running total of the percentage increase, while N counts the number of operations. The role of the PriorValue variable is the most important. At the end of the loop, it is assigned the value obtained from the current row. When the next row is retrieved, the program can now compare the current (new) value to the old (PriorValue) value. This trick is useful for many cursor-based programs, so you should study the code until you understand it. Use a basic SELECT statement to test the function in the package. Note that you need to use the FROM dual clause so you can see the result. Depending on the actual values in your database, the result should be about 16 percent. Note that this routine does not quite provide the detail Katy wants, but it is straightforward to restrict the query using starting and ending week parameters and call the function twice.

Action

Create the SalesAnalysis package with the AvgPercentWeeklyChange function.
Use SQL to call the function:
SELECT SalesAnalysis.
AvgPercentWeeklyChange FROM dual.



Activity: Test Optimistic Locks

The issue of locking records to prevent concurrency errors on Web pages, including Oracle JDeveloper, is almost always handled using optimistic locks. Pessimistic locks could result in problems with Web sites because a lock is set on a piece of data when it is opened by the user. The lock prevents other people from using the data until the original user completes a transaction. But with Web sites, the connection might be lost, the user might close a browser, or even walk away for lunch. Leaving locks open for extended times causes performance problems with the Web site. Instead, most tools use optimistic locks—assume that collisions are rare. The big benefit is that it requires minimal overhead by the DBMS server. The application code simply caches the original values read from the table and then tests for changes when data needs to be written back to the database. If the underlying values changed from when they were read, a collision has occurred and it is up to the current form to handle the error.

Begin by testing the current process embedded in JDeveloper. Open the EditCustomer.jspx page in JDeveloper and run it. You should see data for one of the

Action

Open the EditCustomer.jspx page in JDeveloper and run it.
Open a Database query window connected to AllPowder.
Run a SELECT query to verify the data matches the ID on the form.
Run an UPDATE query to change the ZIP Code.
Change the ZIP Code value on the Edit Customer form and click the Submit or Commit button.
Note the error message.

Figure 7.35

```
SELECT CustomerID, LastName, ZIP
FROM Customer
WHERE CustomerID=1110;
```



```
UPDATE Customer
SET ZIP='32777'
WHERE CustomerID=1110;
Commit;
```

Figure 7.36

customers—probably Amos Abbot with ID=1110. It is fine if you see a different customer, but note the ID number.

The essence of locking is to have two changes occur at the same time—or at least interfere with each other by making changes before the first one is finished. So open a database query window connected to AllPowder. You can click the Database tab in JDeveloper and click the AllPowder connection—or you could open a separate SQL Developer window. Run the small SQL query shown in Figure 7.35 to ensure the data matches that shown in the form. Be sure to use the CustomerID value that matches the form. Now alter the query so that it becomes an UPDATE query to change the ZIP Code. Figure 7.36 has an example. Be sure to include the Commit statement so the change is written to the database immediately.

Return to the EditCustomer form. The ZIP Code must still be the same because the page has not even returned to the server. Change the ZIP Code value to something different, such as 32555. Click the Submit button (which is really a Commit command) to write the change to the server and the database. Figure 7.37 shows the error message generated by the form—indicating that a collision has occurred. Remember, as a developer, you did nothing special when you created this form. JDeveloper automatically included the code, and the pop-up error message to test for changes by other users.

The message is clear, but the user might be confused about how to handle the problem. Fortunately, the user can simply click the resubmit button again and override the message. In any collision, there is a question of which change should succeed—the last one, or the one that was made by someone else. Although the default error message does not display the value recorded by the other user, the current user could cancel the changes and refresh the database to find the value. More likely, the current user will simply overwrite the other changes, which is usually fine for simple items such as ZIP Code, name, and address.

Figure 7.37

The screenshot shows a web form titled "Edit Customer" for Customer ID 1110. The form contains several input fields: Lastname (Abbott), Firstname (Amos), Phone (713-100-7401), Email (AbbottA83@msn.com), Zip (32555), Gender (Male), and Dateofbirth (12/9/1985). A red-bordered error message box is overlaid on the form, stating: "Error: Another user has changed the row with primary key oracle.jbo.Key[1110]". Below the error message, the text "Another user has changed the row with primary key oracle.jbo.Key[1110]." is visible. At the bottom of the form, there are navigation buttons (First, Previous, Next, Last) and action buttons (Submit, New Customer, Commit).

For more complex problems, such as account balances or inventory quantity, you should probably override the default processing and write code to automatically retrieve the updated values and then re-issue your update statements. If you follow the rule that these types of changes should be processed in PL/SQL on the server, you do not need to worry about the individual form code. Whenever possible, you should use PL/SQL and EXCEPTION processing to handle everything behind the scenes on the server and never bother the user.

Exercises



Many Charms

Inventory control is a critical success factor for determining profitability at Many Charms. Madison and Samantha need to watch the quantity on hand—particularly for the high-cost items. The suppliers are a complicating factor. Some of them are known for being inconsistent in delivering items ordered. As a result, Samantha and Madison have to carefully check every shipment they receive and cross-match it to the orders. Many times the shipment is missing items, and once in a while, the companies send items that were not ordered. These items have to be returned, but the supplier billing is just as bad. Madison has to continually watch the supplier bills to ensure that they are only billed for items they actually ordered and received. As a result of problems, she also wants to track the unordered items that were sent back, so if they show up on a bill, she can provide the details of when the item was returned.

1. Create a form to handle purchase orders to suppliers. Create a second form to handle received shipments. Be sure that it can handle receipt of partial orders and track the day that each partial order arrives. It must also handle receipt of unordered items (which should be stored in a separate table).
2. Add a button to the Received Orders form so that if they receive an interesting unordered item, it can be added to the orders and inventory and paid for. Create it as an entirely new order and be sure to handle optimistic locks and transactions.
3. Create a form that enables Madison to select a product category and metal, and then enter a percentage price increase. Write the SQL update code so that this increase is applied to the list price of the selected categories.
4. The company often ships orders to three states, each of which charge different sales tax rates. Write a function that takes the state code and the amount and returns the tax due.
5. Create a form and write a program that for a given type of charm and type of metal, computes the average of (1) the number of days between sales of that item, and (2) the average number of days between purchase orders for that item.



Standup Foods

While food items and celebrities are important aspects of the business, the day-to-day operations depend on managing the employees. In particular, Laura wants to reward the workers who continue to do well. The evaluation and rating system she has implemented is a major component of this plan. Now she has to set up the

system to make it easy to use so everyone can enter the necessary data. She also needs a way to analyze the data to help managers select the best employees for the next job, and to reward people who do well.

1. Create a form to enter data about an event, with an emphasis on the jobs performed by the employees and their evaluations. Make sure the form includes the revenue received from the event, the costs, and the dates involved. Create a separate form to enter and display data about employee specializations.
2. Create a form for Laura that lets her select a job category and then displays the top-rated employees in that category. (Hint: Create a subform and modify its Record Source query using code.) Create a text box so Laura can enter an average rating as a cut-off value. Create a second text box so Laura can enter a percentage raise increase. Add a button and write the code to give that raise increase to all of the selected employees.
3. Sometimes managers need to hire part-time workers on the spot. Create a simple form that lets managers add basic employee data without allowing them to see or change data for other employees.
4. Workers often want to estimate how much money they will make after all withholdings are deducted. Calculating withholdings is a complex process, but create a simple version to use as an estimate. The function should have number of exemptions, wage rate, and hours worked as inputs. It returns an estimate of the take-home pay. Use sample paychecks or research the Internet to estimate the tax withholding based on the number of exemptions. Create a simple form so employees can plug in these three values and receive the estimate.
5. Laura needs to provide some documentation to the bankers regarding the firm's growth. Create a new table with columns for month, revenue, costs, and percent change for revenue and cost. Write a query to compute the total revenue and costs per month and insert those values into the new table. Write a cursor-based program to compute the percent changes and insert the values into the appropriate columns.



EnviroSpeed

Tracking the knowledge of the workers and experts along with recording the experiences obtained in the many clean-up situations is a primary element of the company. You need to create forms that make it easy for workers to enter the data and knowledge gained. However, for the company to stay in business, you also need to track costs and revenue. Revenue is generally straightforward—the company bills based on the underlying costs, but payments are generally received over time. You will need a form to record the receipt of payments by the customers.

1. The company is trying to standardize its fee structure. Write a function that has inputs for the cost of the crews, the cost of expert time, the cost of chemicals, transportation costs, equipment, and miscellaneous costs. Compute a billing fee based on a percentage profit from each of these costs (crews: 20 percent, experts: 30 percent, chemicals: 15 percent, transportation: 10 percent, equipment: 50 percent, miscellaneous: 15 percent). Also include a \$50,000 fixed cost for overhead.

2. Create a form that enables managers to quickly put together a crew in an emergency. The form will have selection boxes for specialty and years of experience (subtract date hired from today). Clicking a button will retrieve a list of crew members meeting the desired conditions. Double-clicking on a name should add that person to the crew required for this disaster.
3. In the middle of an incident, crew members still need to record all of the details so they can be retrieved later. Create a form that enables them to enter the needed information. Be sure to include a way to quickly add a list of chemicals encountered in the incident. Mostly they should be able to select from a known list, but they sometimes encounter new chemicals. Be sure to control for concurrency, since several people may be entering data at the same time.
4. Write a program that evaluates payments by each customer. Assuming payments are due at the end of each month, assess an interest charge of one-half percent of the outstanding balance. Also, assess a late fee of \$200 for each month that a payment is late. Automatically add these values to the customer's balance. Note, You will have to enter several payments and late or missing payments to test the function.
5. Enter enough sample incident data to cover at least a year. Write a cursor-based program to calculate and display the percent increase in revenue per month.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks.

1. Make the forms easier to use by automating as many tasks as possible.
2. Examine the case for situations where you can use SQL to update records selected by the users. For example, consider price increases, employee raises, and automated inventory orders.
3. Look for potential reports that require comparing data over time. Write the cursor-based code to generate the necessary change data.

Applications

Chapter Outline

Applications, 176

Case: All Powder Board and Ski Shop, 176

Lab Exercise, 177

All Powder Board and Skip Shop Application, 177

Connecting Pages with Task Flows, 185

Testing Login Credentials, 190

A Report for One Customer Using the Login Data, 194

Connect Table Row to Detail Report, 198

Exercises, 215

Final Project, 216

Objectives

- Build applications that connect forms and reports.
- Create styles and templates for a consistent look.
- Add toolbars and menus to forms.
- Add Help files to the database application.

Applications

The main purpose of the DBMS is to store data efficiently and provide queries to retrieve data to answer business questions. But from the perspective of businesses, the true value of the DBMS lies in the applications that can be built on top of the database. Chapter 6 shows you how to build the forms and reports that make up the heart of an application. This chapter shows you the additional steps needed to make the application integrated and easy to use.

A finished application contains all of the forms and reports needed to solve a particular problem. It also needs finishing touches such as menus and other navigation links between forms. Additionally, you usually have to create Help files to provide assistance to users when they first learn the system.

Case: All Powder Board and Ski Shop

The primary application at All Powder Board and Ski Shop is the need to track sales and rentals. Of course, these applications also require you to build forms and reports for inventory items and customers as well. Eventually, you will have forms that store data into each of the tables in the relationship diagram. As shown in Figure 8.1, these forms and reports are integrated into a common style and structure. Particularly for Web sites, a home page or startup form is often used to direct users to the rest of the application. Buttons or menus are used to link to forms and reports. You can also create custom menus to highlight the main operations available to users on a particular form. Finally, you need to build help files to provide additional information or instructions to users.

Figure 8.1

The figure displays four screenshots of the All Powder Board and Ski Shop application, illustrating its integrated forms and reports. The screenshots are arranged in a collage:

- Startup form:** A web page titled "Customer Login" with input fields for "Last Name" (containing "Peck") and "Phone Number" (masked with asterisks).
- Help files:** A page titled "Introduction to All Powder Board and Ski Shop" with a "Help files" link highlighted in a red box.
- Customer Sales Report:** A report titled "Customer Sales Report" showing a list of customer records. The table has columns for name, phone number, and date. The first record is "1406 Abel Marshall 582.00" with a date of "1090 2010-01-04 514.00".
- Custom menu:** A page titled "Edit Ski Board Styles" with a "Custom menu" highlighted in a red box. It features a table of ski board styles with columns for "Style", "StyleDescription", and "Category".

Style	StyleDescription	Category
Back-Country	Back country and telemark	Ski
Cross-Country-Ski	Cross-country skate skis	Ski
Cross-Country-Trai	Traditional cross-country	Ski
Downhill	Basic downhill and racing	Ski
Extreme Board	Crazy boards	Board
Freestyle	Show jumps; shorter skis	Ski
Half Pipe	Turns and jumps	Board
Jump	Ski jumps from structures	Ski
Ride	Basic board	Board

Lab Exercise

All Powder Board and Skip Shop Application

Integrating the forms and reports is the first major step in creating the application. You need to identify the tasks performed by various user groups. With this knowledge, you can build sets of forms and reports that match the tasks of each group. While you are integrating the forms and reports, you should also make all of them consistent. Actually, you should create a design template and style sheet for an application before you begin creating forms and reports. The template contains the layout and primary elements that you want on every form, such as logo, title, and perhaps a copyright notice or other contact links. A design standard spells out details such as the fonts, page sizes, margins, colors, and naming conventions. A style sheet is used on the Web to provide a standard set of fonts and styles that can be applied by name.

Web pages have used cascading style sheets (css) for several years. Oracle ADF has the ability to create skins—which are essentially style sheets that are embedded in the application. For example, you can define a style for a PageTitle that is applied to an Output Text box on the top of each page. Later, designers can simply edit the skin or style sheet and all of the pages will pick up the changes automatically when they are displayed. You should create a style sheet for every application; and you will probably need to hire a graphics designer to help with the details.

A template form in Oracle is a blank page that contains standard design elements. You simply create a new blank form, add the logo, title, menu bar, and any code that will apply to all forms. Save the form with a name that everyone will recognize. You can create multiple templates—such as one for input forms and one for reports. However, it is best to stick with a small number to ensure that developers choose the correct template. A nice feature of Oracle templates is that even after they are applied to pages, you can go back and alter the underlying template and all changes will be reflected on the pages that use that template. However, it is difficult to apply a template to an existing page. An existing page already uses a specific layout and each template creates its own layout containers. Technically, if you hand-edit the XML page file, you can assign the template to that page. But you will have to manually move all of the page elements into the new template containers. In most cases, it is probably easier to start from the beginning and build a new page based on the template, then drag the desired data controls onto the new page.



Activity: Create a Skin and Style Sheet

Applications need a consistent look and feel. In particular, it is important that fonts and colors be consistent across pages. Each element on a page can have its own style, and these styles should be defined in a style sheet. Oracle ADF enables you to define a style sheet and ap-

Action

Create a new Style Sheet: Powder.css.

ViewController right-click: New, Web Tier/HTML: CSS.

Create styles for: PageTitle, TitleBackground, MenuBackground, MessageText, MessageBackground.

Define the new skin by creating an XML file:

WEB-INF, right-click, New, General, XML: XML Document Name: trinidad-skins.xml

Edit the trinidad-config.xml file to tell it to use the new skin <skin-family>.

Main menu: Tools/Preferences/CSS Editor: CSS Level 3.

ply it to the entire application—by defining a skin. With this approach, you do not need to assign the style sheet to individual pages—you assign it in one location—the skin definition. The styles within that skin are then automatically available to every page in the application.

Ultimately, each project or application should consult with a graphics designer to help define colors, fonts, backgrounds, and other styles that are appealing and useful. Just remember that design is art, so design goes through trends and fads. Web sites need to be aware of these trends, and each company needs to determine its identity and decide how conservative or radical it wants to be in its presentations. Large companies have in-house designers to set standards and help with individual projects. For smaller applications, you are more likely to hire a consultant.

Not counting the overall layout of the page (something handled by templates in the next section), the design is controlled by identifying primary elements on the page and assigning styles to them. A style is just a named collection of attributes, including typeface, color, background color, borders, and so on. Web pages support hundreds of style attributes, but typically, each element needs only a handful of changes. A style sheet is a file that contains the name and attribute settings for every style needed. The standard CSS page has a specific syntax so it must be created carefully. Some Web browsers are finicky about the CSS and even small errors can cause major page-display errors. Some design tools are better than others at creating CSS pages. Oracle's JDeveloper is on the lower end of acceptability. However, the CSS page is most commonly built by graphics designers who will use other tools. JDeveloper can be used for small files or to make minor changes.

If you use JDeveloper to create the CSS file, right-click the ViewController and pick New. In the Web Tier, choose HTML and then CSS. Figure 8.2 shows that you can create a style by entering a name preceded by a dot and followed by opening and closing braces.

Figure 8.2

```
.PageTitle {
  color : #000060;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 200%;
}
.TitleBackground {
  background: #D0F0F0;
}
.MenuBackground {
  background: #FFFFFF ;
}
.MessageText {
  color: red;
  font-family: Arial, Helvetica, sans-serif;
}
.MessageBackground {
  background: white;
}
.MainBackground {
  background: white;
}
```



```
<?xml version="1.0" encoding="windows-1252" ?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
<skin>
  <id>APSkin.desktop</id>
  <family>APSkin</family>
  <extends>blafplus-rich.desktop</extends>
  <style-sheet-name>css/Powder.css</style-sheet-name>
</skin>
</skins>
```

Figure 8.3

JDeveloper provides prompts to help you select attributes and values; however, you should stick to relatively basic attributes unless you understand the details of style sheets. For a start, you should create entries for PageTitle, TitleBackground, MenuBackground, MessageBackground, MessageText, and MessageBackground. The page title will appear at the top of the page, so it should have larger font, perhaps in a different color. The message text will be used at the bottom of the page—primarily to display status or error messages. The main background and probably the message backgrounds should start as white. The title background can be a contrasting color at the top of the page. The power of using the style sheet is that it is easy to change later, so it is not critical what values you choose now—as long as you choose text colors that are visible against the backgrounds.

The next step is to define the Powder.css file as the main component of a new skin. This process is handled by creating a new XML file following the “Trinidad” specification within Java. First create a blank XML file. Right-click the WEB-INF node and choose New. In the Web Tier, select HTML, then XML Document. The file must be named: trinidad-skins.xml (all lower-case letters).

Figure 8.3 shows the required contents of the skin file. The tags are the required part—you can choose values within the tags. “APSkin” is a reasonable name because it defines the main skin for All Powder. The data for the <extends> tag is important—Oracle ADF includes only a few base skins and the blafplus skin has the most detailed specifications. Also, note that when the style sheet is specified it needs to include the css folder: css/Powder.css because JDeveloper automatically puts style sheets in this separate folder.

After you have defined the new skin using the XML file, you need to tell the project to use that file. This assignment is made in the existing trinidad-config.xml file. As shown in Figure 8.4, edit that file to change the value for the <skin-family> tag to the <family> name you created when you defined the skin: APSkin.

These are the only changes you need to define the skin. All of the values in the Powder.css file will now be available to any page within the project. However, Oracle experts recommend one change that should be made to the overall settings of the project. The skin definitions are designed to work with Level 3 style sheets,

Figure 8.4

```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <skin-family>APSkin</skin-family>
</trinidad-config>
```

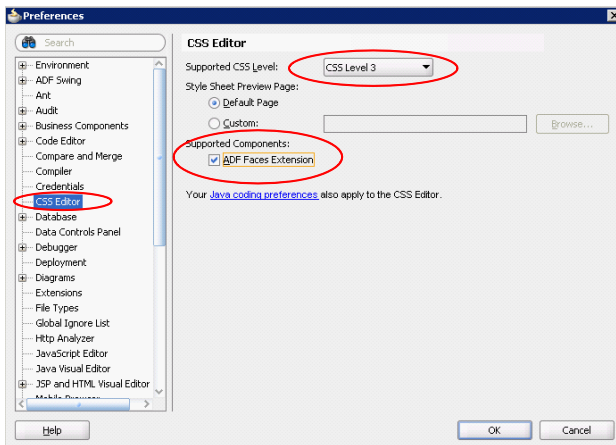


Figure 8.5

but JDeveloper defaults to Level 2. On the main menu, choose Tools/Preferences. As shown in Figure 8.5, choose the CSS Editor option and change the level to CSS Level 3. Also select the ADF Faces Extensions by checking the indicated box. Technically, most browsers use Level 2 style sheets, so the Oracle ADF actually converts the style sheets you create back into Level 2 sheets. However, the Level 3 designation works better within the ADF framework.

Pages should have common elements, such as a title, menu, a message area, and a standard location for the data elements. Many applications also include a company logo and possibly text recommended by the legal department. All of these elements should appear in consistent colors and locations on the form. When you built forms in Chapter 6, you probably spent most of your time just getting the data elements to work the way wanted. Now, you have to pay attention to colors and formats. Actually, you would normally create a standard skin and template before you build the first forms. Later, the styles can be changed by editing the single Powder.css file.



Activity: Create a Template for Pages

A template is created as a jsp page. It defines the overall structure of the page and can include fixed elements that will be displayed on every page that uses the template. These fixed elements can be edited only on the template page. Typically, templates include logos and basic company information or links that will appear on every page. You build a template page almost the same as you would build a regular jsp page. The main difference is that you must include “facet refs” on the template.

These facets are areas that will be available to the final page. When a new page is based on a template, the new page picks up all of the fixed elements and the facet refs. Developers can add new items only to the defined areas indicated by the facet refs.

Action

Create a new template.

Right-click ViewController, New, Web Tier, JSF: JSF Page Template.

Name: MainTemplate.

Check option for Create Associated ADFm Page Definition.

Uncheck option for Quick Start Layout.

Define five facets: TitleArea, ButtonArea, MenuArea, MainArea, and MessageArea.

It is relatively easy to create a design template. Basically, you just create a blank page and add the items that will be displayed on every page. Figure 8.6 shows the primary elements that you will usually include on any form. Create a new template by right-clicking the ViewController project and selecting Web Tier/JSF: JSF Page Template. Templates are unique to each project and rarely shared. You can create multiple templates. For instance, reports might use a different template than input forms. For now, just create one template and name it: MainTemplate. Check the box to: Create Associated ADFm Page Definition—eventually it can be used to store data. If necessary, uncheck the option to use a Quick Start Layout.

Action

Drag a Panel Stretch Layout control onto the blank MainTemplate.jspx.

Place a Panel Group Layout into the top element and set its Appearance/Layout to horizontal.

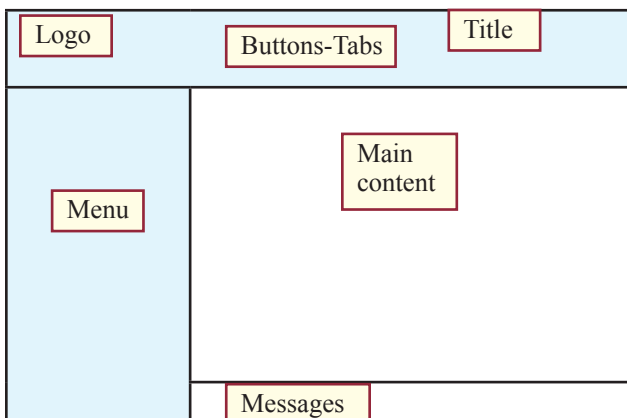
Place an Image Control in the left of this group and locate an icon file.

Add a new Panel Group Layout just to the right of the image and set its layout to vertical.

The main step in creating a template is to define areas that will be used to hold content. Generally, a page can be defined into various sections. At this point, you simply need to assign names to these facets. Figure 8.7 shows the definitions for five facets: TitleArea, ButtonArea, MenuArea, MainArea, and MessageArea. You can choose any number of predefined facets and give them any name that will be understood by developers. The four main areas here are relatively standard: Titles at the top of the page, Menus (probably at the side), Messages at the bottom of the page, and the Main area in the center of the page. The Button area is somewhat experimental to hold options that are not part of the main menu. A page does not have to use every facet, so it is acceptable to define facets that are used only on some pages.

After the definition is entered, the wizard will create a new blank MainTemplate.jspx page. To begin, drag a Panel Stretch Layout control onto the page to provide the overall structure. Drop a Panel Group Layout the top and set its Properties/Appearance: Layout to horizontal. The top of the page will hold a logo and the title area. First you need to locate or create an image file that you can use as a logo. Size of the image can be a problem. You need to consider the desired height of the top line as well as the width of the logo. Ultimately, you need to make guesses about the final size of the page. Many developers assume brows-

Figure 8.6



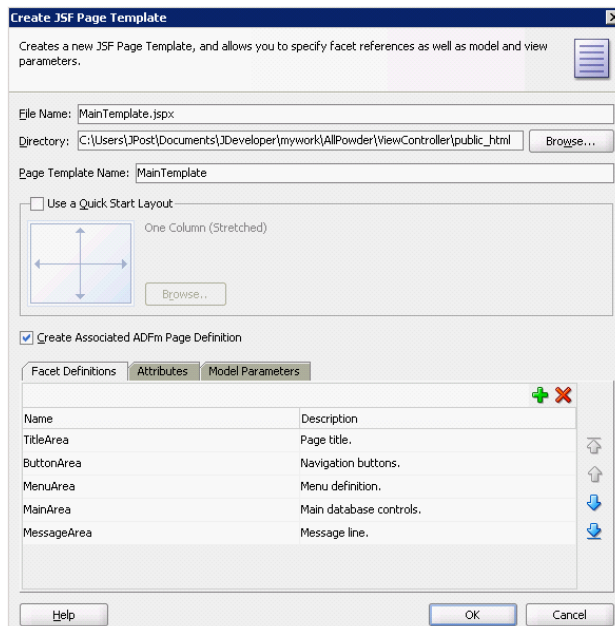
ers can display approximately 1024 by 768 pixels, without too much scrolling. But, as mobile devices become popular, you might have to consider browsers with smaller dimensions. You can create a logo using images from Microsoft PowerPoint. Just be sure to rescale the image to about 70 vertical pixels and save the image as a Web file (png, jpg, or gif). On the project server, create a new images folder under the public_html folder and place your logo file in that folder. In JDeveloper, drag an Image control onto the top-left corner of the page and select the new image file. If necessary, adjust the height of the top section to display the entire logo.

Now you need a panel to hold the title area and the button area, so drag a new Panel Group Layout and place it to the right of the image—within the original horizontal group. You might need to edit the source or use the structure window to get the correct position. Set its Appearance/Layout to vertical. This layout will hold the TitleArea and ButtonArea facets. From the Component window, under Common Components, drag a Facet Ref and place it in the new vertical panel group layout. Again, this step might be easier using the source view. When the Facet Ref is selected, assign it to the TitleArea facet. Repeat the process and place a Facet Ref for the ButtonArea just below the first one.

Action

- Drop a Facet ref into the new vertical layout and assign it: TitleArea.
- Drop a second Facet ref below that one and assign it to: ButtonArea.
- Drop a third Facet ref in the start area for the MenuArea.
- Drop a fourth Facet ref in the center for the MainArea.
- Drop a fifth Facet ref on the bottom for the MessageArea.
- Delete the “end” facet on the right side of the page.
- Assign styles. Main Panel Stretch Layout: MainBackground.
- Top main horizontal Panel Group Layout: TitleBackground.

Figure 8.7

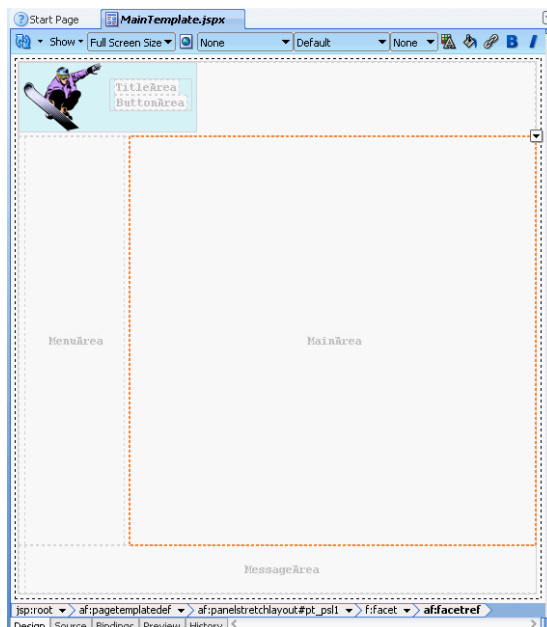


Use a similar process to assign the other three facets. Set the left side (start) to be the MenuArea, the center to be the MainArea, and the bottom to be the MessageArea. You can delete the “end” area on the right side of the page. Remember, developers will only be able to use areas that are defined as facets within the template. Figure 8.8 shows the end result. Also, keep in mind that the underlying stretch panel will resize each section to fit the data and controls when the page is run. For example, you do not need to set the width of the title area. In fact, if you do set the width of a section within the template—that width becomes fixed for all pages that use the template. Eventually, you will probably have to establish a width for the menu area, but remember that it can be done later. Because menus are covered in a later section of this chapter, the details will be handled then.

You do need to set the styles for the main and title areas. Note that styles cannot be applied to facets—because they are simply placeholders. You could add panel layouts to each section before adding the template facets and then assign styles to those panels. For now, assign the MainBackground style to the overall stretch panel. It is easiest to select the stretch panel in the structure window and then assign the style in the property window. Similarly, assign the TitleBackground style to the main horizontal panel group in the top section. Save everything and close the template.

The new template with its associated skin styles can be used as the foundation for all of the pages. However, you will probably have to rebuild all of the pages. To understand the process, create a new page to edit employee data. If you are anxious to see how the template will work, you can start with the JSPX page. Follow the standard process to create a new JSPX page, with a filename of EditEmployee.jspx. As shown in Figure 8.9, simply select the MainTemplate as the layout option instead of one of the standard layout choices. Be sure to set the backing page in case you need code to handle page events.

Figure 8.8



You can add basic elements to the page, but do not add the data until you create the Employee data objects and LOVs. First, drag an Output Text box to the TitleArea. Enter: Edit Employee as its value and assign the PageTitle as the style. Similarly, drag an Output Text to the MessageArea, change its value to blank, and set its style to MessageText. Run the page to see how it looks. Figure 8.10 shows a sample that uses standard LOVs for selecting the department and the manager. However, the Insert button will not work because of the link to the Manager LOV. As usual, the solution is to write your own Insert function in PL/SQL, add the code to the DBModule, and put custom button code on the page.

Action

Create a new EditEmployee.jspx page using the MainTemplate.

Drag an Output Text control into the TitleArea.

Value: Edit Employee, Style: PageTitle.

Drag an Output Text control into the MessageArea.

Value: "", Style: MessageText

Run the page to test it.

Edit the Powder.css TitleBackground to add: min-width: 100%;

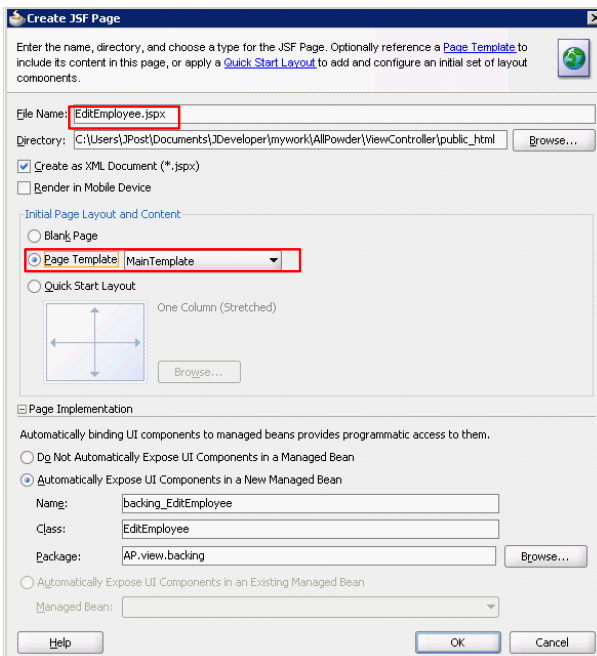
Create the Employee data entity and an LOV for Manager.

Add buttons for Commit.

Write PL/SQL and Module code to add an Insert Employee button.

Follow the same process with every page to ensure that they have the same layout and colors. Remember to assign the styles to the various items and avoid setting properties directly on the page. If you need to make changes, edit the underlying Powder.css style sheet or the MainTemplate.jspx page. All pages using this template will be updated when they are displayed.

Figure 8.9



The screenshot shows a web browser window with the URL `http://127.0.0.1:7101/AllPowder-ViewController-context-root/faces/EditEm...`. The page title is "Edit Employee". The form contains the following data:

Employeeid	14
Firstname	Arno
Lastname	Adam
Taxpayerid	224-10-2726
Phone	837-337-3861
Department	Repair
Address	736 Main
City	Valdez
State	AK
Zip	89286
Managerid	Street, Picabo 208-222-3333

Navigation buttons: First, Previous, Next, Last, Submit, CreateInsert, Commit.

Figure 8.10



Activity: Create the Home Page and Task Flows

After you have created the forms and reports, you need to combine them into an application. Particularly on the Web, a startup or home page is a key element of an application. It is a page that forms the starting point and contains links to the other forms and reports. Generally, it is easy to create—the challenge lies in determining how to organize all of the forms and reports. You have to create a structure, beginning with the home page that guides them through their tasks. This process will often include links on other forms as well. You will have to test this sequence with the users to make sure that it matches their job workflow.

Action

Create a new JSPX page: `home.jspx` based on the `MainTemplate` and including the backing bean.

Add a title and place a form and simple text on the page: Welcome to the shop!

On the main menu: Run / Choose Active Run Configuration / Manage Run Configurations...

For the Default Run Target, click Browse, search for the `home.jspx` page (up a level in the folder then `public_html`).

Connecting Pages with Task Flows

Oracle handles connections between pages as task flows. Every ADF application contains the base unbounded task flow which enables you to connect pages by defining control flows and assigning names to them. Clicking buttons or links triggers specific pages by returning the appropriate name. You can even define the task flows using a graphical editor. Oracle also supports bounded task flows—which are encapsulated tasks that might be started from various points—essentially subroutines of a task that consist of their own pages and control flows. This chapter uses only a few basic task flows, but once you understand how to cre-

ate and use task flows, you can easily create more complex patterns. Figure 8.11 shows an example of the initial home page. Eventually, it will have more complex menus and more content, but for now you want to focus on the layout.

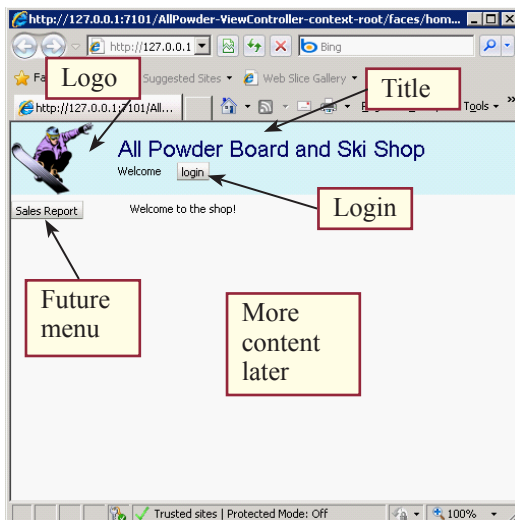
To create a home page, begin with a new blank page: `home.jspx`. You do not need to create data views. Add the title to the page. Eventually, you will want to include a login button on the page—which means it needs to have a form background. So, drag a form panel onto the main area of the page, then add an output text item so you can display a simple text message (Welcome to the shop!). More content can be added later, but you need something in place now so you can see what the page will look like.

This new home page needs to be defined as the starting point for the application. This process is handled through the main menu. Select Run / Choose Active Run Configuration / Manage Run Configurations from the main menu. If necessary, select the Default option and click the Edit button. On the entry line for the Default Run Target, click the Browse button. You need to select the `home.jspx` file you created, but you will probably have to search for it. Typically, you move up one level in the displayed folder list, then open the `public_html` folder to find and select the file. If you click the Run option on the main menu, it should now open to the `home.jspx` file. While testing, you can still start other pages first by opening them and choosing the run option in the right-click menu.

The next step is to begin building the task flow diagram that will form the foundation for the application. An ADF application already contains a default but basically empty task flow container. In the ViewController project under the WEB-INF folder, you should find and open the `adf-config.xml` file. If necessary, click the Diagram tab at the bottom to switch to the graphical interface.

From the main Application Navigator window, drag the `home.jspx` page onto the diagram to create a new view item. You can also create view items before you create the underlying page. Find and expand the Components section of the Component Palette (top right side). Drag a View item and drop it onto the diagram near the home entry. Enter the name: `login`. If you already had a `login.jspx` page, you could drag and drop it onto this view object to establish a connection. But wait a minute before creating the underlying page.

Figure 8.11



Ultimately, the home and login pages will work together. A user should start at the home page—perhaps exploring anonymous pages for a while. The user can then click a login button on the home page to open the login form and enter credentials to register with the system. After a successful login, the user should be returned to the home page. This situation requires two control flows: (1) from home to login, and (2) back to home from the login page.

As shown in Figure 8.12, click the Control Flow Case (green arrow) in the Components palette. Then click the home object on the diagram to set the starting point, followed by clicking the login object to set the target. Enter the identifier as: toLogin. Repeat the process but build a flow from the login page to the home page and name it: toHome. The value of the identifier is completely arbitrary, but you should choose something that indicates the purpose or direction because you will need to use the value later. Believe it or not, you have completed the major step in connecting the two pages. You still have to edit the two pages and create a trigger event (such as a button click) that returns the matching identifier (toLogin or toHome), but that step is straightforward.

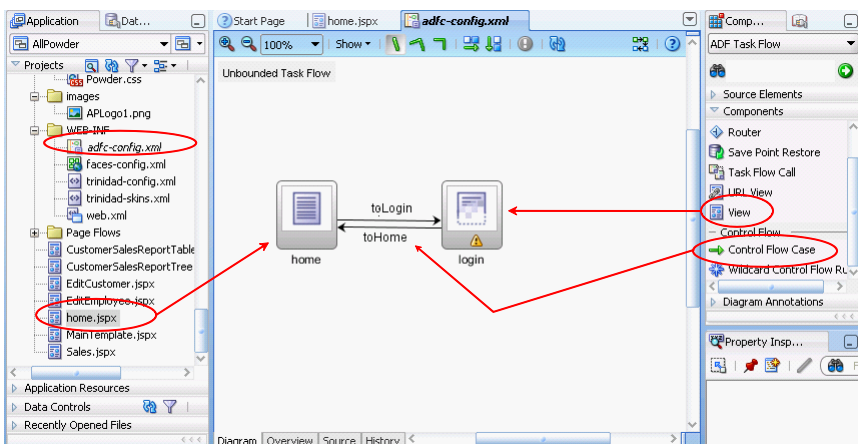
However, there is one critical catch with the current version of JDeveloper. For some reason (bug?) changes made to `adfc-config.xml` are only picked up when JDeveloper restarts. Save all of your work and completely close JDeveloper. Then restart JDeveloper. The `adfc-config.xml` file should still be open, but open it again if it is closed. Remember this step later. It is a pain, but it is the only way I have found to get the task flows to work.

As shown in Figure 8.13, now you need to create the login.jspx page. Double-click the login view icon in the task flow diagram and it will automatically start the JSF page wizard. As usual, add a title and the Message text box. The main section needs a form, so drag a Panel Form Layout control onto the main page. Add

Action

- Open `adfc-config.xml`, click the Diagram tab.
- Drag the `home.jspx` file onto the diagram.
- From component palette drag a View onto the diagram, name it: login.
- In the component palette, click the Control Flow Case object.
- In the main diagram, click the home object followed by login object.
- Name the flow: toLogin.
- Repeat the process to create a connection from login to home: toHome.
- Save everything, close JDeveloper and restart it.

Figure 8.12



input text boxes labeled Last Name and Phone. The database does not yet contain values for username and password, so these items are not very secure. You could use the e-mail address as the username, but the password would need to be encrypted, which makes the process more complicated. It is better to add this feature later after you learn the basic steps. Oracle has other built-in features for handling security and logins but this simple form is useful for understanding the basic steps. You might want to add spacers to make the page easier to read. Also, you can adjust the properties of the text boxes to make them required (under the Behavior section).

To reduce problems with people spying to read passwords, you can set the Secret property to true for the phone number—so that the browser displays only dots or asterisks when the data is entered.

The login button is not finished, but it requires some coding, so put it off for a minute. Save everything to be safe and return to the home.jsp page design. Add a Panel Group Layout to the Button Area of the home page and set its layout to horizontal. Drag an Output Text item into that group, then add a Spacer and a Button next to it.

The Output text item is going to display the name of the person who is logged in. Think about this statement for a minute. To accomplish this task, the Output control on the home.jsp page needs to pick up the value entered into a text box on the login.jsp page. In other words, you need to transfer data from one page to another. Oracle has several ways to transfer data within an application, but when the display pages are created with a backing page to hold Java code, it is rela-

Action

Double-click the login icon to create a new login.jsp page.

Add the title and Message box.

Add a Panel Form Layout to the main section.

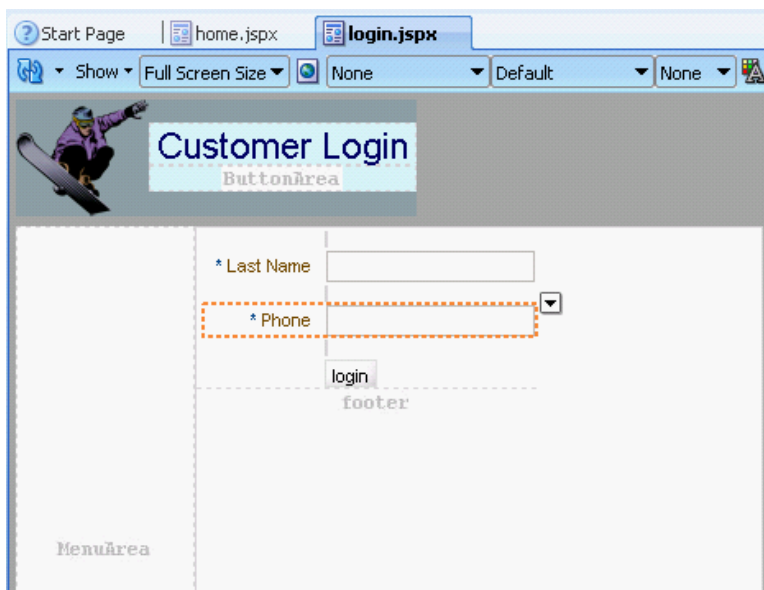
Add input text boxes for Last Name and Phone.

Set Behavior/Required to true for both and Secret for the Phone box.

Add a command button named login.

Save everything even though the login button is not finished.

Figure 8.13



tively easy to pick up data items across pages. Select the Output Text object on the page and find the Value property. Click the drop-down indicator at the right-side of the Value property box and select the Expression Builder. Figure 8.14 shows the edit window for the builder. Type the word Welcome and leave a space at the end of it. Expand the tree for ADF Managed Beans, then the backingBeanScope entry, followed by the backing_login node which represents the code for the login page. The it1 entry is the id value for the Name input text box. You could have set a more descriptive ID value for the box when you created it, but it is too late now. Something to remember for the future, but most of the Oracle examples stick with the default ID values, so get used to them. Scroll down the alphabetical list until you find the value entry and select it. Selecting it will enter the appropriate formula in the Value window:

```
Welcome #{backingBeanScope.backing_login.it1.value}
```

Close the edit window and select the login button. Change the text for the button to: login. The real key is the next step. Select the Action property. Use its drop-down selector on the right-side of the box to see your choices. Select the toLogin entry. This value was entered from the adfc task flow diagram. If you do not see this entry or something similar, you probably need to close and restart JDeveloper. With this selection, the button is now tied to the login page. Actually, you could change what the button does simply by editing the adfc diagram and connecting the task flow to a different page. To test the button and the task flow, save your work and run the home.jspx page. Click the login button and the browser should

Action

Add a horizontal group layout to the Button Area of home.jspx.

Add an Output Text box and use Expression Builder to set its value to:

Welcome Welcome

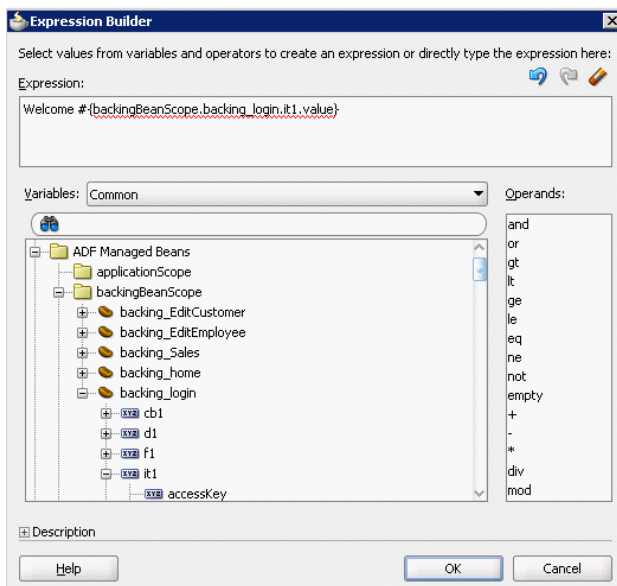
```
#{backingBeanScope.backing_login.it1.value}
```

Add a spacer and a login button after the output box.

Set the login Button Action to: toLogin using the selection choices.

Save everything, run the home.jspx page and test the login button.

Figure 8.14



open the login form. The login form does not work yet because you need to write some code to compare the values entered by the user with those in the database.

Testing Login Credentials

Now that you understand the concept of task flows, you can write the code needed to test the login credentials. The overall structure is to write a PL/SQL function that takes the incoming Lastname and Phone values and runs a query to see if they match a person in the database. If so, the routine returns the CustomerID value so it can be used on other pages. If not, the routine returns a negative number to indicate no match. Later, if you add encrypted passwords, the same PL/SQL routine can be used to handle encryption to do the comparison at the database level. As before, you need an application module to work as an intermediary between the page code and the database. You could use the existing DBModule or even the AppModule, but it is convenient to store the login code in a separate module so it is easier to find (and secure) later. Using the intermediate module also provides the ability to count the number of failed logins—so you can issue a warning message, trigger alters, or even block attackers who try to guess passwords.

Begin by creating the PL/SQL function. Figure 8.15 shows that without the need to worry about encryption, the function is straightforward. The function has two input variables (sLastname and sPhone) which will come from the login page. The basic query checks the Customer table for a row that matches that phone number and name. If a match is found, the CustomerID is transferred to the temporary nID variable which is then returned to the caller. If errors occur or no match is found, a value of -1 is returned to indicate an error. Note that this simple query does not handle cases where two people might have the same last name and phone number. You could handle this case by adding a WHERE rownum<2 condition, which will restrict the values returned to just the first person. In the end, converting to a username/password will solve the problem if the application enforces the condition that usernames must be unique.

The second step is to create the LoginModule to serve as the intermediary. Recall that application modules are created in the Model section of the application, and be sure to extend the underlying AP.model.AppModule module so the new one will have direct access to the database. Also remember to check the option to create the Application Module class. Once the module is created, open the Log-

Figure 8.15

```
CREATE OR REPLACE FUNCTION TestCustomerLogin(
    sLastname NVARCHAR2, sPhone NVARCHAR2)
    RETURN NUMBER
AS
nID integer;
BEGIN
    nID := -1;
    SELECT CustomerID INTO nID
    FROM CUSTOMER
    WHERE Lastname=sLastName AND Phone=sPhone;
    return nID;
EXCEPTION WHEN others THEN
    return -1;
END TestCustomerLogin;
```

```

int loginAttempts = 0;
Number nCustomerID=-1;
public Number TestCustomerLogin(String lastname, String phone) {
    loginAttempts++;
    Number result =
        (Number)callStoredFunction(NUMBER, " TestCustomerLogin(?,?)",
            new Object[] { lastname, phone });
    nCustomerID=result;
    if (result.longValue() > 0) {
        loginAttempts = 0;
    } else {
        if (loginAttempts < 3) {
            result = 0;
        }
    }
    return result;
}
public Number getCustomerID() {
    return nCustomerID;
}
}

```

Figure 8.16

inModuleImpl.java file to edit the code. You still need the Oracle helper code to connect to the database, so open the DBModuleImpl.java file and copy the helper code with its definitions and paste it into the new module. You might need to bring along the import statements to ensure the components are registered correctly. Close the DBModuleImpl file to avoid accidental changes.

Figure 8.16 shows the function needed to connect to the database. The figure does not include the helper functions, but it does include a second function that might come in handy later: `getCustomerID`. The function stores the `CustomerID` internally and the second function makes the value available to other pages that might need it for later tests to see if a customer is logged in. The code in the main function basically just passes the input values to the PL/SQL code and returns the result. However, the function also counts the number of times the current person has tried and failed to log in. If the attempt is successful, the function returns the `CustomerID` value (a positive number). If the attempt fails, the routine counts the number of previous attempts. If the count is less than three (an arbitrary number), then the function returns a zero; otherwise it returns a value of negative one. As you will see in a minute, the calling page can check these return values and issue different messages—or take different actions if the user is trying to attack the system by guessing at the credentials.

Recall that these new functions will be visible to other pages only if they are published. Open the `LoginModule.xml` file and click the Java tab. Click the pencil icon to edit the Client Interface. Move both of the new functions to the right-side list of selected items. Save everything and return to the Java

Action

Write PL/SQL code to search Person table for Lastname and Phone.

Add a new Application Module: LoginModule.

Write module code to connect to the database TestLogin function.

Write the login.java code to call to the TestLogin function.

code page. Right-click to run the Make option to compile the module to check for typos then save and close it.

Finally, you can write the code on the login page to pass the values from the input text boxes to the intermediate module and test the login with the database code. Open the login.jspx page and double-click the login button. Accept the defaults to open the login.java page. Figure 8.17 shows the code. You should be able to copy-and-paste this code into your application. However, be careful with the quotation marks—they often need to be replaced with straight quote marks. The first half of the code is similar to what was used earlier—pick up the input values from the text boxes and store them as parameters. Verify that the ID values are correct for the text boxes on your page. The conditional test (if...) checks the return value and chooses one of two messages depending on the number of failed attempts. Failures also clear the result value to null. Notice at the top of the code that the result value is set to “toHome.” This is the identifier you created on the

Figure 8.17

```
public String cb1_action() {
    // Add event code here... Login button
    String result = "toHome";

    // compare in database
    BindingContainer bindings
        = BindingContext.getCurrent().getCurrentBindingsEntry();
    OperationBinding operationBinding
        = bindings.getOperationBinding("TestCustomerLogin");

    String sLastname = getIt1().getValue().toString();
    String sPhone = getIt2().getValue().toString();
    Map paramsMap = operationBinding.getParamsMap();
    paramsMap.put("lastname", sLastname);
    paramsMap.put("phone", sPhone);
    Object oID = operationBinding.execute();
    long nID = Long.parseLong(oID.toString());

    String sMessage = "";
    AdfFacesContext afContext = AdfFacesContext.getCurrentInstance();
    afContext.getPageFlowScope().put("CustomerID", nID);
    //System.out.println("## Login: nID: "+nID);

    if (nID <= 0) {
        result = null;
        if (nID < 0) { // have exceeded allowed number of attempts
            sMessage = "Please stop guessing.";
        } else {
            sMessage = "Incorrect login. Please try again.";
        }
        FacesContext messageContext = FacesContext.getCurrentInstance();
        messageContext.addMessage(null, new FacesMessage(sMessage));
        // or just use the output text at the bottom which is faster
    }
    else {
        it2.setValue(null);
    }
    return result;
}
```

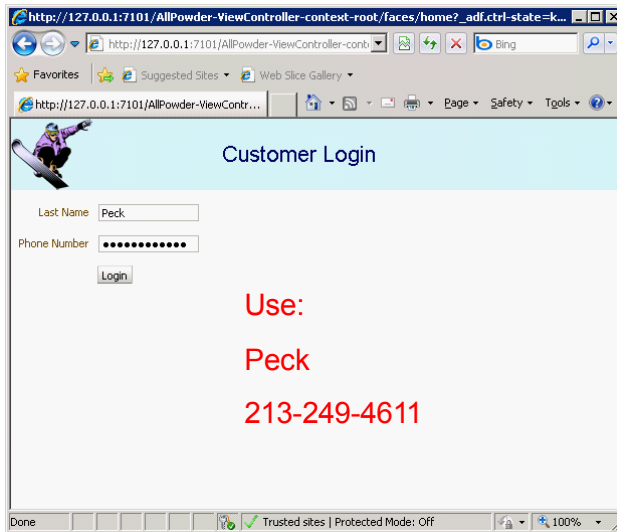


Figure 8.18

task flow diagram. If the test is successful, this value is returned to the calling button, which triggers the task flow to return to the home page. If the test fails, this value is cleared and the browser stays on the login page. If desired, you could set a third value which is returned when the number of attempts exceeds the counter. Then edit the task flow diagram and create a new page with a stronger warning message.

Notice near the end of the code that this routine uses a new method to display messages to the user. The FacesContext tools will display a pop-up message box. You do not have to use this method—it is a little slow. You could simply write the message to the message text box at the bottom of the page. The point is simply to show you the syntax for using the pop-up message box in case you need it for future use.

Two additional lines are a little trickier. The AdfFacesContext lines take the CustomerID value and store it in a new variable on the server in the pageFlow-Scope. This variable is available to any other page in the current page flow. It will be used in the next section to identify the current user. Similar to session scope, variables stored in the page flow scope stay on the server and exist across multiple pages. The main difference with session scope is that you can have multiple page flow scopes at one time—which is particularly useful when you create bounded task flows as subroutines or when users log in multiple times. See the Oracle documentation for more details on page flow scope.

Remember that you must edit the page bindings to link the function call to the intermediate module. Select the Bindings tab on the login.jspx page.

Save everything, compile the page, and run the application starting from the home page. Click the login button to open the login form. Figure 8.18 shows the login form with some sample data you can use to test the login functions. However, first enter some fake data to ensure that the login fails. Then test the real numbers: Peck and 213-249-4611 (with the hyphens). As you will see in a few minutes, this particular person has placed a couple of orders, so the account provides useful data to illustrate the next steps. Once you have logged in successfully, the browser should return to the home page. You should now see the last name (Peck) displayed at the top of the page: Welcome Peck.

A Report for One Customer Using the Login Data

Now that the customer is logged in, your application can use that information (the CustomerID) to present data specifically for that customer. To handle this connection, you have to build reports and pages that specifically link to the stored CustomerID. The process is similar to building any data bound page, with a couple of extra steps. You might want to review the steps in Chapter 6 used to build reports. The goal is to create a Customer Sales report that lists all of the sales for a single customer. The customer can think select one of the sales to see the details. Eventually, you need two report pages: one for the customer sales and one for the sale itself.

Begin by creating a new read-only view in the model section. Figure 8.19 shows that the query just retrieves basic data from the Customer table. However, it has one critical addition: `WHERE CUSTOMER.CUSTOMERID = :customeridVar`. This line uses a bind variable (`customeridVar`) which will be assigned based on values collected from the application. Notice that the colon (`:`) in front of the variable name is critical. Test the query to ensure the syntax is correct.

As you work through the View wizard, you will be given the opportunity to define the bind variable. Figure 8.20 shows the basic form. Click the New button and enter `customeridVar` as the name of the bind variable. This name must match the name of the variable you used in the query. Choose the Number data type to ensure the variable matches the CustomerID data stored in the Customer table. For more complex problems and queries, you can create multiple bind variables. Most of the time, one or two variables are sufficient because you almost always use variables that compare to primary keys. Finish the wizard to complete and save the view object.

The report page needs a detail subsection that lists the all of the sales for that customer. You need to create a new read-only view for that section. Figure 8.21 shows the basic query. By now, you should be comfortable with SQL queries. Because it is a read-only query, it can compute the sum of the items (price times quantity). Just remember to use `GROUP BY` to compute the subtotal for each sale.

Action

- Create a new Customer report where the Customer query uses a bind variable to retrieve data for a single CustomerID.
- Define the bind variable `customeridVar`.
- Define the subform query and build the View Link.
- Create the new `OneCustomerSalesReport` page.
- Drag the `OneSalesTotalsReportView2` object to create a Master-Detail: ADF Master form, Detail table.

Figure 8.19

```
SELECT
  CUSTOMER.CUSTOMERID CUSTOMERID,
  CUSTOMER.LASTNAME LASTNAME,
  CUSTOMER.FIRSTNAME FIRSTNAME,
  CUSTOMER.EMAIL EMAIL,
  CUSTOMER.STATE STATE,
  CUSTOMER.GENDER GENDER
FROM
  CUSTOMER
WHERE CUSTOMER.CUSTOMERID=:customeridVar
ORDER BY CUSTOMER.LASTNAME, CUSTOMER.FIRSTNAME
```

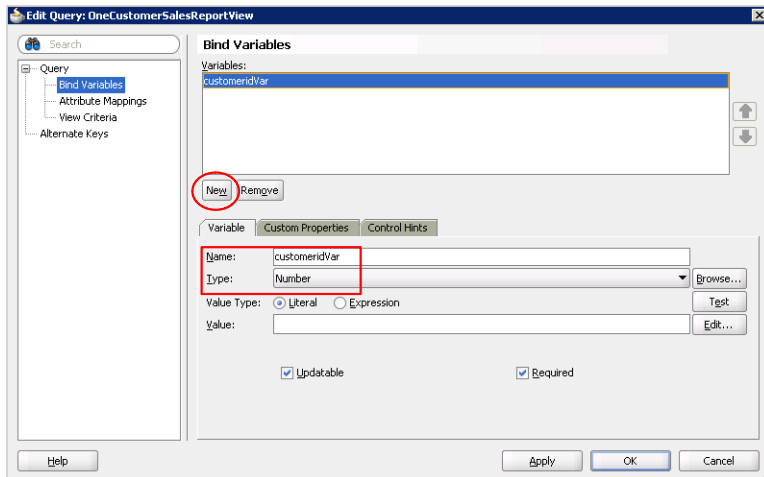



Figure 8.20

This query is going to show each sale and its total. It is not going to display the individual items sold. Those details can be displayed on a second report page that you will create in a few minutes. Name this new view: `OneSalesTotalsReportView` to indicate it will be used as part of the `OneCustomerReport`.

Once the two views have been built you must create a `View Link` to join them. Follow the standard wizard steps to join the two views based on `CustomerID: OneCustomerSalesReportView.Customerid = OneSalesTotalsReportView.Customerid`. With the view link created, it is possible to compute and display the total of all sales in the main `OneCustomer` view. It is not required, but it can be useful to show the total of all of the sales. As you did in Chapter 6, open the `OneCustomerSalesReportView` and add a new attribute. Set its `Expression` to: `OneSalesTotalsReportView.sum("Saletotal")`. Be careful to match the names of the view and the column exactly. One critical warning is needed at this point: Do not define any dependencies for this calculation! There is an extremely strange error in `JDeveloper` (or `WebLogic`) that causes the login process to fail if you define a dependency here. It is strange and hard to find because it is in a completely unrelated page! Perhaps this problem will be fixed in later releases, but you do not need dependencies, so it is not critical—just avoid them to be safe.

Now create the display page: `OneCustomerSalesReport.jspx`. Set the title and the message text box. Refresh the `Data Controls`. Expand the `OneCustomerSalesReportView2` and drag the nested `OneSalesTotalsReportView2` onto the main

Figure 8.21

```
SELECT
  SALE.SALEID SALEID,
  SALE.SALEDATE SALEDATE,
  SALE.CUSTOMERID CUSTOMERID,
  SUM(SALEITEM.QUANTITYSOLD*SALEITEM.SALEPRICE) SaleTotal,
  COUNT(SALE.SALEID) SaleCount
FROM
  SALE INNER JOIN SALEITEM
    ON SALE.SALEID=SALEITEM.SALEID
GROUP BY SALE.SALEID, SALE.SALEDATE, SALE.CUSTOMERID
ORDER BY SALE.SALEDATE
```

section of the page. Choose the Master-Detail: ADF Master form, Detail table option. Clean up the subform.

The next step is to create the page bindings so that the Customer query underlying the report page picks up the CustomerID of the current customer. Remember that the login page stores the CustomerID in a pageScope variable called CustomerID. You simply need to define the page bindings to connect that variable to the underlying customeridVar used in the query. On the OneCustomerSalesReport.jspx page, click the Bindings tab. In the Bindings window (left side of page), click the green plus sign to add a new binding. Choose the “action” method and click the OK button to proceed. Figure 8.22 shows the main steps. The underlying data

view is in the AppModuleDataControl so select it first. In the Iterator selection box, choose OneCustomerSalesReportView2Iterator. This iterator retrieves the data for the top part of the report, which contains the Customer query with the bind variable. For the Operation, select ExecuteWithParams, which opens the parameters window. The first two entries should be populated automatically; customeridVar and the oracle.jbo.domain.Number data type. All you have to enter is

Action

OneCustomerSalesReport page, click the Bindings tab.

In Bindings window click the green plus sign, choose “action.”

Select the main AppModuleDataControl

Select Iterator:

OneCustomerSalesReportView2Iterator.

Select Operation: ExecuteWithParams.

For parameter: customeridVar, oracle.jbo.domain.Number

Value: #{pageFlowScope.CustomerID}

In Executables window click the green plus sign, choose “invokeAction”

id: invokeExecuteWithParams

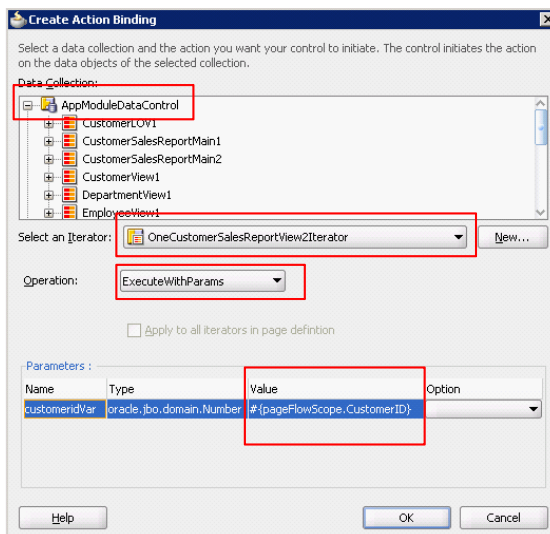
binds: ExecuteWithParams

In the Executables window select the new invokeExecuteWithParams entry and click the edit pencil icon.

Select the Common tab and set Refresh to “ifNeeded” and

RefreshCondition: #{adfFacesContext.postback == false}

Figure 8.22



the value. The value needs to come from the `pageFlowScope` variable created on the login page, so carefully enter:

```
#{pageFlowScope.CustomerID}
```

Choose the buttons to finish the binding entry until you return to the main binding page. This binding entry establishes the connection from the scope variable to the query, but it does not cause the query to actually run. You need to create a new executable function to run the query. In the Executables window (middle of the page), click the green plus sign and choose “invokeAction” as the method. In the pop-up box, set the id to `invokeExecuteWithParams` and choose `ExecuteWithParams` in the binds selection box. Return to the main binding page. This function is used to execute the query and process the binding data, but it does not run automatically. You need to set its refresh parameters to cause it to run whenever the page is loaded the first time.

In the Executables window, select the new `invokeExecuteWithParams` method and click the pencil icon to edit the parameters. As shown in Figure 8.23, in the Refresh selection box choose “ifNeeded.” You then need to type in a special condition for the `RefreshCondition`:

```
#{adfFacesContext.postback == false}
```

Be sure to enter the equals sign twice—which the comparator operator in Java. This context causes the page to requery the first time it is processed (when post

Action

Open `adfc—config.xml` diagram.

Drag `OneCustomerSaleReport` onto the diagram.

Add a Control Task Flow from home to the new report, name it: `toSalesReport`.

Add a return control flow named: `back`.

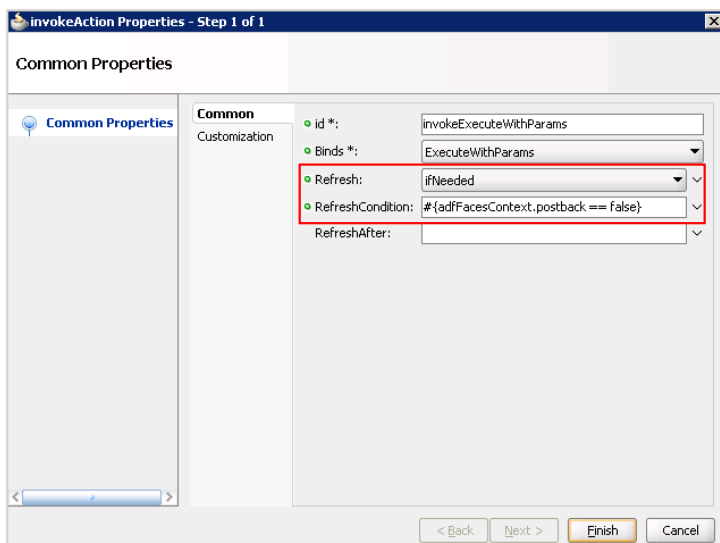
In `home.jsp`, add a Panel Group Layout set to vertical and place a command button in it.

Set the button label to: `Sales Report` and select its action as: `toSalesReport`.

Add a button to the `Sales Report`, label it `Back` and set its Action to `back`.

Test the login and open the `Sales report` for Peck.

Figure 8.23



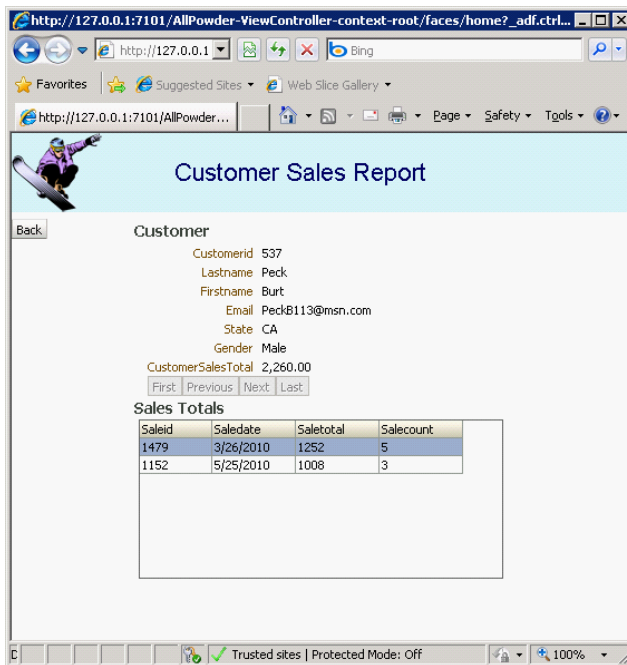


Figure 8.24

back is false). But if you put buttons on the page to process data the page will not be requeried.

The main pages have now been created. The next step is to create a new Task Flow to connect them. Open the `adfc-config.xml` diagram and drag the new `OneCustomerSaleReport` onto the page. Add a Control Task Flow from home to the new report and name it: `toSalesReport`. Add a return control flow named: `back`. Now all you need is a couple of buttons on the pages. However, you will probably have to close JDeveloper and restart it to force the changes to be available.

Open the `home.jspx` page and add a Panel Group Layout to the menu area. Set its layout to vertical and Valign property to top. Place a command button in the new group and label it Sales Report. In the Action property select the `toSalesReport` action. Repeat the process on the new `OneCustomerSalesReport` page and set the button label to Back. Choose back for its Action so that it will return to the home page.

Finally you are ready to save everything and test the form. Save everything and choose the Make or Rebuild option to check for syntax errors. When everything is correct, run the home page. Login and click the button to open the new Sales Report. As shown in Figure 8.24, it should show customer data for Peck and list his corresponding sales.

Connect Table Row to Detail Report

You are making great progress and the project is beginning to look like a useful application. Looking at the last report, one more commonly-used concept comes to mind. It would be nice to include a button on the Sale detail table so the customer can click one button and see the details for the selected sale. Oracle ADF has a process for building this type of form and action without using code. You can find the detailed steps in some of the documentation. However, it is worth-

while to write the code yourself so you can see the underlying process. It is rare that you will be able to use the Oracle process—most of the time you will need to edit the underlying code or make special tweaks, so you should know how to do it yourself.

The basic process is to add a column to the detail table on the OneCustomerSalesReport page. Write a short function that takes the SaleID key value and puts it into a page scope variable. Then create a new Sale Report query with a bind variable for SaleID. Create a display page for the Sale Report and set the bindings to match the page scope variable to the query bind variable. Add a return button to the report page and you are done. A couple of the steps have some new twists, but the concepts are straightforward.

Begin by adding a new column to the detail table in the OneCustomerSalesReport. Open that page in design view. It seems to be challenging to add the column graphically, so switch to the source tab. It is straightforward to create a new column using the XML for the existing columns as a pattern. Figure 8.25 shows the full XML for the new column which includes a command button. You could use an expression builder to set the Action for the button, but it can be figured out by looking at other examples. The goal is to call a function on the java backing page for the report, so the action is:

```
#{backingBeanScope.backing_OneCustomerSalesReport.showSale}
```

The name of the new function will be: showSale. You can switch to the Design tab on the page to see if the table is displayed correctly. You probably have to reset the width of the table to display the new button column.

The trickiest part of this new process is to write the showSale function. The concept is easy: Get the SaleID value from the currently selected row of the table and store it in a page scope variable. Then return an id value that will be defined as

Action

- Add a column to the OneCustomerSalesReport detail that includes a command button.
- Write code function in OneCustomerSalesReport2.java.
- Create a new Sales view (DisplaySaleReportView) with a bind variable for saleidVar.
- Create the subreport query (DisplaySaleItemsView) and the view link (DisplaySalereportToItemsViewLink).
- Create a new page for DisplaySale.jspx using the Master-Detail form/table for the DisplaySaleItemsView2 object.
- Add the page bindings setting saleidVar to #{pageFlowScope.SaleID}.
- Add the DisplaySale page to the adfc-config diagram.
- Add a Control Flow Case from OneCustomerSalesReport to DisplaySale using: showSale.
- Add a reverse control flow with the keyword: back.
- Add a button to the DisplaySale.jspx page and assign the back Action.
- Save everything and test the reports.

Figure 8.25

```
<af:column headerText="Details"
  sortable="false" id="c6" width="60">
  <af:commandButton id="cbT0" text="Details"
    action="#{backingBeanScope.backing_OneCustomerSalesReport.showSale}"
  />
</af:column>
```

```

public String showSale() {
    RichTable tbl = getT1();
    List<oracle.jbo.Key> kList = (List<oracle.jbo.Key>)tbl.getRowKey();
    // Could be multiple keys, but know there is only one key with a single value
    oracle.jbo.Key key1 = kList.get(0);
    Object oList[] = key1.getAttributeValues();
    long nID=-1;
    try {
        nID = Long.parseLong(oList[0].toString());
    } catch (Exception ex) {
        nID=-1;
    }
    AdfFacesContext afContext = AdfFacesContext.getCurrentInstance();
    afContext.getPageFlowScope().put("SaleID", nID);
    return "showSale";
}

```

Figure 8.26

a task flow to the new Sale Report page. The only part you have not done before is getting the key value from the Sale table. However, there is one slight twist. Remember that the OneCustomer report page was based on the oneCustomerSalesReportView2. Therefore, you need to place the new function in the file: OneCustomerSalesReport2.java. Be sure to open the correct file.

Figure 8.26 has the showSale code—you should be able to copy-and-paste it into your file. Most of the steps are used to extract the key value from the form using the proper transformations. The getT1 function returns the table and its getRowKey function returns the key list for the selected row. Because primary keys might contain multiple columns, the data is returned as a list. In this case, we know that list contains only a single column (SaleID). The code uses the lists to obtain the first (zero) entry. It then explicitly converts the key value to a string and parses it to a long integer. You could have used a simple cast, but this approach makes it easier to test for numeric conversion errors. It will also be easier to edit if Oracle changes the way key values are passed in the future. The rest of the code simply writes the value into a new SaleID variable stored in the page scope. Lastly, notice that the function returns a string value of “showSale.” You could pick any phrase, but remember this value because you will need it when you define the task flow to the new page.

Figure 8.27

```

SELECT Sale.SaleID, SaleDate, Sale.CustomerID,
ShipAddress, ShipCity, ShipState, ShipZIP, SalesTax, PaymentMethod,
Sum(SalePrice*QuantitySold) As SaleTotal,
LastName, FirstName, Phone, Email
FROM Sale
INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
INNER JOIN Customer ON Sale.CustomerID=Customer.CustomerID
WHERE Sale.SaleID=:saleidVar
GROUP BY Sale.SaleID, SaleDate, Sale.CustomerID, ShipAddress, ShipCity, ShipState,
ShipZIP, SalesTax, PaymentMethod, LastName, FirstName, Phone, Email

```

Now you need to create the two views and the view link for the new report. Much like the Sale input form, the goal is to display basic Sale and Customer data on the top of the form along with a detail section listing the Items purchased.

Figure 8.27 shows the query for the read-only view (`DisplaySaleReportView`) used on top-half of the report that combines the Sale data and a couple of columns from the Customer table. Notice that the query also computes the subtotal of price by quantity for that sale. On the original Sale form, this total was computed on the fly using the view link. Here, it is easier to handle on this query because the data cannot change on the report. Be careful to include the clause `WHERE Sale.SaleID = :saleidVar` and remember the leading colon. Then remember to define the bind variable on the appropriate wizard screen using the standard Number data type.

Creating the read-only view (`DisplaySaleItemsView`) for the subreport is easier. As shown in Figure 8.28, it pulls all of the columns from the `SaleItem` table and a couple of display columns from the `Inventory` table. It's a good thing you learned the SQL syntax so you can build all of these queries by hand.

A new view link is needed to associate the detail subsection with the main form. Name it `DisplaySaleReportToItemsViewLink` and set the `SaleID` values from the two queries equal to each other.

By now you should know how to create the new display page (`DisplaySale.jspx`). Set the usual title and message lines. Refresh the data controls. Drag the nested `DisplaySaleItemsView2` object onto the page and choose the Master-Detail, ADF Master form, Detail table option. Clean up the displays and double-check the width and height of the detail table.

Because the main view uses a bind variable in the query, you need to go through the binding steps for the page. Click the Bindings tab and in the Bindings window, click the green plus button. Choose “action,” select the `AppModuleDataControl` and choose the iterator: `DisplaySaleReportView2Iterator`. Choose the `ExecuteWithParams` option. For the `saleidVar`, set its value to `#{pageFlowScope.SaleID}`.

The tricky part is to remember that you also need to create the `invokeAction` in the Executables window. Set the id to `invokeExecuteWithParams` which binds to the `ExecuteWithParams` function. You also have to edit this new `invoke...` method and set the refresh to “ifNeeded,” and the `RefreshCondition` to: `#{adffacesContext.postback == false}`. Save everything.

The last big step is to define the task flow. Open `adfc-config.xml` and drag the new `DisplaySale.jspx` page onto the diagram. Add a Control Flow Case to connect `OneCustomerSalesReport` to this new page using the keyword: `showSale`. Remember that this keyword must match the keyword returned by the `showSale` function. Add a reverse control flow with the keyword: `back`. Again, you probably have to close and restart `JDeveloper` for the changes to take effect—until Oracle manages to fix this problem.

Open the `DisplaySale.jspx` page and select the Design tab. Add a button to return and select the back Action. The button on the `OneCustomer` page has already

Figure 8.28

```
SELECT SaleID, SaleItem.SKU, QuantitySold, SalePrice,
ModellID, ItemSize,
QuantitySold*SalePrice As LineTotal
FROM SaleItem
INNER JOIN Inventory on SaleItem.SKU=Inventory.SKU
```

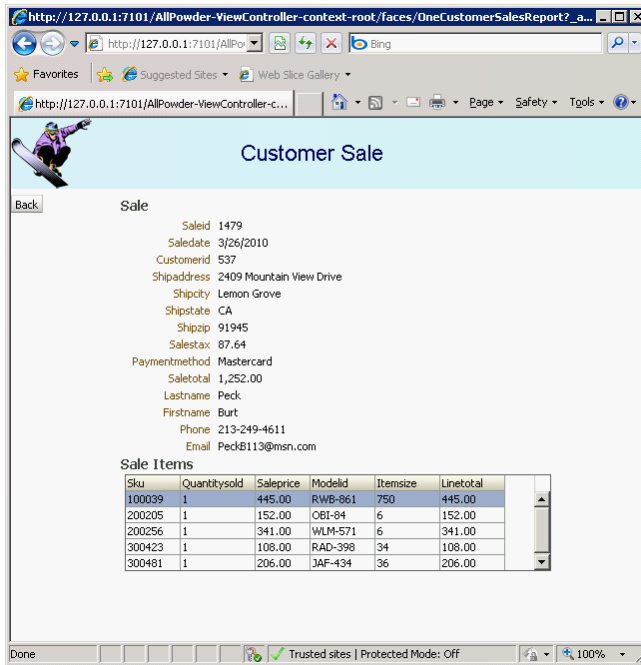


Figure 8.29

been created, so you are done. Save everything and rebuild the project to check for syntax errors. Run the home page, login, open the Customer Sales report and click the Detail buttons to ensure the browser opens the matching Sale report. Figure 8.29 shows an example of the report. This particular customer (Peck) was selected because he has a couple of usable Sales.

This lab section was long and somewhat complicated. But now you know how to connect pages in an application. You learned several ways to share data across multiple pages and how to use variables in queries to show specific data on a page. These are the primary concepts used to build any application. Oracle ADF includes many other tools that can be used to display complex data on a page, such as pivot tables and charts. Many of them are useful at displaying and understanding data. The concepts for building a page and connecting them are the same for all of these controls.



Activity: Build Menus

The home startup form is important for Web sites, but you often need to build menus that make it easy for users to find the other pages in the application. Some application tasks need to follow a specific set of steps. For instance, Web checkout procedures often require users to enter data on several different forms. Control task flows can be used to navigate users automatically through pages. Menus, bread crumbs, and “trains” can be used to show users where they are in the task and provide a means to move back to an earlier step.

Action

- If you have time create more forms.
- Create the Rentals form following the details for the Sales form.
- Create edit forms (tabular) for ProductCategory, SkiBoardStyle, and Binding Style.
- Add the forms to the `adfc-config.xml` diagram. If you lack time, just add View objects.
- Do not add flow connectors.

Oracle ADF provides support for menus, bread crumbs, and train steps. This section focuses on the menus because they form the foundation for the methods. You can find details and examples of bread crumbs and train stops in the Oracle documentation and on the Web. However, the Oracle documentation points out that the current version of ADF has some problems. Notably, the browser “Back” and “Refresh” buttons. Because of the way ADF handles task flows and session state, these browser buttons can cause the application to get out of synchronization with the page. So, you should avoid investing huge amounts of time and energy building a complex application that attempts to force people to follow a fixed sequence. Instead, try to design a flexible system that enables users to choose the way they want to work and the way they want to proceed through the pages.

JDeveloper has some tools that help you create menus. This section will use one of the tools to get started, but it is better if you learn how to do a couple of steps manually. Customizing menus is easier to do by editing the underlying XML files. A menu is simply a list of possible options. Typically, it is displayed at the top of the page, but you can choose where it will be displayed and you can select from a handful of standard formats. Menus can be hierarchical—items in the top level can be selected to display a set of sub-choices. The default options in Oracle menus are basically static—users click each option to open a page—which might contain submenus. If you want dynamic menus with hover-over effects, you can install add-in components, such as Adobe Flex Menu.

The first step in building a menu is to think about the overall structure of the application. It often helps if you build the pages first—or at least design them—so you can see how users might need to navigate through the application. If you have time, you should build the Rentals form and tabular input forms for the Product-Category, SkiBoardStyle, and BindingStyle tables. If you lack the time, you can still build the menus using View placeholder objects; but at least look through the database to see what forms and reports might be needed. Then you can think about the overall structure of a menu. Figure 8.30 shows one possibility. Ultimately, you might want to control the menus programmatically so that users only see the options that they are allowed to use. For instance, anonymous customers would see only a couple of basic choices. After users log in, they can see the Customer link which would let them edit their personal data and see their specific sales. These options can be controlled through the menu item visibility or by dynamically installing different menus for each user, but these details are not covered in this section. For now, focus on the steps needed to design and build the basic menu.

You can use the JDeveloper wizard to create the foundation for the main menu. Open the `afdc-config.xml` file and click the Diagram tab. Add the pages to the diagram that will make up the nodes (points) on the menu. If you have not yet built the jsp pages, drag View objects onto the page and assign the appropriate name. The diagram might become crowded, and it might be easier for you to read

Figure 8.30

Home	Login	Checkout	Customer	Admin	Help
		Sales Rentals		Edit Customer Edit Employee Setup	
				Categories Ski Style Binding Style	

if you organize the pages in groups. For now, it only matters that the pages or views are located in the diagram. When you are ready, save everything and then right-click a blank space on the diagram and choose the Create ADF Menu Model option. Accept the default file name (`root_menu`) which creates a new XML file.

Open the new `root_menu.xml` file to see what you have. It should consist of a single `<menu>` tag entry that contains several `<itemNode>` entries—one item node for each page on the diagram. Because you did not define any hierarchical task flows, all of the item nodes are at the same level. This XML file is the key

Action

Right-click the main diagram and choose: Create ADF Menu Model.

Sketch a potential menu layout.

Edit the `root_menu.xml` file to add `groupNodes`.

Change the label values.

Create three main groups: Checkout, Customer, and Admin.

Create a nested group (Setup) inside the Admin group.

Figure 8.31

```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <itemNode id="itemNode_home" label="home" action="adfMenu_home"
    focusViewId="/home"/>
  <itemNode id="itemNode_login" label="login" action="adfMenu_login"
    focusViewId="/login"/>
  <groupNode id="groupNode_Checkout" label="Checkout" idref="itemNode_Sales">
    <itemNode id="itemNode_Sales" label="Sales" action="adfMenu_Sales"
      focusViewId="/Sales"/>
    <itemNode id="itemNode_Rental" label="Rental" action="adfMenu_Rental"
      focusViewId="/Rental"/>
  </groupNode>
  <groupNode id="groupNode_Customer" label="Customer"
    idref="itemNode_OneCustomerSalesReport">
    <itemNode id="itemNode_OneCustomerSalesReport"
      label="One Customer Sales Report"
      action="adfMenu_OneCustomerSalesReport"
      focusViewId="/OneCustomerSalesReport"/>
  </groupNode>
  <groupNode id="groupNode_Admin" label="Admin" idref="itemNode_EditCustomer">
    <itemNode id="itemNode_EditCustomer" label="Edit Customer"
      action="adfMenu_EditCustomer" focusViewId="/EditCustomer"/>
    <itemNode id="itemNode_EditEmployee" label="Edit Employee"
      action="adfMenu_EditEmployee" focusViewId="/EditEmployee"/>
    <groupNode id="groupNode_Setup" label="Setup" idref="itemNode_EditProductCategory">
      <itemNode id="itemNode_EditProductCategory"
        label="Edit Product Category"
        action="adfMenu_EditProductCategory"
        focusViewId="/EditProductCategory"/>
      <itemNode id="itemNode_EditSkiBoardStyle" label="Edit Ski Board Style"
        action="adfMenu_EditSkiBoardStyle"
        focusViewId="/EditSkiBoardStyle"/>
      <itemNode id="itemNode_EditBindingStyle" label="Edit Binding Style"
        action="adfMenu_EditBindingStyle" focusViewId="/EditBindingStyle"/>
    </groupNode>
  </groupNode>
</menu>
```

to your menus—it defines all of the entries for the menu.

If you were to build a menu with the current model, it would contain every one of the pages on a single menu level. This approach would create a menu that is too long to read, so you need to group some of the pages together. You need to add groups to the XML file by creating `<groupNode ... >` entries. Figure 8.31 shows the XML file with three primary groups added (Checkout, Customer, and Admin) and one secondary grouping (Setup under the Admin group). This XML file will create a menu layout to match the design in Figure 8.30.

A `groupNode` tag needs a couple of parameters:

```
<groupNode id="groupNode_Checkout"
  label="Checkout" idref="itemNode_Sales">
```

The ID value can be almost any unique identifier. The Label entry is the value that will be displayed as the menu entry. Check your XML file and you will see that you need to change all of the labels that were generated automatically. The IDRef parameter must match the ID value of one of the item nodes within the group. It represents the page that will be displayed first when the menu is selected. Sub-level groupings are created by defining a `groupNode` nested within another `groupNode`. In the example, the Setup group node is defined within the Admin group node XML tag. You can use as many subgroup levels as you want—but if you get too many levels, the users will become confused so try to keep it reasonable.

The `root_menu.xml` file is the heart of the menu system. You can edit this file later to add new pages or change the menu structure. The only remaining step is to display the menu. Notice that each menu item contains an action and focus parameter, so the menu system knows what to do when each item is selected.

The best place to display the menu is on the MainTemplate. The template is used by every page in the application, so the menu will be displayed on every

Action

In `MainTemplate.jspx`, add three `navigationPane` tags—one for each menu level.

Place the tags in the top facet beneath the title and button panes.

Set `var="menuInfo" value="#{root_menu}" hint="tabs" level="0"`.

In the `commandNavigationItem` tag, set `text="#{menuInfo.label}" destination="#{menuInfo.destination}" action="#{menuInfo.doAction}"`

For the second and third versions of `navigationPane`, set `level` to 1 and 2 respectively.

For the third version of `navigationPane`, set `hints="list"`

Figure 8.32

```
<af:navigationPane id="np1" var="menuInfo" value="#{root_menu}"
  hint="tabs" level="0">
  <f:facet name="nodeStamp">
  <af:commandNavigationItem text="#{menuInfo.label}"
    id="cni1" icon="#{menuInfo.icon}"
    destination="#{menuInfo.destination}"
    action="#{menuInfo.doAction}"
    visible="#{menuInfo.visible}"
    rendered="#{menuInfo.rendered}" />
  </f:facet>
</af:navigationPane>
```

page—simply by defining it in one location. If you need more dynamic menus, you can create pop-up menus or use code to define buttons and menus on other areas of the page.

Open the `MainTemplate.aspx` page and switch to Source view to edit the XML. Find the top section that contains the title and button areas defined for the template. Just below the button area but still within the vertical group panel layout, add a `navigationPane`. Figure 8.32 shows the tag options for the `navigationPane` control. You can drag the control onto the page or simply type the XML shown in the figure. Note that either way you need to edit the various attributes. Most of these could be entered using the Property viewer, but it is usually easier to simply edit the tag directly. The `var` attribute assigns a name for the binding options. The `value` attribute must be set to match the Menu Model XML file: `#{root_menu}`. The `hint` attribute sets the type of display, you can experiment with the other choices once the menu is running to see which version best fits your design. The `commandNavigationItem` tag needs the most editing. The default attributes use fixed values. You need to change all of them to use binding items that will be extracted from the menu model file. As shown in the figure, the basic format is: `#{menuInfo.label}`, where `menuInfo` is the var name assigned in the navigation pane.

Recall that the current XML menu model has three levels. Therefore, you need to define navigation panes for all three levels. The easy method is to create the first entry and copy-and-paste it to create the base for the other two entries. Figure 8.33 highlights the differences. You need to include the full `commandNavigationItem` tag in each entry, but it is the same in all three cases. The main change you need to make is to ensure each `navigationPane` has a different ID value and that each one is defined for a different level: 0, 1, and 2. The last pane (level 2) also uses a “list” display instead of “tabs” just to show a different layout.

That is all you need to do. Two steps: (1) Create and edit the `root_menu.xml` file and (2) define a `navigationPane` in the `MainTemplate.aspx` page for each of the levels. Although, check the Design layout quickly to ensure that the three menu

Figure 8.33

```
<af:navigationPane id="np1" var="menuInfo" value="#{root_menu}"
  hint="tabs" level="0">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuInfo.label}" ... />
  </f:facet>
</af:navigationPane>
<af:navigationPane id="np2" var="menuInfo" value="#{root_menu}"
  hint="tabs" level="1">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuInfo.label}" id ... />
  </f:facet>
</af:navigationPane>
<af:navigationPane id="np3" var="menuInfo" value="#{root_menu}"
  hint="list" level="2">
  <f:facet name="nodeStamp">
    <af:commandNavigationItem text="#{menuInfo.label}" id ... />
  </f:facet>
</af:navigationPane>
```

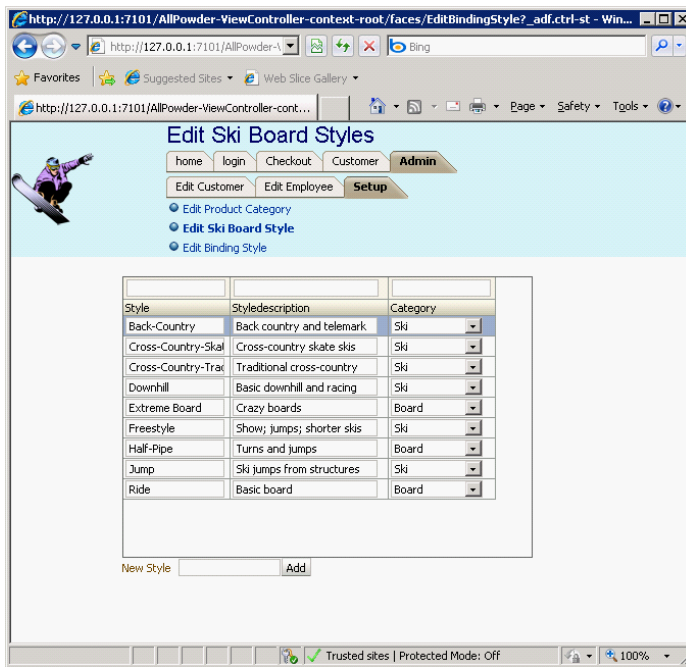


Figure 8.34

levels will fit in the header section. If necessary, drag the boundaries to add more space.

Because the template is already applied to every page, you can simply rebuild and run the application. Each page based on the template will pick up the menu. As you select menu items, the new page will be displayed and the menu will be updated to show the current location. As shown in Figure 8.34, when a group tab (Admin) is selected, the next level menu is displayed as well. The figure shows the Admin and subgroup Setup tab selected, which displays all three menu levels. The user is free to select among the three items in the lowest menu or switch to an completely different section of the menus.

Notice that the menus on the page are static. They do not use hover or pop-up actions and lower-level menus are not displayed until the user selects the group tab. All of the basic Oracle choices work the same way—only the visual display is different. If you want to create other types of menus, you need to install other tools. Fortunately, you can find notes on the Web that can extract your menu model data and feed it to these other menu tools automatically. So you can use this same process to create your menus and then install other tools later if you want to change the appearance and usability.

Menus have additional options that are explained in the Oracle documentation. For instance, you can use resource binding to store the label text in a separate resource file instead of typing the name directly onto the tab. Resource files can be translated into multiple languages and the system can automatically display menu labels in the appropriate browser language. You can also use an ampersand to signify a shortcut key, such as &Home to highlight the H on the label.



Activity: Write Help Files

A finished application also needs customized Help files. Users should be able to select the Help menu option and receive additional information to help them

perform a task or understand the data that needs to be entered. Detailed Help systems can become complex, with large applications requiring hundreds of pages of Help text and instructions. On large projects, companies often hire a special team just to create and edit the Help files. For these situations, you will want to purchase a dedicated Help system editor. However, you can build Oracle Help files with a text editor and a couple of free downloads. Search the Oracle OTN site for the Java help or Oracle Help (OHW) files (<http://www.oracle.com/technology/tech/java/help/index.html>). Figure 8.35 shows the basic steps involved in creating a Help system. First you write individual Help pages as HTML text files. These pages

can have links to each other and to external Web sites. One of the pages should be the startup page. You should also keep a list of keywords and topics for each page so you can create the index and table of contents (TOC) later. You should also create a mapping file that assigns a topic identifier to each page. The Help system file contains links to these other files so the Help system can find everything. Finally, in each Oracle form, you define a custom Help menu item that uses a hypertext command to open the Help topic for that form. The Help form also includes a Table of Contents page, as well as an index of keywords, and a full-text search engine so users can find additional information.

Figure 8.36 shows that you can create Help pages using a simple text editor, or you can use most HTML editors. You should create a style sheet just for the help files. Overall, the help pages are created as a separate Web site.

Once you have created the individual HTML pages, you should create the mapping file that assigns a name to each topic. In HTML, you refer to each topic by

Action

Create at least three HTML Help files for the All Powder forms using an HTML editor or Wordpad.

If necessary, download and install the Oracle Web Help and Java Help systems.

Create the Map, Index, TOC, and Link files.

Build the search.idx file.

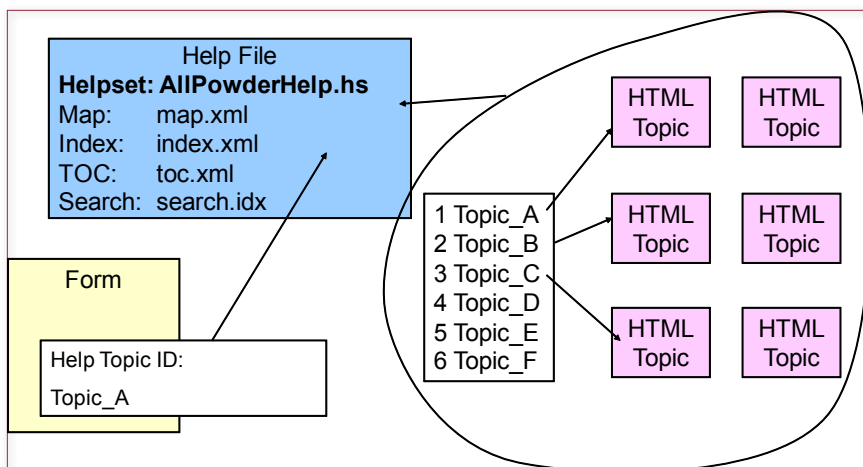
Create the new HelpSystem .hs file.

Edit the ohwconfig.xml file to add your helpsystem project.

Copy your files to the server and test the project.

Add a help link to a page using `http://localhost:7101/ohw-rcf-demo-thin/APHelp?topic=Customers.help`.

Figure 8.35



```

<html>
<head>
<title>All Powder Board and Ski Shop</title>
<link rel="stylesheet" type="text/css" href="Styles.css">
</head>
<body>
<h1>Introduction to the All Powder Board and Ski Shop</H1>
<table>
<tr>
<td></td>
<td>All Powder Board and Ski Shop sells and rents snowboards and skis for all levels
of riders and skiers.</td>
</tr>
</table>
<h2>The Board and Ski Shop</h2>
<ul><li><a href="Customers.html">Customers</a></li>
<li><a href="Sales.html">Sales </a></li>
</ul></body>
</html>

```

Figure 8.36

the name of the file, but Oracle references topics by a name. The name should contain only letters, numbers, and underscores. As shown in the sample in Figure 8.37, you can assign almost any name, but it should be recognizable because you will need it later. Note that the URL can also include a link to an identifier within a page, such as `AllPowder.html#section2`. This process enables you to place multiple, related topics within a single file. However, it is generally best to write shorter help pages and link them together.

Unless you have an automated Help editor, you will have to create the table of contents and index files manually. They are separate xml files with considerable flexibility. You can provide multiple levels of indented listings. Essentially, you enter a heading, followed by individual items. The individual items can link directly to a Help Topic page. Of course, you do not have to use headings, and you might not need them in the index. Figure 8.38 shows a sample TOC file. Notice the indentation used to show the levels of the final contents. The text entry is what the user will see, while the target entry is the name of the topic to be displayed. The topic must match an entry in the map file.

Figure 8.39 shows a sample index file. In general, index files can be quite long. Automated generators are sometimes useful in creating an initial word list. However, it helps if someone goes through and adds synonyms or other search topics that might be more useful to users. Again, the text entry provides the list of words the user will see, and the target is the topic that is displayed when the user selects a keyword. The `IndexEntry` values provide a simple hierarchical list of items, en-

Figure 8.37

```

<?xml version='1.0' ?>

<map version="1.0">
  <mapID target="AllPowder_help" url="AllPowder.html" />
  <mapID target="Customers_help" url="Customers.html" />
  <mapID target="Sales_help" url="Sales.html" />
</map>

```

```

<?xml version='1.0' ?>
<toc version="1.0">

  <tocitem text="Introduction to All Powder Board and Ski Shop">
    <tocitem target="AllPowder_help" text="The Board and Ski Shop" />
    <tocitem text="Sales Options" target="Sales_help" />
  </tocitem>
  <tocitem text="Customer Options">
    <tocitem target="Customers_help" text="Adding New Customers" />
    <tocitem target="Sales_help" text="Sales Options" />
  </tocitem>
</toc>

```

Figure 8.38

abling the indexer to group related topics together. For example, you might have a Customer topic, with Add, Delete, and so on as subtopics.

The search index is a little trickier, because it is a proprietary binary file. You need a tool to create the file for you. Commercial help editors can build it automatically. Oracle's Java Help system also has a Java-based tool to create the file. If you install the Java Help system (which is separate from the Oracle Web Help system), you can run a command-line Java program that reads the HTML files and builds the full-text index automatically. Figure 8.40 shows the two basic commands you need. However, you must customize the CLASSPATH variable and the folder name for your particular system. These commands might change as new versions of the tool are released. Hit the Enter key after you have entered the full

Figure 8.39

```

<?xml version='1.0' ?>
<index version="1.0">

  <indexitem text="All Powder">
    <indexentry target="AllPowder.html" text="All Powder" />
  </indexitem>
  <indexitem text="Management">
    <indexentry target="AllPowder.html" text="Management" />
  </indexitem>
  <indexitem text="Start">
    <indexentry target="AllPowder.html" text="Start" />
  </indexitem>
  <indexitem text="Client">
    <indexentry target="Customers.html" text="Client" />
  </indexitem>
  <indexitem text="Customers">
    <indexentry target="Customers.html" text="Customers" />
  </indexitem>
  <indexitem text="Sales">
    <indexentry target="Sales.html" text="Sales" />
  </indexitem>
  <indexitem text="Introduction">
    <indexentry target="AllPowder.html" text="The Company" />
    <indexentry target="Customers.html" text="Customers" />
    <indexentry target="Sales.html" text="Sales" />
  </indexitem>
</index>

```



```
set CLASSPATH=%CLASSPATH%;c:\program files\ohelp\help4-indexer.jar
java -mx64m oracle.help.tools.index.Indexer -l=en_US -e=8859_1 D:\Oracle\ohw\loc4j\
j2ee\home\applications\ohw-eapp\ohw\helpsets\AllPowder search.idx
```

Figure 8.40

CLASSPATH statement, and at the very end. Do not put line breaks in the middle of either command. The long folder name (D:\Oracle\...) is the full pathname of the folder holding your HTML files. The search.idx parameter is the name of the file to be created. You can change the name, but search.idx is a reasonable name for most applications.

As shown in Figure 8.41, the next step is to create the HelpSystem file. This is another XML file, although it is typically saved with an .hs suffix. It contains links to the other important files. For the most part, you can simply copy this example and change a couple of lines. For instance, you will want to change all of the <title> entries. But, as long as you stick with the standard names for the other files (map.xml, toc.xml, index.xml, search.idx, and link.xml), you can leave most of the file alone. The Links file is a little different. In many cases, you will not need it. It provides a way to consolidate several items into one associative link. You can simply copy the Oracle sample Link file: it will be ignored unless you need to create these special links.

The next step is to copy your Help system to the server and install it. This process might be slightly easier if you are using the full Oracle WebLogic server. In this case, the Help Listener service should already be installed (or you can in-

Figure 8.41

```
<?xml version='1.0' ?>
<helpset>
  <title>All Powder Board and Ski Shop</title>
  <maps>
    <mapref location="map.xml" />
  </maps>
  <links>
    <linkref location="link.xml"/>
  </links>
  <view>
    <label>Contents</label>
    <type>oracle.help.navigators.tocNavigator.TOCNavigator</type>
    <data engine="oracle.help.engine.XMLTOCEngine">toc.xml</data>
  </view>
  <view>
    <label>Index</label>
    <type>oracle.help.navigators.keywordNavigator.KeywordNavigator</type>
    <title>All Powder Board and Ski Shop</title>
    <data engine="oracle.help.engine.XMLIndexEngine">index.xml</data>
  </view>
  <view>
    <label>Search</label>
    <title>All Powder Board and Ski Shop</title>
    <type>oracle.help.navigators.searchNavigator.SearchNavigator</type>
    <data engine="oracle.help.engine.SearchEngine">search.idx</data>
  </view>
</helpset>
```

```

<books>
  <helpSet id="APHelp" location="APHelp/APHelp.hs" />
  ... other help set files
</books>

```

Figure 8.42

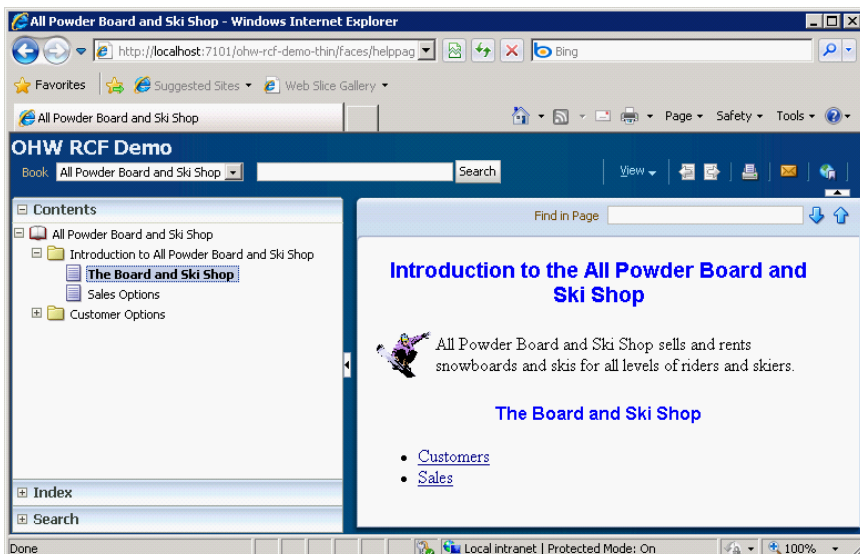
install the Help system's Deploy tool). A special folder and the ohwconfig.xml file should already exist. Create a new subfolder under the helpsystem folder, and edit the ohwconfig.xml file to add a line pointing to your new Help system (.hs) file. If you are running the Developer Suite, you will have to install yet another OC4J servlet, find the helpsystem subfolder, and locate the ohwconfig.xml file. Figure 8.42 shows the portion of the configuration file you need to edit. Simply add a row that lists the location of your new .hs file.

You can now open the help file with a Web browser. On most systems, the Help file will open with the link: `http://localhost:7101/ohw-rcf-demo-thin/ohguide`. Figure 8.43 shows the initial Help file for the All Powder case. Notice the tabs for Contents, Index, and Search. Clicking these tabs displays the appropriate forms that you created. You should test each one to ensure that they work properly. Also, be sure to test all links. The configuration file enables you to set a few additional options. For instance, you can change the branding text ("Oracle Help for the Web") and even add an image.

The final step is to link your Help topics into the database forms. Because Oracle has changed Help systems several times, and the help system and ADF system are both new, there are two methods of adding context-sensitive help. Figure 8.44 shows one approach. Simply enter the help topic into the HelpTopicId property for a control. The topic name must match one of the entries in the map file that you created earlier.

For this approach to work, you must also include a reference file in your application that builds a link to the Oracle help server. Figure 8.45 shows a sample

Figure 8.43



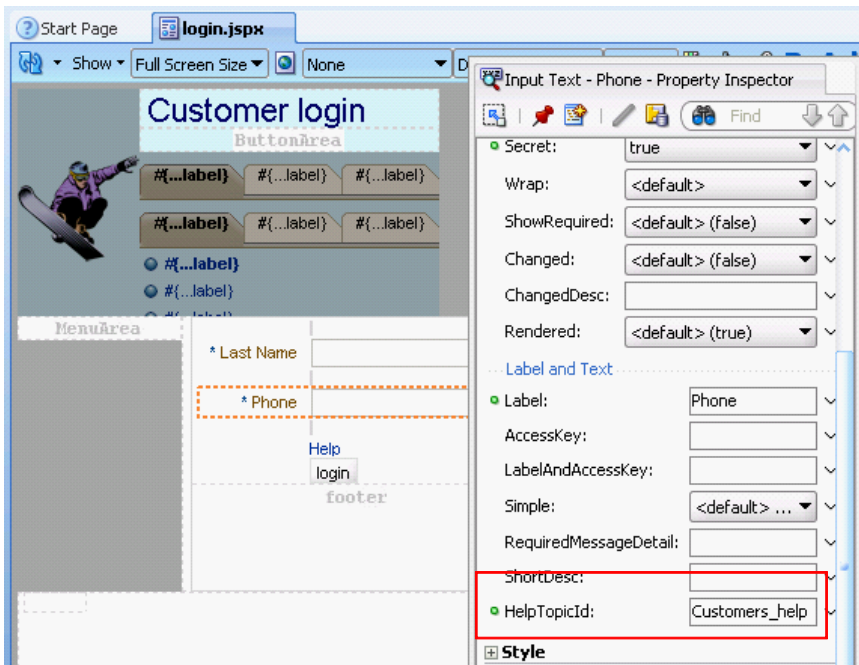


Figure 8.44

adf-settings.xml file. This file needs to be manually created in your project's .adf/META-INF folder.

The adf-settings file contains a link to the Oracle Web server and the ohwconfig.xml file. In theory, these settings should enable JDeveloper to add a help icon to the page (question mark) which then links to the specified topic. However, I have been unable to find the combination of options to make it work automatically.

Instead, you can add a help link on the page and point it to the help server. Move all of your help files to a new helpset folder on the Oracle Help server. If

Figure 8.45

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/settings">
  <help-provider>
    <help-provider-class>oracle.help.web.rich.helpProvider.OHWHelpProvider
    </help-provider-class>
    <property>
      <property-name>ohwConfigFileURL</property-name>
      <value>/helpsets/ohwconfig.xml</value>
    </property>
    <!--property>
      <property-name>group</property-name>
      <value>null</value>
    </property-->
    <property>
      <property-name>baseURI</property-name>
      <value>http://localhost:7101/ohw-rcf-demo-thin/ohguide/</value>
    </property>
  </help-provider>
</adf-settings>
```

you installed the demo project, you can search for the “Shakespeare” folder that contains some of the demonstration files. Create a new APHelp folder in parallel with that one and add the APHelp book to the associated `owhconfig.xml` file. The WebLogic server needs to be running (start a form in JDeveloper if necessary). Then test the link by opening a Web browser directly to the server: `http://localhost:7101/ohw-rcf-demo-thin/ohguide`. If you can manage to find the main `web.xml` file, you can replace the `ohguide` entry with the new APHelp folder. Then you can create context-sensitive help links by adding the topic name at the end of the URL: `http://localhost:7101/ohw-rcf-demo-thin/APHelp?topic=Customers_help`. Eventually, you will want to create a global bind variable to hold the base URL, and store its initial value in a file to make it easier to move the application to a different server. However, as long as you know the name of the help server when you build the pages, you could just use the simple link on each page.

In the end, placing a single Help link on each page is probably more useful than defining help topics for every single text control on the page. So, building a separate link—probably with your own icon—is the most efficient approach.

Did the help system seem like a lot of work to you? It is. And getting everything to run within an Oracle application is even more work. There is a better answer: just write your own help files in HTML and stop worrying about Oracle’s approach. You can add your own TOC and index system easily. In fact, if you license a standard Web search engine, you can save even more time and provide better search capabilities. You still have to write the HTML pages either way, and you still need to build a table of contents page. You can use standard server-side scripting in the help system to pick up name of the sending page and use that to display the appropriate topic. With this approach, you can also make the help pages directly available to your users via a simple Web link, without going through the Oracle system. With the wealth of HTML development tools available, it is much easier to build the help system as a regular Web site.



Activity: Deploy Application

Creating an Oracle application is a complex task. You have to define the database tables, create the forms and reports as well as the help files, and integrate everything together into a consistent application. But you are still not finished: You have to deploy the application to an application server so it can be accessed by the users. Because users access the application over the Internet, the deployment process is easier than it used to be—but it still requires the assistance of a database administrator, and considerable testing. One of the most important steps is to build SQL script files to create the database and transfer the initial data. Once the database tables and triggers have been created, the DBA can help assign the proper security permissions. (Security and other DBA tasks are covered in Chapter 10.)

As shown in Figure 8.46, you also have to transfer your compiled forms, reports, and help files to the application server. Of course, that means you have to purchase the application serv-

Action

Create SQL scripts to rebuild your database on a new server.

Try to get to an Oracle Application Server.

Have a DBA configure a new database and the AS for a new application.

Change your global variables to point to the new locations.

Build the new database.

Compile and transfer the application files.

Test everything.

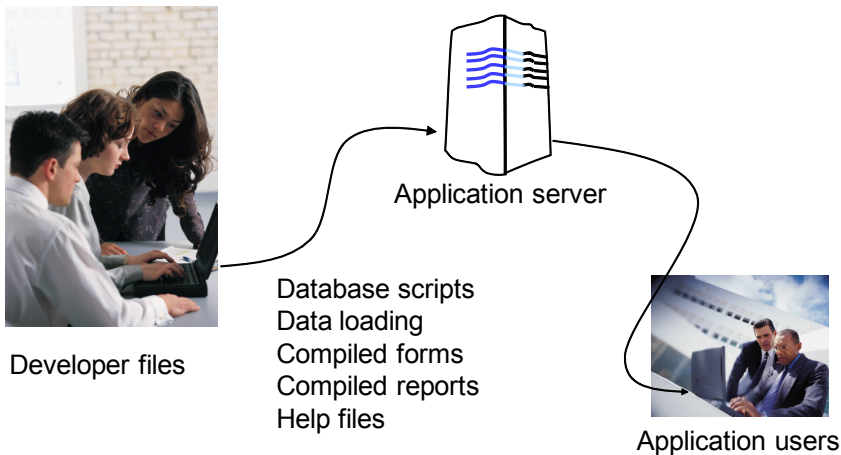


Figure 8.46

er license, and get an administrator to configure the new application. Once the files have been transferred and tested, the users can open the startup form with a browser. See Chapter 36 in the Fusion Developer Guide for details on deploying applications to the server. JDeveloper has an option to deploy directly to the server or to create an archive (EAR) file which can then be installed on the server.

When you copy your forms to the application server, you will probably have to reconfigure most of your global variables. Recall that global variables are used to point to locations for files, which means you can change the pathname in one location without needing to edit every single function call. You will have to recompile the forms, transfer the executable versions, and test everything. Although the configuration process can be cumbersome, and you will need the assistance of an application server administrator, once the system is configured, it is relatively easy to change. If you need to modify a form or report, you simply post the new version on the server and everyone will be using the new copy. Remember that any changes to the application will require substantial testing.

Exercises



Crystal Tigers

The Crystal Tigers club is mostly interested in tracking members and events. The officers who will use the system do not know much about computers, but they can enter data into forms. They are also interested in a few key reports. For instance, they want to be able to get totals for the number of hours members devoted to charity events. They also want monthly summaries of the amount of money raised. The vice president also wants to be able to print a simple listing of the officers, their phone numbers and e-mail addresses. Sometimes, she also wants a similar list for members who have participated in the initial steps of an event. She wants to be able to carry the list with her when the event starts so she knows who to contact if problems arise.

1. Create a design template and standardize the forms and reports.
2. Build the forms and reports into an application with a start-up form.
3. Create the Help files for the system, and remember that the users have limited computer experience.



Capitol Artists

Job tracking is the most important aspect of the application needed by Capitol Artists. In particular, the employees need to be able to quickly select a job and enter the time and expenses for the task performed. This data is then used to create a monthly billing report for the client. Consequently, you need to focus on creating the forms to capture this data. You need to make sure they are fast and easy to use. The managers also want weekly reports showing the hours and money generated by each employee so they can use the data in personnel evaluations.

1. Create a design template and standardize the forms and reports.
2. Build the forms and reports into an application with a start-up form.
3. Create Help files for the system.



Offshore Speed

Special orders have always been a complex problem for the Offshore Speed managers. Customers come to the shop because it is one of the few that can obtain the custom parts they want. But the company has always had problems training employees to collect all of the order data and, keep track of getting the orders placed and delivered in a timely manner. Some of these orders include contracts with other local firms to perform customization and finish work on the boats. Although these firms do excellent work, most are terrible at keeping records. Consequently, the managers want to use the system to generate reports on individual boats for each contract shop that can be used to remind the other owners of the details. The company also needs reports on the inventory status of the specialized parts. They are having trouble keeping some items in stock, and other items seem to sit on the shelves forever; but they have no good way of keeping track at the moment.

1. Create a design template and standardize the forms and reports.
2. Build the forms and reports into an application with a start-up form.
3. Create Help files for the system.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you pick one or your instructor picks one, perform the following tasks.

1. Define a form template and standards for consistency.
2. Build the forms and reports into an application with a start up form.
3. Build a menu toolbar that makes the application easier to use.
4. Create help files for the system.

Data Warehouses and Data Mining

Chapter Outline

Data Warehouse, 218

Tools and Downloads, 219

Case: All Powder Board and Ski Shop, 220

Lab Exercise, 220

All Powder Board and Ski Shop, 220

Introductory Data Analysis, 236

Exercises, 246

Final Project, 248

Objectives

- Extract data from spreadsheets and import it to a data warehouse.
- Create and browse an OLAP cube.
- Analyze time-series data.
- Analyze data classifications.
- Analyze data with regression.
- Analyze association rules for market baskets.

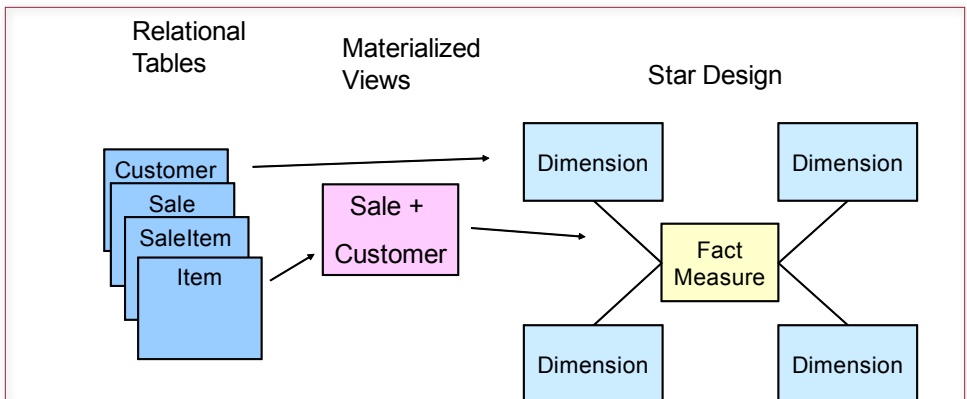
Data Warehouse

Data warehouses have evolved because of the need for online analytical processing (OLAP) and its conflicts with online transaction processing (OLTP). The goal of a data warehouse is to hold consistent data, possibly obtained from several sources, which can be quickly searched and analyzed. Oracle has several data warehouse and OLAP capabilities. Over time, Oracle has acquired several other firms that specialize in data mining tools. Two of the bigger purchases were Seibel and Hyperion. Consequently, Oracle has several tools to create data warehouses and analyze data. Unfortunately, it will take years for them to be integrated. So, one of the first problems you face in using Oracle for data warehousing and analysis is to determine which set of tools to use. The purchased components largely remain as separate tools so they are not covered in this chapter. The older tools (Discoverer series) still exist and they work reasonably well. Those tools are covered in the 10g Workbook. They are particularly useful if you need a consistent set of tools that can deliver content over the Web. Because they are covered in the 10g Workbook, they are not covered here. Oracle also has some interesting extensions to SQL (and Java) that are useful for data analysis that needs to be automated. These tools work reasonably well and are explained in the Oracle data warehouse documentation.

Oracle has two newer tools for OLAP and data mining. Oracle Web Analytics Workspace Manager is positioned to be used to define multidimensional cubes for data browsing. Because it follows relatively standard processes, it is used to explain the creation of data cubes. However, it has some problems (in early 2010) that might or might not be fixed by the time you work with it. Be sure to download the most recent version. For Data Mining, Oracle Data Miner provides a graphical interface to the underlying analysis tools. It is a relatively painless way to begin exploring some traditional data mining tools.

Figure 9.1 shows the basic concepts involved in the Oracle data storage method. First, you have to import all of the necessary data into Oracle tables. Second, you define the OLAP business perspective queries in terms of a star (or snowflake) design by creating dimensions that are related to the fact measures. You often create hierarchical groupings on the dimensional data. Although this process appears similar to the traditional OLAP approach, the big difference is that the new star design does not hold any of the data. It is simply a meta-data definition that describes how to retrieve data from the relational tables. But since data stored

Figure 9.1



http://www.oracle.com/technology/products/bi/olap/index.html	Main Business Intelligence site.
http://www.oracle.com/technology/products/bi/odm/index.html	Oracle Data Mining including download
http://www.oracle.com/technology/products/discoverer/index.html	Discoverer
http://www.oracle.com/technology/software/products/ias/htdocs/101320bi.html	OTN BI Suite EE
http://www.oracle.com/technology/products/bi/olap/olap_downloads.html	Analytic Workspace Manager Download

Figure 9.2

across multiple tables requires joins that can be slow to compute, Oracle introduced materialized views. A materialized view is essentially a temporary, non-normalized table. When extensive joins or complex calculations are needed, you can create a materialized view that holds the results of the computations and joins. The OLAP system then pulls the data from this temporary view instead of returning to the original tables. To improve performance, Oracle relies on snapshots of the data called materialized views. These snapshots can be designed to reload data automatically on a schedule.


Tools and Downloads

The OLAP and data mining tools needed in this chapter are not automatically installed with the database. In a production environment, you need to pay extra money for the components. However, they are available as free downloads from OTN for learning and trial purposes. At a minimum, you need to download and install the Analytic Workspace Manager and the Oracle Data Miner software. Installation for both of these tools largely consists of unzipping the files to separate folders (not in the Oracle database directory). Then create a shortcut to the startup file—see the Read Me instructions with the two files because the process changes with different releases. Figure 9.2 shows the main links to the Business Intelligence tools at Oracle including the download URLs. You might also want to download many of PDF documentation files.

After you install the tools, you might need additional permissions on your account to use the new features. Figure 9.3 summarizes the additional permissions you need to create cubes and to use the data miner tools. See the Oracle documentation for details. For example, the EXECUTE permission for `ctxsys.ctx_ddl` is

Figure 9.3

```
Additional DB Permissions
CREATE MATERIALIZED VIEW
CREATE CUBE
CREATE CUBE DIMENSION
CREATE MINING MODEL
CREATE SESSION
OLAP_USER
CREATE MINING MODEL
CREATE JOB
CREATE TYPE
CREATE SYNONYM
EXECUTE ON ctxsys.ctx_ddl
```



Sales Dimensions
State (ship)
Month
Category
Style
SkillLevel
Size
Color
Manufacturer
BindingStyle
WeightMax?
ItemMaterial?
WaistWidth?

Figure 9.4

only needed for text mining in ODM. That feature is not used in this workbook, so it is not needed here.

Case: All Powder Board and Ski Shop

Like most businesses, the managers of All Powder need to analyze data to spot trends and solve problems. One of the most challenging aspects of a board and ski shop is the huge variety of inventory needed. As vendors produce even more styles and variations, it becomes difficult to stock all of the items in a collection of sizes. Yet, if the store does not have items in stock, it will lose sales. This balancing act between inventory costs and sales revenue has destroyed many other firms. The owners of All Powder are committed to running a large enough shop so that they can afford to carry a large selection of snowboards and skis. However, managers need to constantly evaluate styles and products so items can be cleared out if needed. For that analysis, one of the main tools they need is an OLAP cube browser or perhaps the Microsoft PivotTable that shows sales split by several features and categories. Figure 9.4 lists some of the main dimensions that managers want to examine in terms of sales. They are not certain about the validity of the last three, so they are displayed with question marks.

Managers also occasionally raise some more challenging statistical questions, such as whether customers who rent equipment are likely to buy that equipment, and whether skiers buy certain types of poles or boots with their skis. They also need to forecast sales by categories. In particular, they often argue about whether certain styles are increasing or decreasing in popularity. Some of these analyses might require the help of a statistician to build a formal model, but the managers would at least like to see some rough analyses.

Lab Exercise

All Powder Board and Ski Shop

Note: You might want to skip the first lab and jump to the second lab. The first lab explores issues in importing data and building data warehouses. It is useful, but time consuming. You can jump to the second lab and install the sample data files that already have the data.

As organizations grow over time, the internal processes undergo changes, data changes, systems improve, and number systems rarely stay the same. Consequently, most information systems consist of a mix of technologies and databases.

The figure shows two overlapping screenshots of Microsoft Excel. The top screenshot displays the 'Sales 2001-2003' worksheet, and the bottom screenshot displays the 'Rentals 2001-2003' worksheet. Both worksheets show columns for dates, IDs, descriptions, prices, and categories.

SaleID	SaleDate	ShipState	ShipZIP	PaymentM	SKU	Quantity	StSalePrice	ModelID	Size	Manufactu	Category	Color
2	1566	1/1/2001	CA	95014	Check	50863	1	\$289.00	MKY-77	1200	37 Board	Tun
3	1566	1/1/2001	CA	95014	Check	51591	1	\$399.00	HG-C-505	750	39 Board	Wh
4	1566	1/1/2001	CA	95014	Check	920182	1	\$5.00	OTN-6	2	84 Wax	Tun
5	1587	1/3/2001	VA									
6	1655	1/4/2001	FL									
7	1655	1/4/2001	FL									
8	1655	1/4/2001	FL									
9	1658	1/9/2001	IL									
10	1658	1/9/2001	IL									
11	1543	1/18/2001	IA									
12	1521	1/19/2001	NV									
13	1521	1/19/2001	NV									
14	1521	1/19/2001	NV									
15	1521	1/19/2001	NV									
16	1561	1/21/2001	IL									
17	1519	1/21/2001	WA									
18	1519	1/21/2001	WA									

RentDate	RentID	ExpectedP	PaymentM	SKU	RentFee	ReturnDate	ModelID	Size	Manufactu	Category	Color	Mo
2	1/1/2001	6253	1/4/2001	Mastercar	151330	\$45.00	1/3/2001	BVG-290	6	44 Boots	White	
3	1/1/2001	6253	1/4/2001	Mastercar	751630	\$60.00	1/4/2001	UWW-707	210	5 Ski	Purple	
4	1/1/2001	6253	1/4/2001	Mastercar	752736	\$60.00	1/2/2001	TZH-696	180	24 Ski	Green	
5	1/1/2001	6491	1/2/2001	American I	151202	\$15.00	1/2/2001	AEF-37	6	41 Boots	Black	
6	1/1/2001	6491	1/2/2001	American I	151226	\$15.00	1/4/2001	VOO-380	6	42 Boots	Magenta	
7	1/1/2001	6491	1/2/2001	American I	51535	\$20.00	1/3/2001	EKR-463	600	26 Board	Yellow	
8	1/2/2001	6028	1/3/2001	Other cred	550273	\$5.00	1/5/2001	MBG-699	900	87 Poles	Green	
9	1/2/2001	6028	1/3/2001	Other cred	752503	\$20.00	1/3/2001	DVY-990	165	10 Ski	Red	
10	1/2/2001	6028	1/3/2001	Other cred	752659	\$20.00	1/4/2001	DMR-560	165	18 Ski	Turquoise	
11	1/2/2001	6072	1/4/2001	Debit Card	151202	\$30.00	1/3/2001	AEF-37	6	41 Boots	Black	
12	1/2/2001	6080	1/5/2001	Debit Card	751696	\$60.00	1/5/2001	CWQ-771	165	9 Ski	Blue	
13	1/2/2001	6159	1/3/2001	Other cred	751574	\$20.00	1/3/2001	FGA-779	165	1 Ski	Black	
14	1/2/2001	6159	1/3/2001	Other cred	752593	\$20.00	1/5/2001	YTN-679	180	16 Ski	Turquoise	
15	1/2/2001	6159	1/3/2001	Other cred	752741	\$20.00	1/4/2001	EXM-971	180	24 Ski	Blue	
16	1/2/2001	6180	1/5/2001	Cash	50947	\$60.00	1/4/2001	VFJ-740	750	33 Board	Red	
17	1/2/2001	6180	1/5/2001	Cash	51579	\$60.00	1/3/2001	DFA-210	750	38 Board	Black	
18	1/2/2001	6181	1/4/2001	Cash	151203	\$30.00	1/5/2001	AEF-37	6	41 Boots	Black	

Figure 9.5

Rarely is the data consistent across all of these systems. For the All Powder shop, before the database was created, the managers kept limited records in Microsoft Excel. These records are not perfect: They are organized by Sales and by Rentals and the data is not normalized. Also, they are focused primarily on the equipment and did not keep data on customers. From our more modern database perspective, the records are a pain, but at least they are electronic and not paper so you do not have to enter all of the data by hand.

Nonnormalized data is common in business, and you will often be asked to convert this data into a relational database. Fortunately, you can use the power of SQL as a magical super tool to impress mere mortals with your skills. Figure 9.5 shows the layout of the data in the two worksheets. Again, notice that lack of normalization. Each row represents an item that is sold or rented. Fortunately, the worksheets repeat the SalesID and RentalID so you can still recover which items are grouped onto a single sale or rental. Likewise, they repeat the descriptive item data for each time the model was sold. To ensure your information is really accurate, you should eventually check to see that the managers were consistent in recording this data. For example, ModelID BVG-290 might have been given a different description at different times. If there are many inconsistencies of this type, it will be difficult and time-consuming to clean up this data. Most of the corrections would have to be handled manually, unless you have a third source of data that you know is correct. These are the types of problems you often face when extracting data from diverse systems.



Activity: Extract and Transform Data

One of the most challenging and time-consuming steps in creating a data warehouse is transferring and cleaning the data. This section explains some of the tools Oracle provides for this process, but Oracle also has a complete data warehouse system with even more options. In terms of an in-class lab, this section is time-consuming. If you do not have time to go through the data loading steps in this

section, you can run the BuildAllPowder file for this chapter. This command will delete your existing tables, rebuild them, and load them with the full dataset.

The first step in extracting and transforming this data is to get it into the database where you can use SQL to work on it. Oracle provides two related tools to import data from text files: the SQL*Loader and external tables. SQL*Loader is a command-line utility (sqlldr) that reads data from a variety

of text files and transfers it into SQL tables. It uses a sophisticated control file to specify the location of the data and the format of the data in the files. The files have to be plain text files but can have delimited or fixed-width columns. It often takes considerable time to set up the control file correctly. However, once the control file is correct, you can use it to extract data from the same source time after time. It is a useful tool for transferring data on a regular basis.

External tables have similar capabilities—since the data reader is based on the SQL*Loader engine. However, data is not really transferred out of the flat files. Once you define the data format and declare it as an external table, you can use SQL and treat the data as if it were stored in a regular Oracle table. Because an external table is declared using PL/SQL, it is a little easier to set up and to work with interactively until you get the definition correct. At that point, it takes a simple SQL SELECT INTO command to transfer all of the data from the external file and copy it into a standard Oracle table. Oracle also provides the Warehouse Builder tool that has similar features to import data from flat files. This tool is actually better at handling other tasks, such as cleansing addresses and rebuilding OLAP cube dimensions. Since the older sale and rental data only needs to be loaded one time, and it requires some effort to split it into normalized form, it is easier to do the transformation one time. If you check the initial AllPowder files you used to load the sample database, you will see that the CSV approach was used to load that data. Now you get to look at the process in a little more detail so you can use it for your projects.

Because they are a little easier to set up, you should use an external table to extract the data for this activity. Unfortunately, the loader cannot read an Excel worksheet directly; consequently, the two worksheets have been saved as comma-separated value (CSV) files. CSV files are a common format used to transfer data. The files are simple text files that hold one row of data on a line. The values for each column are separated by commas (hence the name). Although the system is relatively easy, you must be careful that the text data does not contain additional commas that would throw off the parser.

In the All Powder case, some of the earlier exercises have added new data to the Sale table. You need to ensure that the sales being imported do not have SaleID values that conflict with the existing data. To be safe, delete any sales with a SaleID greater than 2000 in the existing Oracle database. A related problem arises from the trigger that automatically generates a new key value for SaleID whenever a row is inserted. This trigger should be disabled while you import the new data: ALTER TRIGGER GenKeyForSale Disable;

Action

Decide if you want to skip this section.

Delete all sales with SaleID>2000.

Disable the trigger that creates SaleID keys:

```
ALTER TRIGGER GenKeyForSale
  Disable;
```

Modify and run the code to create the external tables for the old Sale and Rental data.

Insert CustomerID 0 and EmployeeID 0.

You are now ready to create external table links in Oracle that point to the two CSV data files. The data files and the PL/SQL code to create the links are on the student CD. Copy these three files to a folder on the server. Figure 9.6 shows the code for the first half of the ReadOldSalesAndRentals.sql code file. At the top of the file, you will have to edit the name of the folder that contains the two CSV files. Then you can save and run the program to create the links. Before running the code, it is worth examining it so that you can modify it to handle similar prob-

Figure 9.6

```

rem change the folder to point to the location of the CSV files
create or replace directory csv_dir as 'D:\students\BuildAllPowder\
AllPowderSampleDataCSV';

create table OldSale_Ext
( SaleID                INTEGER,
  SaleDate              DATE,
  ShipState             VARCHAR2(50),
  ShipZIP               VARCHAR2(50),
  PaymentMethod        VARCHAR2(50),
  SKU                   VARCHAR2(50),
  QuantitySold         INTEGER,
  SalePrice             NUMBER(10,2)
  ModelID              VARCHAR2(250),
  ItemSize              NUMBER,
  ManufacturerID       INTEGER,
  Category              VARCHAR2(50),
  Color                 VARCHAR2(50),
  ModelYear             INTEGER,
  Graphics              VARCHAR2(50),
  ItemMaterial          VARCHAR2(50),
  ListPrice             NUMBER(10,2),
  Style                 VARCHAR2(50),
  SkillLevel            INTEGER,
  WeightMax             NUMBER,
  WaistWidth            NUMBER,
  BindingStyle         VARCHAR2(50)
)
organization external (
  type oracle_loader
  default directory csv_dir
  access parameters (
    records delimited by newline
    fields terminated by ','
    optionally enclosed by '"' ltrim
    missing field values are null
  )
  (
    SaleID,
    SaleDate char date_format date mask "mm/dd/yyyy",
    ShipState, ShipZIP, PaymentMethod, SKU, QuantitySold,
    SalePrice, ModelID, ItemSize, ManufacturerID, Category,
    Color, ModelYear, Graphics, ItemMaterial, ListPrice, Style,
    SkillLevel, WeightMax, WaistWidth, BindingStyle
  )
)
location ('Lab 08-01 Early Sales.csv')
)
reject limit unlimited;

```

lems in the future. The first section is just a standard CREATE TABLE command that identifies the columns and data types used in the table. These elements are necessary because they describe the table as it will be accessed by SQL. The phrase “organization external” and the associated parameters are the features that establish the link to the actual data file. Notice the specification of oracle_loader and the directory. If you look near the bottom of the code, you will see the actual name of the file specified in the location parameter. The main section describing the parameters indicates that each record is stored on

a single line and the fields are separated by commas. Some CSV systems place quotation marks (") around text items, so that option is specified to be safe. The null values note is probably not needed, but it makes it clear how missing data should be handled. The listing of the fields must match the order and type of data as it is listed in the table. Note that you will almost always have to specify a mask for dates, because Oracle reads only dd-MMM-yyyy formats by default, and most other software uses the mm/dd/yy format. Numeric and character fields generally do not need formatting hints, but you can provide them if necessary. One final note of warning: Currency data must not contain a \$ character. Many export systems, including Excel, use them by default, and you must remove them from the file before trying to load it. Generally, you can quickly remove all dollar sign symbols using global search and replace in a text editor. You should now be able to save and run the code to attach to the two files.

Once the external files have been defined, you can access them with SQL as if they were Oracle tables. Without indexes, they will be a little slower, but you need SQL to extract the data correctly and move it into the main relational tables. Once the data has been extracted, you can delete the links to the external tables.

Looking through the temporary Sale table, you will see that the data needs to be split into four tables: SaleItem, Sale, Inventory, and ItemModel. Go back and examine the relationships for those tables, and you will see that because of the dependencies, you will have to enter data first into the tables for ItemModel, Inventory, Sale, and finally SaleItem. The relationships and foreign keys require that data be entered in that order. You must also be careful with the Customer and Employee data. If you try to create a row in the Sale table, the system will try to set a value of zero for the CustomerID and EmployeeID. But there is no matching data for a zero ID in either of these tables. So, either you try to force a blank CustomerID and EmployeeID, or you create a new Customer and new Employee

Action

Create a new query that retrieves DISTINCT values from the saved UNION query.

Verify that it works.

Add an INSERT INTO statement above the SELECT statement to copy the data to the ItemModel table.

Run the query.

Use a similar process to add SKU, ModelID, and Size to the Inventory table.

Follow a similar process to copy the Sale, Rental, SalesItem, and RentalItems tables.

Figure 9.7

```
INSERT INTO Customer (CustomerID, LastName)
Values (0,'Walk-in')
```

```
INSERT INTO Employee (EmployeeID, LastName)
Values (0,'Staff')
```

```

SELECT DISTINCT OldSale_ext.ModelID, OldSale_ext.ManufacturerID, OldSale_ext.
Category,
OldSale_ext.Color, OldSale_ext.ModelYear, OldSale_ext.Graphics, OldSale_ext.
ItemMaterial,
OldSale_ext.ListPrice, OldSale_ext.Style, OldSale_ext.SkillLevel, OldSale_ext.
WeightMax,
OldSale_ext.WaistWidth, OldSale_ext.BindingStyle
FROM OldSale_ext;

```

Figure 9.8

called “walk-in” and “staff.” This latter approach is slightly better than relying on blank data. So your first task is to create these new entries in the respective tables. Figure 9.7 shows the basic SQL commands needed to create these two entries.

SQL makes it relatively easy to extract the new model data and copy it to the ItemModel table. The first step is to create a SELECT query that retrieves the model data from the temporary tables and removes the duplicates. This process is slightly complicated because of the two tables. It is possible that an item model has been sold but not rented and vice versa. The easiest way to handle this problem is to write two queries and use UNION to combine the results. Figure 9.8 shows the basic query to retrieve the model attributes from the OldSale table. Move this query to the side and build a similar one from the OldRentals table. Be extremely careful to list the columns in exactly the same sequence.

```

SELECT DISTINCT ModelID, ManufacturerID, Category, ...
FROM OldSales
UNION
SELECT DISTINCT ModelID, ManufacturerID, Category, ...
FROM OldRentals

```

Figure 9.9

Add the data rows from the two queries with the UNION statement. Figure 9.9 shows the basic structure of the query but yours will contain several more columns. Save this query as qryOldModels so you can use it as one set of data.

Now that you can retrieve the new model data, it is relatively easy to write a query to insert these rows into the base ItemModel table. Build a new SELECT query using the qryOldModels query with all of its columns. Add the DISTINCT keyword to be absolutely certain that all duplicates are removed. Run the query to make sure it retrieves the data. As shown in Figure 9.10, at the top of the query

Figure 9.10

```

INSERT INTO ItemModel (ModelID, ManufacturerID, Category, Color, ModelYear,
Graphics, ItemMaterial, ListPrice, Style, SkillLevel, WeightMax, WaistWidth,
BindingStyle)
SELECT DISTINCT qryOldModels.ModelID, qryOldModels.ManufacturerID,
qryOldModels.Category, qryOldModels.Color, qryOldModels.ModelYear, qryOldModels.
Graphics, qryOldModels.ItemMaterial, qryOldModels.ListPrice, qryOldModels.Style,
qryOldModels.SkillLevel, qryOldModels.WeightMax, qryOldModels.WaistWidth,
qryOldModels.BindingStyle
FROM qryOldModels;

```

```
INSERT INTO Inventory (ModelID, SKU, Size, QuantityOnHand)
SELECT DISTINCT qryOldInventory.ModelID, qryOldInventory.SKU,
qryOldInventory.ItemSize, 0 As QuantityOnHand
FROM qryOldInventory;
```

Figure 9.11

add the phrase: `INSERT INTO Item Model (ModelID, ...)`. Because you do not have data for all of the columns, you must list them in the parentheses and they must be in the order of the columns being selected. Run the query and all of the new models will be added to the ItemModel table.

Follow a similar process to add the SKU, ModelID, and Size data to the Inventory table. Note that you should set the QuantityOnHand to zero for each of these items since the store probably does not have any of the old models in stock. If they do happen to have a few items around, the quantity can be entered by hand later. Figure 9.11 shows the final step that inserts the data into the Inventory table. Remember that you have to create the UNION query first. Notice the use of the column alias to force a zero value into the QuantityOnHand column for each row.

The Sale and Rental data is considerably easier because they are separate and you will not need the UNION command to merge the two sets of data. In fact, you can copy the Sale (or Rental) data with one SQL command. First, build a query to retrieve the distinct sales data from the OldSale_ext table. Be sure to include the DISTINCT keyword in the SELECT statement. After you test the SELECT statement, add the INSERT INTO line above it. Figure 9.12 shows an additional trick that is often helpful. If you added new rows of data to your Sale table, the system might have generated values that would conflict with the values from this older dataset. To avoid this problem, you can add an offset number to the old SaleID (+5000 in this example). If you choose a large enough offset, this step will ensure that all of the new ID values will be safe. However, you must also remember to add the same calculation in the final step of transferring the SaleItem rows.

Figure 9.13 shows that the query for the SaleItem table is almost identical to the query that copied the sale data, but with slightly different columns. Remember that if you transform the SaleID in the Sale table, you must make the identical transformation for the SaleItem table. Otherwise, the data will never match and cannot be joined. If you forget, you will usually receive several error messages. But some of the data might be joined to your existing Sales data, making it difficult to reverse the query. Finally, you need to do the same two steps for the Rental and RentalItem tables. The Rental table uses columns RentID, RentDate, ExpectedReturn, and PaymentMethod. The columns for the old rental table do not include repair charges and are limited to RentID, SKU, RentFee, and ReturnDate. At this point, you have successfully imported the old data and cleaned it

Figure 9.12

```
IINSERT INTO Sale (SaleID, SaleDate, ShipState, ShipZIP, PaymentMethod)
SELECT DISTINCT OldSale_ext.SaleID+5000, OldSale_ext.SaleDate,
OldSale_ext.ShipState, OldSale_ext.ShipZIP, OldSale_ext.PaymentMethod
FROM OldSale_ext;
```



```
INSERT INTO SaleItem (SaleID, SKU, QuantitySold, SalePrice)
SELECT DISTINCT OldSales.SaleID+5000, OldSales.SKU, OldSales.
QuantitySold, OldSales.SalePrice
FROM OldSales;
```

Figure 9.13

up so it can be used within your database. Finally, now that the data loading is finished, you might want to reenable the data trigger that generates keys for the sale table (GenKeyForSale). At this point, you should also drop the two external tables because they are no longer needed. You can use a simple DROP TABLE OldSale_ext command.



Activity: Use Analytic Workspace Manager to Create a Cube

Investigating sales by a variety of dimensions is an important task for the managers and owners of All Powder. It would be difficult to train all of them to build queries to examine all of the items that might be of interest. A faster and more flexible solution is to create an OLAP cube that contains the sales value (price times quantity) as the factor, along with the dimensions. Using the Analytic Workspace Manager, the cube can be manipulated to see subtotals and sort or filter the dimensions. Managers can also create charts and select the data to be displayed. The critical first step in any project is to interview managers to determine what types of data they need to see and how it will be analyzed. For this example, you will create a simple business area that enables managers to analyze sales by various dimensions.

Begin the process by running the OLAP Analytic Workspace Manager. Create a connection to the server using your normal account login. In a production environment, you would create a separate login account just to handle the data storage for the OLAP tables. The system also creates several views in the account, so you have to be careful not to delete them. These tables and views can be confusing when they are stored in your regular schema, but for experimentation it is easier to stick with a single account.

When the system is set up, right-click the Analytic Workspaces node and choose the option to create a new Workspace. Name it AllPowder and accept the default tablespace. Expand the new workspace and check out the options.

The basic steps to creating an OLAP cube with this tool are to define the dimensions (such as time, location, and product color). The dimensions are essentially look-up values that contain the available data. For example, a color dimension would contain a list of possible colors. Ultimately, these data values must be loaded into the dimension so they can be selected by the users. Dimensions can be defined in terms of hierarchies. The most common hierarchies in business are time (Year, Quarter, Month) and location (Nation, State, City). However, you can create hierarchies based on any type of data, such as manager-employee relationships, or product categories. Hierarchies give managers the ability to drill down or rollup the data to see subtotals and compare values at different levels.

Once the dimensions have been created, you can define a cube. A cube is defined by assigning dimensions to it and creating measures. Measures are the values that managers want to see. The most common example is total sales which is the sum of price times the amount purchased.

The first task is to define a new dimension. You might as well start with the hardest one first: time. Time is a classic hierarchy, and Oracle has special procedures to handle time dimensions. Creating the dimension and the hierarchy is relatively straightforward. Right-click the Dimension entry and choose the option to create a new dimension. Name it: Dim_Sale_Date. Because of some acknowledged bugs in the program, it is important to choose slightly mangled names. You must avoid using any name that might match a reserved word—and there are hundreds of reserved words.

As shown in Figure 9.14, on the same first page, set the Dimension Type to Time Dimension. Next click the Levels tab and enter the three levels in order: Sale_Year, Sale_Quarter, and Sale_Month. Finally, select the tab for Implementation Details. Change the key type from generated surrogate key to Use Key from the Data Source. It is almost always easier to use the keys in the database. However, if data in different levels can be the same, then you will need to use the generated keys. For instance, if you built a dimension on geographic data that had a state of New York and a city of New York, you could not use these as keys because they match exactly. Yet, rather than using generated keys, it would be simpler to change the name of the city to its proper name: New York City so it no longer exactly matches the state key. Anyway, you can click the Create button to save the new dimension.

In the main navigator, expand the new dimension and right-click the Hierarchies entry. Select all three of the defined levels and place them in the right-side selection box. Double-check to ensure they are in the correct order from most general down to detail: Year, Quarter, Month. You have now defined a dimension.

Action

If you skipped the first activity, run the BuildAllPowder file for this chapter. Run the Analytic Workspace Manager and connect to the database with your login. Right-click Analytic Workspaces, to create a new Analytic Workspace. Name it AllPowder and accept the default tablespace. Expand the new workspace and check out the various options.

Figure 9.14

The screenshot shows the 'Create Dimension' dialog box with the following details:

- Tab: Implementation Details
- Title: Specify General Dimension Information
- Name: DIM_SALE_DATE
- Short Label: Dim Sale Date
- Long Label: Dim Sale Date
- Description: Dim Sale Date
- Dimension Type: Time Dimension (highlighted with a red box)
- Buttons: Help, Create, Cancel

The problem is that the dimension does not contain any data. Each dimension ultimately needs to have the data loaded into it. That means you need a table or query that lists every year, quarter, and month for the data in the desired time period. To understand what data is needed, select the Attributes entry for the dimension. Figure 9.15 shows the list of items needed for the time dimension. To assign dates the appropriate category, Oracle uses the end date for the period—such as 31-DEC-2009 for the year 2009. To facilitate aggregations, the dimension also needs to know the number of days (span) in each interval. Each level is also given a short and long description. Technically, each description also can be translated into multiple languages. These data items have to be generated in a view or stored in a table before they can be loaded into the dimension. If you are using multiple languages, you ultimately will have little choice but to create a table to hold all of the data. Remember that you have to define these entries (as columns) for every month in your database.

A slightly easier approach is to create SQL views that extract the desired values directly from the Sale table. The approach uses SQL date functions to compute the desired columns. The time span entries are the most challenging because the que-

Action

Right-click the Dimension entry and choose the option to create a new dimension.

Name it: Dim_Sale_Date

Select the option to make it a Time dimension (not user).

Click the Levels tab and add: Sale_Year, Sale_Quarter, Sale_Month

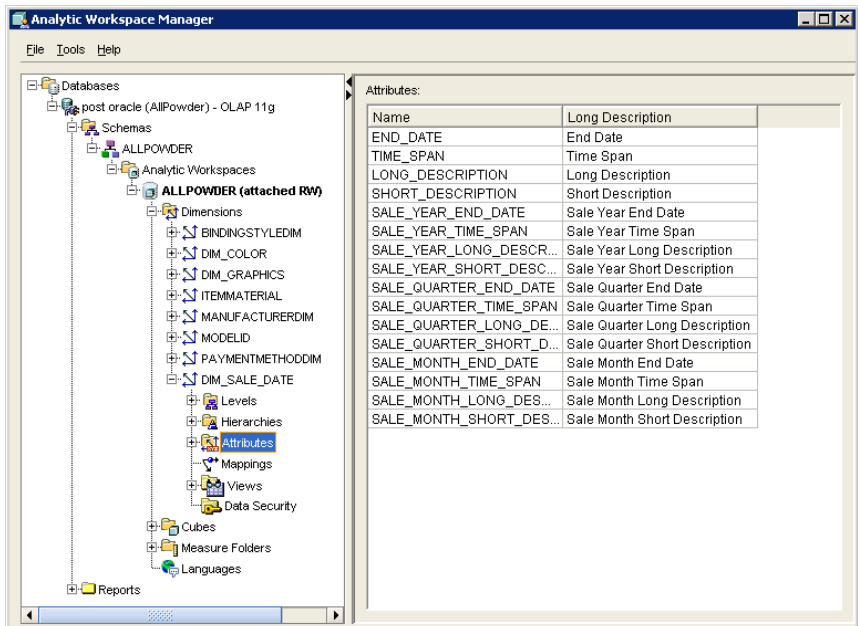
Click the Implementation Details tab and select to use Keys from Data Source, not the surrogate keys.

Expand the new dimension and right-click Hierarchies to add a new hierarchy.

Name: Sale_Calendar

Choose the levels in order: Year, Quarter, Month.

Figure 9.15



ries need to find the ending and starting dates for each period and subtract them to obtain the number of days.

You need to create three separate queries—one for each level in the hierarchy. The queries need to contain some matching data so they can be linked through “parent” values. For instance, the data for quarters needs to include the year value. Figure 9.16 shows all three queries. Use copy-and-paste to place them into a SQL Developer window. Run them to create the three views. You should use `SELECT *` from each view to verify that the data columns contain the correct values. Spend a couple of minutes to examine the queries to check out the date functions. Ultimately, if you need to create many views of this type, you should create new user-defined functions to handle the basic tasks with a single function. For example, finding the starting and ending dates for a time period would be a useful function that would greatly simplify the three queries.

Action

- Click the Attributes entry for the dimension to see the data needed for each level.
- Create three queries (year, quarter, month) to build the data needed for the dimension.
- Select the Mappings entry in the new time dimension.
- Select the Snowflake schema instead of Star.
- Drag the entries from the time views and drop them onto the matching rows in the dimension.
- Click the Apply button.
- Right-click the dimension in the navigator and choose Maintain... to load the data.
- Right-click the dimension and choose View Data.
- Verify that the hierarchy and data are correct.

Figure 9.16

```
CREATE or REPLACE VIEW SaleYears AS
SELECT DISTINCT To_Char(SaleDate,'YYYY') As YearID,
To_Char(SaleDate,'YYYY') As YearValue,
To_Date(Extract(Year from SaleDate) || '1231', 'YYYYMMDD') As YearEnd,
To_Date(Extract(Year from SaleDate) || '1231', 'YYYYMMDD')-To_Date(Extract(Year
from SaleDate)-1 || '1231', 'YYYYMMDD') As YearSpan,
TO_CHAR(SaleDate, 'YYYY') As YearDesc, TO_CHAR(SaleDate, 'YYYY') As
YearLongDesc
FROM SALE
ORDER BY YearValue;

CREATE or REPLACE VIEW SaleQuarters AS
SELECT DISTINCT To_Char(SaleDate,'YYYY') As YearID,
To_Char(SaleDate,'YYYYQ') As QuarterID,
To_Char(SaleDate, 'Q') As QuarterValue,
Add_Months(TO_DATE((Extract(Year from SaleDate)*100 + TO_CHAR(SaleDate, 'Q')*3
)*100 +1, 'YYYYMMDD'),1)-1 As QuarterEnd,
Add_Months(TO_DATE((Extract(Year from SaleDate)*100 + TO_CHAR(SaleDate, 'Q')*3
)*100 +1, 'YYYYMMDD'),1)
-Add_Months(TO_DATE((Extract(Year from SaleDate)*100 + TO_CHAR(SaleDate,
'Q')*3)*100 +1, 'YYYYMMDD'),-2) As QuarterSpan,
TO_CHAR(SaleDate,'YYYY-Q') As QuarterDesc,
TO_CHAR(SaleDate,'YYYY-Q') As QuarterLongDesc
FROM SALE
ORDER BY QuarterDesc;
```

```

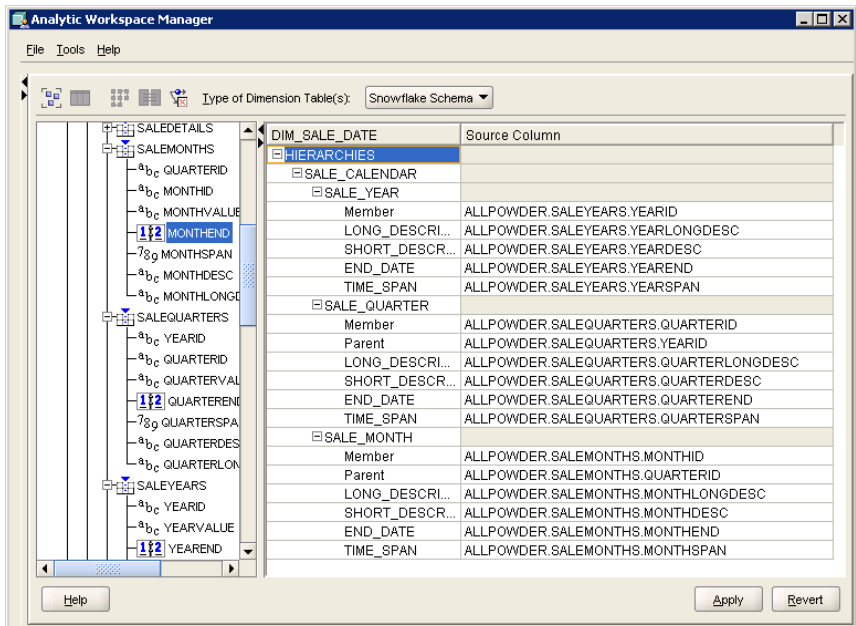
CREATE OR REPLACE VIEW SaleMonths AS
SELECT DISTINCT To_Char(SaleDate,'YYYYQ') As QuarterID,
To_Char(SaleDate,'YYYYMM') As MonthID,
To_Char(SaleDate,'MM') As MonthValue,
Add_Months(To_Date(Extract(Year from SaleDate)*10000+Extract(Month from SaleDate)
)*100+1,'YYYYMMDD'),1)-1 As MonthEnd,
Add_Months(To_Date(Extract(Year from SaleDate)*10000+Extract(Month from
SaleDate)*100+1,'YYYYMMDD'),1)
-To_Date(Extract(Year from SaleDate)*10000+Extract(Month from
SaleDate)*100+1,'YYYYMMDD') As MonthSpan,
TO_CHAR(SaleDate,'YYYY-MM') As MonthDesc,
TO_CHAR(SaleDate,'YYYY-Mon') As MonthLongDesc
from sale
ORDER BY MonthID;

```

The next step is to map the values from these new views to the time dimension you created. Select the Mappings entry under the sale dimension. Notice that the “Source Column” entries are empty. Select the Snowflake Schema (instead of Star) to indicate the data come from multiple views. Expand the AllPowder entry in the schema list, and expand the Views list. Find the three time views you just created and expand those to see the columns. Start with the Year and drag the view columns onto the appropriate location in the hierarchy source. For example, YearID in the SaleYears view becomes the Member value for the Sale_Year level. Figure 9.17 shows the final matching list. With the Snowflake schema, you need to indicate the parent for the two lower levels (Quarter and Month). Click the Apply button when you have finished.

To verify the dimension, in the main navigation window, right-click the new time dimension and choose the option to “Maintain...” the list. Accept the defaults to load the data from the queries into the dimension. When it has finished, right-click the dimension again and choose the View Data option. Figure 9.18 shows the sample time data expanded to display the months for a given year and quarter.

Figure 9.17



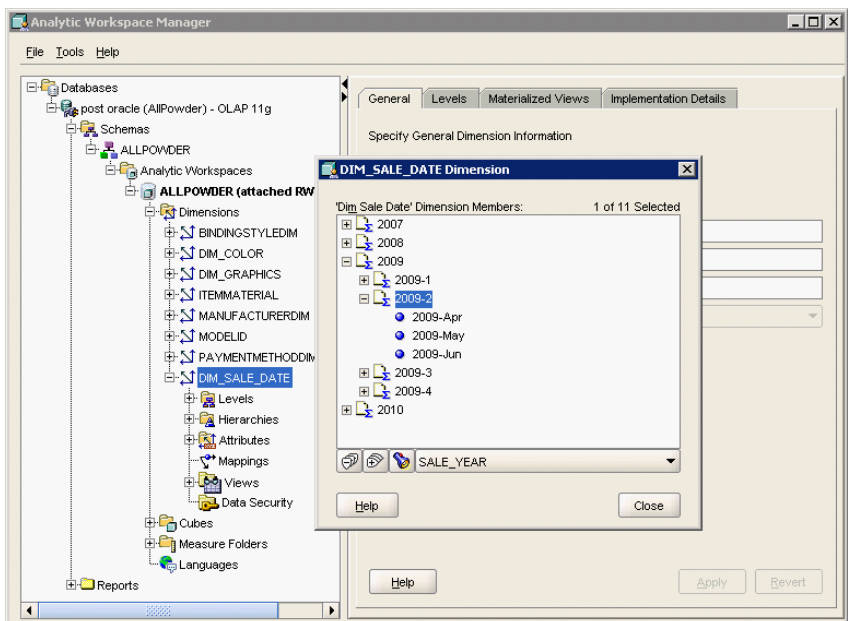
Now that you have created the hardest dimension, it should be straightforward to create the rest of them. The basic steps: (1) Create a dimension, assign levels, set it to use keys from the data source. (2) Define a hierarchy if needed. (3) Create a SQL View to obtain unique data if necessary. (4) Map the view columns to the dimension attributes. (5) Load and test the dimension.

Begin by creating a location dimension for State-City with two levels. Create two queries to extract unique data from the Sale table using the Ship-State and Ship-City columns. You could also create a generic City table (perhaps borrow it from Rolling Thunder). However, Figure 9.19 shows that the two views are straightforward to create—only the time dimensions are painful.

While you are at it, you will need to create views for Color, Graphics, and Item-Material. These queries are also listed in Figure 9.19. The key all of the queries is the DISTINCT statement to retrieve unique values for each dataset.

The location dimension needs a hierarchy with two levels. Similarly, you need to create a category-style dimension with two levels (category over style). This dimension can pull data directly from the ProductCategory table so it does not need a separate view. You then need several flat dimensions: Color, Graphics, Item-Material, BindingStyle, PaymentMethod, ModelID, and Manufacturer. The catch is that the current version of the cube browser does not handle flat dimensions correctly. So, you have to define every dimension with at least one level and then create a hierarchy using that single level.

Figure 9.18



Action

Create views for location (ShipState and ShipCity).

Create dimensions with hierarchies for location and product category style.

Create views to retrieve distinct data for Color, Graphics, and ItemMaterial.

Define dimensions for other attributes: Color, ItemMaterial, BindingStyle, ManufacturerID (with Name), Graphics, PaymentMethod, and ModelID.

Load data (Maintain) into each dimension and test them.

<pre>CREATE VIEW ShipStates AS SELECT DISTINCT ShipState FROM Sale ORDER BY ShipState;</pre>
<pre>CREATE VIEW ShipStateCities AS SELECT DISTINCT ShipState, ShipCity FROM Sale ORDER BY ShipState, ShipCity;</pre>
<pre>CREATE OR REPLACE VIEW ItemColorList AS SELECT DISTINCT COLOR As ItemColor FROM ITEMMODEL ORDER BY ItemColor;</pre>
<pre>CREATE OR REPLACE VIEW GraphicsList AS SELECT DISTINCT GraphicsName FROM ITEMMODEL ORDER BY GraphicsName;</pre>
<pre>CREATE OR REPLACE VIEW ItemMaterialList AS SELECT DISTINCT ItemMaterial FROM ITEMMODEL ORDER BY ItemMaterial;</pre>

Figure 9.19

The process is the same as for hierarchies. Name the dimension, add one level, and set it to use keys within the data source. Define a hierarchy using the single level. When you map columns to the dimension, use the views created or pull the columns directly from tables. Generally, you can use the same column for all three attribute entries: Member, Short Description, and Long Description. However, for Manufacturer, set the Member as the ManufacturerID and use Name for the descriptions. Load data into each dimension as you create it and view the resulting data to test it. Remember to use longer names to avoid conflicts with reserved words.

With the dimensions defined and tested, you can define the actual cube. A cube consists of a measure (fact value) and a collection of dimensions. Because the dimensions are already defined, you primarily need to create the measure. Right-click the Cube node in the navigator and choose the option to create a new one. Name it Sales_Main. Expand it and right-click the Measures entry to define a new

Figure 9.20

```
CREATE OR REPLACE VIEW SaleDetails AS
SELECT SaleDate, CustomerID, EmployeeID, ShipCity, ShipState,
PaymentMethod, To_Char(SaleDate, 'YYYY') As SaleYear,
To_Char(SaleDate, 'YYYYQ') As SaleQuarterID, To_Char(SaleDate,
'YYYYMM') As SaleMonthID, QuantitySold,
SalePrice*QuantitySold As SaleValue,
Inventory.SKU, Inventory.ModelID, ItemSize, QuantityOnHand,
ManufacturerID, Category As ProductCategory, Color As ColorName, Cost As
ItemCost, ModelYear, Graphics As GraphicsName, ItemMaterial, ListPrice, Style
As SkiStyle, SkillLevel, WeightMax, WeightMin, WaistWidth, EffectiveEdge,
BindingStyle, RentalRate
FROM SALE
INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
INNER JOIN Inventory ON SALEITEM.SKU=Inventory.SKU
INNER JOIN ItemModel ON Inventory.ModelID=ItemModel.ModelID;
```

measure. At this point, you simply need to enter its name: Sales_Value. Ultimately, you can define multiple measures for a cube. Typically, you use the cube tools to define calculated measures, such as year-to-date values and percentage comparisons. However, it is best to start with a simple cube and expand it later.

Similar to the way dimensions were created, you now need to create a view that retrieves the data needed for the cube. Figure 9.20 shows the query needed to retrieve the underlying data for the cube. Notice that it has to include all of the data to match the dimensions and it includes a calculation for the sales value measure. It is actually important to compute price times quantity within the SQL query. Do not attempt to perform this computation within the OLAP cube. The cube operates on aggregated data. SQL computations are applied one row at a time—which forces the multiplication to be performed before the sums are computed.

Again, similar to the way dimensions were created, you need to select the Mapping entry under the new cube. As shown in Figure 9.21 mapping has a couple of tricks. The measure (Sales Value) is straightforward. The flat dimensions are straightforward—simply drag the column from the SaleDetail view and drop it onto the matching dimension. The hierarchical dimensions also require you to drop the data column onto the dimension—just be sure to drop only the lowest detail entry. For instance, drop SaleMonthID onto the Sale_Month value. Then you have to define the join condition that matches the same SaleMonthID to the key value in the Calendar hierarchy view. You can drag and drop both values into

Action

Create a view to retrieve data from multiple tables: Sale, SaleItem, Inventory, ItemModel.

Create a new Cube and assign all of the dimensions.

Create a measure (Sales_Value).

Map the columns from the data view to the Cube values.

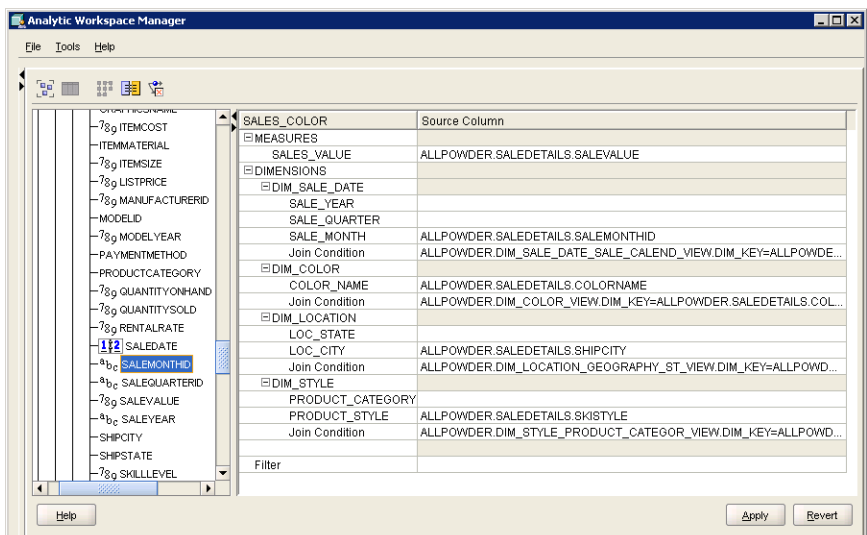
Create join conditions to match the cube data to the view columns.

Load the cube data through the Maintain option.

View the data in the View that was generated.

Test the cube browser.

Figure 9.21



the join box and it will automatically insert the equals sign. The join condition for the time hierarchy is:

```
ALLPOWDER.SALEDETAILS.SALEMONTHID
=ALLPOWDER.DIM_SALE_DATE_SALE_CALEND_VIEW.DIM_KEY
```

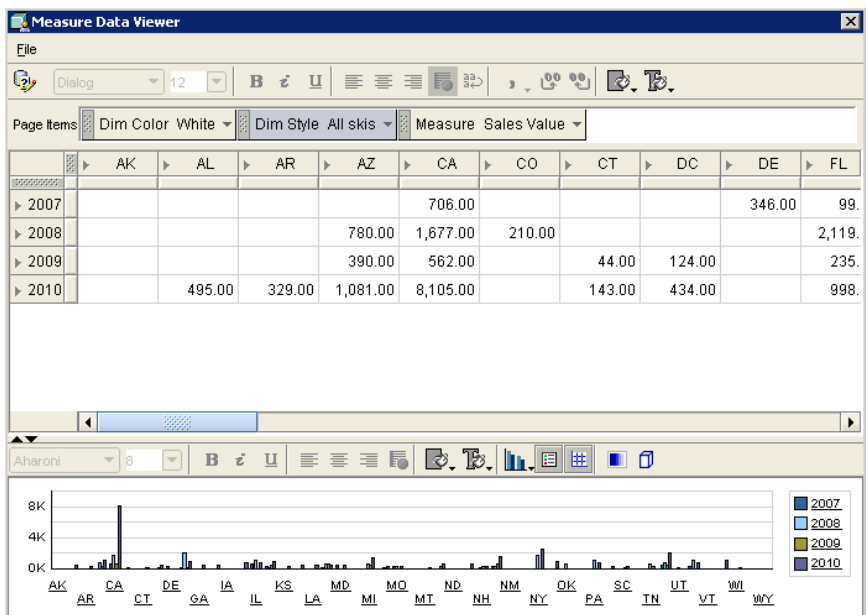
You need to repeat this step for the other two hierarchical dimensions. When the mapping is complete, click the Apply button to assign and save the values. If you receive error messages, delete the assignments and try again.

You need to load data into the cube. Actually, in a production environment, you will want to schedule loading so that new data is transferred on a regular basis. The tool has options to build an automated schedule. For now, right-click the new cube and choose the Maintain option to load the data. It might take a couple of minutes. When the process is finished, expand the View entry under the cube and select the Sales_Color_View that was created. Select the Data tab and you should be able to scroll through the data that was loaded.

If you use a limited number of dimensions, and all of the dimensions are created as hierarchies, the cube browser will work—to a limited extent. Right-click the cube and select the View Data option. Figure 9.22 shows the basic structure of the cube. You can drag the dimensions around (as columns or rows) to change the layout. As long as the dimensions include hierarchies, you can select specific data points in the drop-down lists for the page dimension. However, notice that the All option is not available for any of the dimensions—so you cannot rollup the cube data—the cube must always display data for only one selected value. Perhaps there is a way to change the dimension definitions to obtain the All option—perhaps not. With a small number of page dimensions, this limitation is tolerable, but if you try to include all of the dimensions, you end up with a very restricted list of data.

Hopefully, your cube will be interactive and you can use it to explore the data. But, even then, this cube browser is limited to running on a desktop—the tool cannot be accessed across the Web. Oracle also provides an Add-in for Microsoft

Figure 9.22



Excel that can connect to the database, retrieve the data, and use an interactive cube within the spreadsheet. But, it also runs on the desktop instead of the Web.

However, remember that Oracle has many tools. The APEX report system and the Business Intelligence Publisher can create static cubes used as the basis for charts and tables. Oracle Discoverer can create Web pages that host interactive cubes—but you need to use Discoverer to create the cubes. Discoverer tools are covered in the 10g Workbook. Eventually, Hyperion tools could probably be used to build Web-based cubes as well.

Introductory Data Analysis



Activity: Analyze Time-Series Data

Collecting data and browsing is useful for exploring the data and manually searching for patterns. Ultimately, managers want to use statistical and analytical tools to search for patterns automatically. Oracle has several built-in tools for analyzing data. Some of them run as special functions in SQL, others are based on Java programs. In large projects, it is possible to configure these tools to run programmatically, with minimal human involvement. However, to explore the data and to learn how the tools work, it is

often easier to begin with a graphical tool that can help you set up and run the analyses. Oracle Data Miner provides access to several common analytical tools. It will be used in the following sections. Unfortunately, it does not currently support time series analysis. Because time series analysis is useful in many problems, it is worth taking a few minutes to learn some of the tools available within Oracle. For more extensive time series analysis, you might want to find other tools. A couple of open-source tools are available for free download on the Web. These are explored in greater detail in the Data Mining textbook: <http://www.JerryPost.com/Books/DMBook>.

Oracle has some powerful extensions within SQL to analyze data. In many cases, you can accomplish useful analyses without ever leaving SQL. In particular, the extensions to analyze data over time are useful in many problems. Recall that basic SQL allows you to perform calculations and reference data on a single row at a time. Aggregate functions (Sum and Average) operate across multiple rows and can create subtotals for groups. But, what if you have a set of data over time,

Action

Create a view to compute the total sales per month using TO_CHAR(SaleDate, 'YYYY-MM') to get the month.

Create a query to display the Month, SalesValue, and the Sales Value from the prior month using the LAG function.

Create a query to compute the 3-month moving average of the sales.

Copy the data to Excel and draw a chart showing sales and the 3-month moving average.

Figure 9.23

```
CREATE VIEW MonthlySales AS
SELECT To_Char(SaleDate,'YYYY-MM') As SaleMonth,
Sum(SalePrice*QuantitySold) As SalesValue
FROM SALE
INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
GROUP BY To_Char(SaleDate,'YYYY-MM')
ORDER BY SaleMonth;
```

```
SELECT SaleMonth, SalesValue, LAG(SalesValue, 1, 0)
  OVER (PARTITION BY 1 ORDER BY SaleMonth) AS PriorSales
FROM MonthlySales
ORDER BY SaleMonth;
```

Figure 9.24

such as monthly sales data, and you want to compare the value for one month with the data from the prior month? You need a way to operate on data in multiple rows.

Begin by creating a view to retrieve sales data by month. Figure 9.23 shows the query—using the `To_Char` function to format the date as year and month and using a simple `GROUP BY` clause to compute the subtotals. Run the query, or select all from the new view to verify that the query computes total sales by month. Technically, you do not have to create this view—you could compute the sums later within the following queries—but it is easier to see the queries if you build them in pieces.

Now the fun part: test a couple of the Oracle analytic SQL statements. A common problem is the need to compare sales in one time period with sales in the prior period—for example, to compute growth rates. Figure 9.24 shows how this query is built using the analytic `LAG` function and the `OVER` clause to establish a data window. The `PARTITION BY` statement is optional, but it can be useful for complex problems. For example, you might need to compare prior sales by time and `ProductCategory`. Each category is separate, so you would use `PARTITION BY ProductCategory` to restart the time window for each new category value. When you do not have this additional column, you can partition by a constant or you can just drop the `PARTITION BY` clause.

In terms of time series analysis, a useful function is the ability to compute moving averages. A moving average uses a sliding window to compute the average of fixed number of rows. For example, an `MA3` is a three-period moving average that averages three contiguous values. Begin by averaging items 1, 2, and 3; then slide to numbers 2, 3, and 4; and repeat the process to the end. The analytic window method can accomplish this task with a single query. Figure 9.25 shows the syntax for the sample sales data. Notice that it does not use the partition clause, but it could be added if necessary. Beyond the `Avg` function, the key lies in specifying the “`ROWS 2 PRECEDING`” clause. This statement causes the system to use the two preceding rows along with the current row to compute the average of all three values.

If you have one of the Oracle graphics packages installed, you could build a report to chart the results. However, it is straightforward to copy the results from the SQL Developer output and paste them into Excel. Figure 9.26 shows the resulting chart. The moving average is a smoother chart because it averages out some of the

Figure 9.25

```
SELECT SaleMonth, SalesValue, Avg(SalesValue)
  OVER (ORDER BY SaleMonth
        ROWS 2 PRECEDING) AS Avg3
FROM MonthlySales
ORDER BY SaleMonth;
```

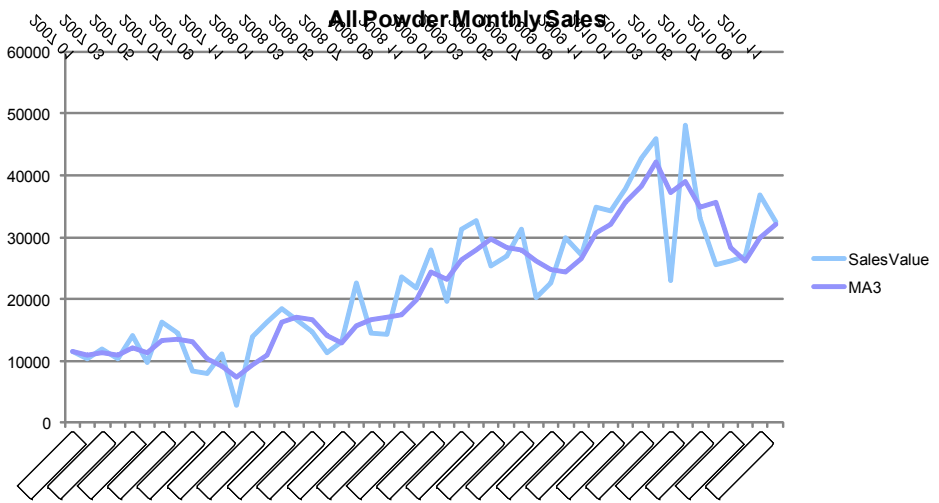


Figure 9.26

detail variations. Hence, moving averages are often called “smoothing” functions. More sophisticated time series analytical tools use moving averages and lagged values to estimate the effects of trends and seasonal components. Still, the ability to derive these values with an SQL query is useful.



Activity: Classify Sales Data

Oracle Data Miner (ODM) provides a graphical interface to several common data mining tools. It is worth examining a couple of them just so you get a feel for how Data Miner works. First, be sure you have downloaded the Data Miner tool. When you run ODM the first time, you will have to specify the connection to the database. Use your standard connection with the server name, port (1521), and SID (orcl). The goal of this section is to set up and run a basic classification problem to create a decision tree. The sample data for this case is not particularly interesting, but it does illustrate the process. Real-world data has many more details and potentially useful results.

As with most data mining problems, you first need to put the data in the proper format for the tool. Classification tools require a categorical variable as the target or forecast. In this example, you want to know if some dimensions lead to higher or lower sales. As shown in Figure 9.27, first create a view that computes the subtotal of sales for various dimensions: ShipState, ColorName, ProductCategory, SkiStyle, PaymentMethod, and SaleYear. You should look through the data to get a feel for the type of data. Computing the average (278) and standard deviation (182) can be useful to indicate what values might be considered “high” or “low.”

Action

- Download, install, and start Oracle Data Miner.
- Set the initial database connection.
- Create a SaleGroups view to compute subtotals.
- Create a SaleTreeData view to classify the sales totals into four categories.
- Run the Classification/Decision Tree analysis.
- Set the target as the new classification column.
- Explore the results.

```
CREATE VIEW SaleGroups AS
SELECT ShipState, ColorName, ProductCategory, SkiStyle,
PaymentMethod, SaleYear, SUM(SaleValue) As SalesTotal
FROM SaleDetails
GROUP BY ShipState, ColorName, ProductCategory, SkiStyle,
PaymentMethod, SaleYear;
```

Figure 9.27

Using the average and the standard deviation, and consulting with the managers, you can define categories that might be interesting. Figure 9.28 shows the query to create a new view that uses a CASE statement to assign category labels to various data ranges. Of course, the company would like to know if any of the variables lead to high sales, but it would also be useful to know if some categories or colors lead to particularly low sales so those items could be cut from the inventory. ODM can build a decision tree that will identify any significant data points that lead to the various categories.

If necessary, start ODM and establish the data connection. In the main menu, select the Activity / Build option. As the Function Type, choose Classification, and then the Decision Tree algorithm. You can read the data mining textbook to learn the difference between the various algorithms, but the common choices are Decision Tree or Naïve Bayes. If necessary, choose the schema that holds your data (AllPowder). Then select the table or view that you created (SaleTreeData). Because this data is based on several groups, no separate key was created, so select the option for Compound or None. All of the columns are selected by default. You do not need the SalesTotal value for this analysis, so uncheck it. As shown in Figure 9.29, on the next screen, select the SalesCat column as the Target—because that is the value you want the system to predict. The preferred target value is “High” because higher sales lead to more profits. Accept the defaults and finish the wizard.

After a few seconds, ODM presents the basic output page. Scroll to the Build section and click the “Result” link. Figure 9.30 shows the summary results with two nodes in the decision tree. Overall the results are somewhat boring because both nodes are based only on the year and both lead to low sales. Again, more realistic data would often lead to more interesting results. However, this type of simple result has some value because it indicates that the selected dimensions do not have a significant effect on the sales classification. Hence, the managers need to look at other factors that might play a role. Sometimes eliminating dimensions is as important as finding critical ones.

Figure 9.28

```
CREATE VIEW SaleTreeData AS
SELECT ShipState, ColorName, ProductCategory, SkiStyle,
PaymentMethod, SaleYear, SalesTotal,
CASE
  WHEN SalesTotal<100 THEN 'Very Low'
  WHEN SalesTotal BETWEEN 100 AND 300 THEN 'Low'
  WHEN SalesTotal BETWEEN 300.01 AND 500 THEN 'Average'
  WHEN SalesTotal>500 THEN 'High'
END As SalesCat
FROM SaleGroups;
```

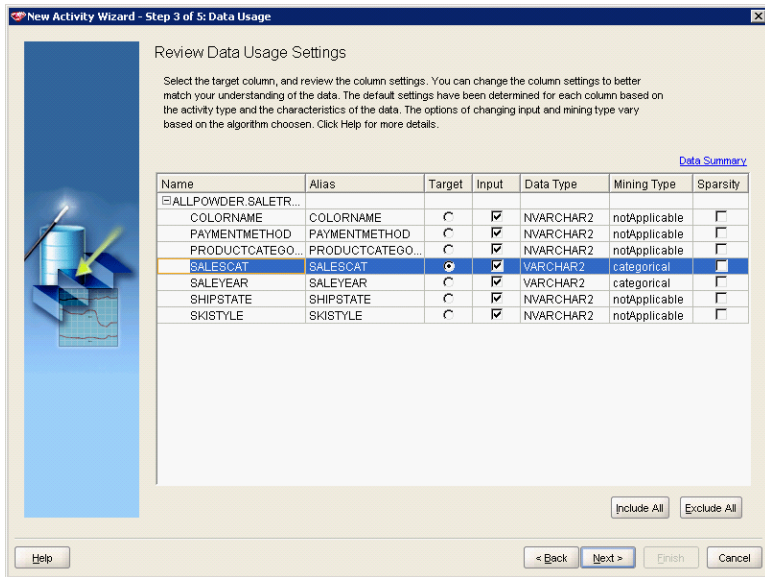
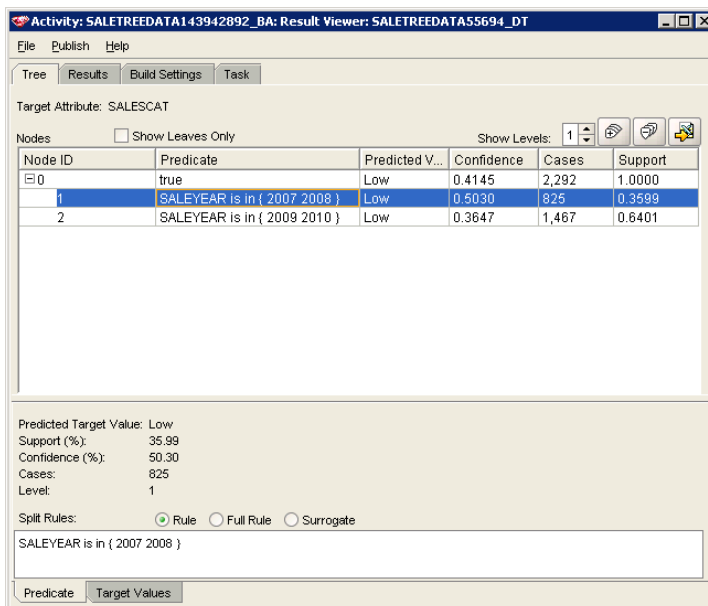


Figure 9.29

The results contain additional information. If you select the Target Values tab at the bottom you can see a chart of the number of observations in each category—which is somewhat bleak because it consists primarily of low sales levels. You should also examine the measurement statistics to see how good the model tree is at prediction. Select the Results tab at the top of the screen. Expand the Test Metrics tab and select the item within that node. Click the View button to see the prediction data. Select the Accuracy tab in the new window and click the button for More Detail. Figure 9.31 shows the number of items that the model predicts correctly. First, notice that it does not predict any values for the Average or Very

Figure 9.30



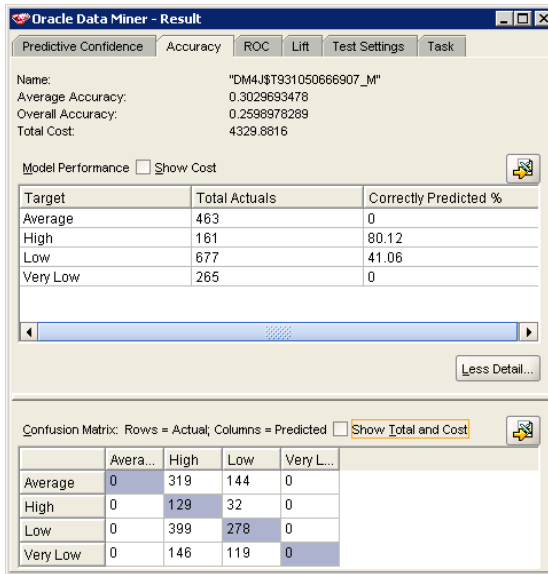


Figure 9.31

Low categories. Overall, the model is marginal, and you should look at using different dimensions, a different model, or getting better data. If you are curious and have enough time, you can rerun the analysis using the Naïve Bayes model. It has slightly better results, but they are still unexciting.



Activity: Analyze Data with Regression

Linear regression is a tool that is relatively easy to use and is supported by a variety of platforms. Oracle Data Miner has a relatively standard multiple regression tool. Multiple regression is a powerful statistical tool used for many research purposes. One challenge to using it is that it works only with numeric data. If you have categorical data (such as gender or product style), you can recode the data to numbers instead of text.

Regression is often used to analyze economic data—particularly any background information you have about customers. Because the database does not include detailed customer data, you can use government data based on the customer location. Federal data on personal income and state production (GDP) are readily available from the Bureau of Economic Analysis (BEA). You can begin your search for data at www.FedStats.gov, or go directly to the interactive tables from the BEA at <http://www.bea.gov/regional/index.htm#gsp>. Someday you should take a look at those Web sites because they contain an enormous amount of useful data. However, you can also pull the data from the CSV file on this book's Web site ([StateDemogBEA2008.csv](#)).

Action

- Create a new table to hold the demographic data.
- Read the data from the CSV file into the new table.
- Create a view that computes total sales by state for 2008.
- Create a view that combines the state sales and demographic data.
- In ODM, select Action/Build and choose the Regression: Multiple Regression (GLM) model.
- Use SPI2008, GDP2008, POP2008, and Sales2008 in the model.
- Set Sales2008 as the target.
- Run the analysis and explore the results.

```

CREATE TABLE StateDemo
(
  FIPS NUMBER(5,0),
  StateCode NVARCHAR2(10),
  AreaName NVARCHAR2(100),
  DPI2008 NUMBER(10,0),
  GDP2008 NUMBER(10,0),
  POP2008 NUMBER(10,0),
  CONSTRAINT pk_StateDemo PRIMARY KEY (FIPS)
)

```

Figure 9.32

You need to import the CSV data into a new table in Oracle, so first you have to create the table to hold the economic data. Figure 9.32 shows the command to create the table. Check the data in the CSV file and you will see that it includes a FIPS column—which is assigned by the government to uniquely identify every region. Unfortunately, the existing data in the database does not have a FIPS number, so the CSV file also contains the two-letter postal code for each state—which can be used to match the values in the database.

The shortest method to import data into Oracle is to define an external table and use a `INSERT...INTO` statement to transfer the data from the external CSV file into the database table. Figure 9.33 shows the command needed to define the external table. Be sure to modify the folder directory to point to the exact location of your file. After you run the command to create this external connection, you can use a simple `INSERT` statement to transfer the data:

```

INSERT INTO StateDemo(FIPS, StateCode, AreaName, DPI2008,
GDP2008, POP2008)
SELECT FIPS, StateCode, AreaName, DPI2008, GDP2008,
POP2008
FROM StateDemoCSV;

```

Figure 9.33

```

CREATE OR REPLACE directory csv_dir AS 'C:\Database\';
CREATE TABLE StateDemoCSV
(
  FIPS NUMBER(5,0),
  StateCode NVARCHAR2(10),
  AreaName NVARCHAR2(100),
  DPI2008 NUMBER(10,0),
  GDP2008 NUMBER(10,0),
  POP2008 NUMBER(10,0)
)
organization external (
  default directory csv_dir
  access parameters (
    records delimited by newline
    fields terminated by ','
    optionally enclosed by '"' ltrim
    missing field values are null
  )
  location ('StateDemogBEA2008.csv')
)
reject limit unlimited;

```



```
CREATE VIEW SalesByState AS
SELECT ShipState, Sum(SaleValue) As StateTotal
FROM SaleDetails
WHERE SaleYear=2010 GROUP BY ShipState;
```

Figure 9.34

You should issue a DROP TABLE command to release the external file. Once the new data is in the database, you can create views to organize it in a form that can be used by the regression tool.

Create a new view that computes the total sales by state. If you have completed the earlier labs in this chapter, you should already have the SaleDetails table that computes price by quantity and includes the ShipState. Figure 9.34 shows the simple query that uses that view. If you do not have that view, you can modify the query to use the Sale and SaleItem tables.

Finally, you can create the view needed by the regression tool by combining the sales data with the demographic data. Because both views contain the state postal code, it can be used to join the views. Using an INNER JOIN, the resulting view will contain only data on states that are included in the sales database. Figure 9.35 shows the query.

As usual, getting the data and organizing it for the tool is the hardest part of the analysis. At this point, you can run the Data Miner, choose the tool, select the view, and run the tool with a few clicks of the mouse. Start ODM, choose Activity / Build. Select the Regression tool and pick the Multiple Regression (GLM) method. For the data view, select the StateEconSales view you just created. The tool will not accept the character state code as a key, so choose the option for None. In the list of columns, deselect the ShipState column. On the target screen, select StateTotal as the target and pick the three economic values as input values. Finish the wizard with the default options and wait for the results to run. Scroll to the Build section and click the Results link.

The initial regression results page presents a table of summary statistics. Typically, the most important value is the R-squared statistic which shows the percentage of variation in the dependent variable that is explained by the input variables. At 72 percent, this value is relatively high for a model with only 47 observations. From a business perspective, the coefficients of the equation are the most valuable results. Figure 9.36 shows the values on the Coefficients tab. First, the values seem small because population and disposable personal income (DPI) are large numbers (millions and billions). You could divide by six zeros if you want to talk about millions instead of raw numbers. For instance, the population coefficient states that the store receives an extra \$83.26 in sales for an increase of one million people. The fact that the population and DPI coefficients are positive indicates that the company should target sales to larger states with higher income levels.

Figure 9.35

```
CREATE VIEW StateEconSales AS
SELECT ShipState, DPI2008, GDP2008, POP2008, StateTotal
FROM SALESBYSTATE
INNER JOIN StateDemo ON SalesByState.ShipState=StateDemo.StateCode;
```

Activity: STATECONSALES232932128_BA: Result Viewer: STATECONSALE44626_GL

File Help

Global Statistics Coefficients Results Build Settings Task

Target Attribute: STATETOTAL

Show Intercept Row

Coefficients Fetch Size: 100 Refresh Filter

Attribute Name	Value	Coefficient	Standard Error	Wald Chi-Squa...	Pr > Chi-Square	Standardized ...	L
POP2008		0.0000832577	0.000745	0.111799	0.91222	0.060745	-C
DPI2008		0.0000683444	0.000049	1.384223	0.183213	1.717073	-C
GDP2008		-0.0298581268	0.037441	-0.797467	0.435574	-0.936374	-C

Sort coefficients based on absolute values

Figure 9.36

The coefficient data presents an additional measure of the model validity. The T-ratios (coefficient divided by standard deviation) are shown in the Wald statistic column. In all three cases, they are a little low. Ideally, all coefficients values should be greater than 2. But, remember that the model was estimated with only 47 observations.

Analyzing regression results, and learning to identify problems, takes practice. A strong background in statistics or econometrics also helps. Still, the basic results are relatively easy to understand and can be valuable in many situations. It also often helps to chart the data and draw the linear regression line onto the data points to show trends.



Activity: Analyze Association Rules for Market Baskets

Association Rules, particularly market basket analysis is one of the classic data mining tools. In the sales context of a market basket, the question is to determine which items are commonly purchased together. The method uses basic probability to find a set of rules that might or might not provide insights into the sales process. The classic apriori algorithm (used by Data Miner) relies on two calculations to limit its searches. Support is the percentage of baskets (or Sales) that contain the items under consideration. So support is just the relative frequency definition of probability. Confidence is the conditional probability that the second item exists in a basket given that the first one is already there: $P(B|A) = P(A \text{ and } B)/P(A)$. The confidence measure is directional and the system can test both whether item B appears if A exists, and whether A appears if B exists.

Implementations of the apriori algorithm typically use one of two methods to organize the data. (1) List each Sale/Basket as a single row and include

Action

- Create a query that lists the SaleID and a concatenation of Category and Style.
- Create a new Activity / Build in Oracle Data Miner.
- Choose Association Rules: Apriori algorithm.
- Choose the new view as the data source and set the ItemCategory as the ItemID.
- For TransactionID, select SaleID.
- Click the Results link then the Get Results button.
- Explore the results.

```

CREATE VIEW CategoryItemsSold AS
SELECT SaleID, ItemModel.Category || N '-' || ItemModel.Style As ItemCat
FROM SALEITEM
INNER JOIN Inventory ON SaleItem.SKU=INVENTORY.SKU
INNER JOIN ItemModel ON Inventory.ModelID=ITEMMODEL.MODELID
ORDER BY SaleID;

```

Figure 9.37

every item in the basket on that row. (2) The relational approach similar to the SaleItem table which contains a column (SaleID) to identify the Sale/Basket, and a second column to list one Item within the basket. Because of the relational background, ODM uses the second, relational approach. Hence, it is straightforward to create a view that includes the SaleID and an identifier for the item.

One of the most important steps in association analysis is to identify the level of detail needed. Consider the SaleItem table, with key columns: SaleID and SKU. Remember that SKU represents not only a type of item (Model), but also its size. If you run analysis at the SKU level, you would be searching for rules that an item of a specific size might be related to some other item of a specific size and color. Does the size, color, and so on, really matter to the analysis? Perhaps. But working at that level of detail requires good data and a lot of patience. For now, it makes sense to start at a much higher level: product category and style. For example, are people who buy downhill skis likely to buy clothes?

Figure 9.38

Activity: CATEGORYITEMSOLD645882622_BA: Result Viewer: CATEGORYITEMS61457_AS

File Publish Help

Rules Build Settings Task

Statistics:

Total Rules: 30

Get Rules

Rule Id	If (condition)	Then (association)	Confidence (...)	Support (%)
17	ITEMCAT.Ski-Freestyle= 1	ITEMCAT.Clothes= 1	48.0702	9.7857
19	ITEMCAT.Board-Half-Pipe= 1	ITEMCAT.Clothes= 1	46.2209	11.3571
23	ITEMCAT.Board-Extreme Board= 1	ITEMCAT.Clothes= 1	45.7143	6.8571
27	ITEMCAT.Ski-Cross-Country-Tradition...	ITEMCAT.Clothes= 1	44.8980	9.4286
2	ITEMCAT.Boots= 1	ITEMCAT.Clothes= 1	44.8148	17.2857
25	ITEMCAT.Ski-Cross-Country-Skate= 1	ITEMCAT.Clothes= 1	43.4615	8.0714
21	ITEMCAT.Ski-Back-Country= 1	ITEMCAT.Clothes= 1	41.6185	5.1429
3	ITEMCAT.Board-Ride= 1	ITEMCAT.Boots= 1	41.3043	5.4286
11	ITEMCAT.Ski-Cross-Country-Skate= 1	ITEMCAT.Boots= 1	41.1538	7.6429
15	ITEMCAT.Board-Ride= 1	ITEMCAT.Clothes= 1	39.6739	5.2143
9	ITEMCAT.Board-Extreme Board= 1	ITEMCAT.Boots= 1	39.0476	5.8571

Rule Detail

IF
ITEMCAT.Ski-Freestyle= 1

THEN
ITEMCAT.Clothes= 1

Confidence (%)=48.07
Support (%)=9.79

As usual, you need to create a query to organize the data. The goal is to create a view with columns for SaleID and the product category and style. Figure 9.37 shows the query. The only trick is to use string concatenation to join the category name to the style. Note that the tool only needs to know the presence of each item, the quantity and price do not matter.

With the data defined, the process is straightforward. In ODM, use the menu Activity / Build to create a new model. Choose the Association Rules and the Apriori algorithm (it is probably the only choice). For Table/View, select the newly created CategoryItemsSold view. Pick the ItemCat column as the Item identifier. If you had used an ID value for that choice, you could also add a second table as a lookup to find the associated name. You can skip the second option for this example. On the next screen, select the SaleID as the transaction identifier. Each SaleID represents a unique basket or purchase.

By default, ODM starts the apriori algorithm with minimum support of 5 and minimum confidence of 10. It also limits the number of items compared together to just 3. These values will work reasonably well for this sample problem. In other cases, you will have to edit them using the Advanced Settings button—or by altering the options after the initial run. The values are critical to the apriori algorithm because the algorithm uses them as cutoff values for its search. Typically, you can start with these values. If the system returns too few rules, you can reduce the confidence number. If the system returns too many rules (either too slow to process or too hard to read), increase the confidence and support numbers.

After the system runs, you can click the Results link. However, the Results page is largely blank. Look for the number of rows found near the top of the page. As shown in Figure 9.38, click the Get Results button to retrieve the rules. The data is stored in a table and the viewer can sort or filter the rules. By default, the rules are sorted in descending order of confidence.

To illustrate the results, consider the first rule: Ski-Freestyle => Clothes with a confidence of 48.1 and support of 9.8. Almost half the people who purchased freestyle skis also purchased clothes at the same time. In fact, all of the first seven rules show that people who purchased equipment tended to purchase clothes. From a management and marketing perspective these rules imply that the store should offer discounts on equipment, and then make up the profits on the clothing that people will buy at the same time. Of course, business situations can be more complex—perhaps people are buying the clothing because the prices are so low now, and increasing the prices might significantly alter the buying behavior. The application of the rules depends on understanding the actual business details. Still, the rules can provide interesting ideas that might have been hidden from the managers. Plus, the tool is relatively easy to configure and run, so it should be easy to set up and let the managers look through the results.

Exercises



Crystal Tigers

The Crystal Tigers club does not have a huge amount of data to analyze within the organization. However, the club members are interested in comparing their service data and the organizations they work with to see if they are serving the needs of the community. Periodically, they survey people in the surrounding areas to determine if they have heard of the club, if they know what charities the club supports, and their overall opinion of the club. In the process, they also ask citizens

about the events and problems that most affect their lives. A substantial part of the survey is a listing of support organizations with which the club is considering partnering. Crystal Tigers has collected this survey data every six months for the last three years, and they get several hundred responses each time. All of the data is stored in Excel spreadsheets.

1. Create two sample spreadsheets with the survey data. Create tables in Oracle to hold the normalized data. Write the SQL statements to transfer the data. Build this code into a form and button that will automate the transfer.
2. Create a query and a Discoverer worksheet that will enable managers to analyze the survey data.
3. Create a worksheet that will enable managers to analyze the existing club service data. Use two possible fact fields: hours worked and money raised. Include all of the dimensions you think managers might need.
4. Do a time series analysis of the money raised. Managers are particularly interested in trends and in identifying the months that raise the most money.
5. Assume you have data on money raised for several years (make up monthly totals if necessary). Obtain personal income data for your state or metropolitan area over those years and see if the income level is correlated with the money raised.



Capitol Artists

The managers of Capitol Artists are primarily interested in identifying the best employees and the most profitable customers. The job-tracking system ultimately generates a considerable amount of data—at the hourly and daily levels. Note that all employee tasks are supposed to be recorded in the system based on the client, job, and task involved. The firm has considerable information on clients, including a size classification (tiny, small, medium, and large), and type of company (such as printing shop, marketing, retail, and medical). This additional client information is currently stored in a spreadsheet, with one page devoted to each client.

1. Create three sample client worksheets with sample data. Modify the tables as needed to handle this new data. Create a form that will enable a clerk to find the worksheet and transfer the data to Oracle.
2. Create a Discoverer worksheet that will enable managers to analyze the hours worked and revenue generated by employees, day of week, client, client size, and so on.
3. Create a Discoverer worksheet that compares employees based on billable hours by day during the past month.
4. Assume that you have approximate sales numbers representing the size of each of the clients (make up the data). Create a categorical variable for the client industry (for example, 1 = printing shop, 2 = marketing, and so on). Perform a regression to see if the client size or industry influence the amount of sales revenue Capitol Artists generates.
5. Analyze the data with the association rules to see if there are relationships between the items purchased.



Offshore Speed

Inventory control is critical for Offshore Speed because it has to stock thousands of small parts for different engines and drives. All of these parts are grouped into categories in terms of the manufacturer and the location within the engine or boat. Lately, the owners think there has been an increased demand for oil pump impellers, but they are not certain because there are several different brands. They also suspect that sales of electronic navigation devices have tapered off. Although they have the sales data available, they are not sure how to analyze and compare it. Of course, the sales data for the past three years is stored in Excel spreadsheets. One sheet for each month of sales, and each line contains a sale number, date, part number, quantity, and price. Unfortunately, the part numbers do not match the new ones entered into the database. However, there is a separate spreadsheet that maps the two numbers. The first column lists the old number and the second column contains the new number.

1. Create at least two sample spreadsheets for the older sales, and the spreadsheet that maps the old numbers to the new ones. Create a form that can be used by a clerk to pick a spreadsheet and import the data into the new database.
2. Create a Discoverer worksheet that will enable managers to analyze sales by category, manufacturer, and time. Note that category should be a hierarchy. For example, managers might want to see detailed parts, or just the parts that are used in engines (or drives, or steering, and so on).
3. Create a worksheet chart that analyzes sales of the major categories over time based on monthly sales.
4. For some reason, an employee of the company has kept records of the weather for the last three years. She has a spreadsheet that contains the date, the amount of rain on that day, and the high temperature for the day. Create a regression to see if there is a relationship between the weather and your sales. (Make up some sample weather data, or find it on the Internet for your area.)
5. If you have access to software that performs association or market basket analysis, this case would be a good application to see what types of parts might be purchased together.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks.

1. Identify at least one primary fact attribute that managers would want to track, along with several dimensions. Create the query and Discoverer worksheet to analyze the data.
2. Identify any data that could be analyzed over time, and create a Discoverer chart and an Excel spreadsheet to forecast the data.
3. Identify any data that could benefit from market basket or association analysis. If you have access to the software, create the queries and analyze sample data.

4. Identify any data that could benefit from geographic analysis. If you have access to the software, create the queries and analyze sample data.
5. Identify any correlations or regression analysis that might help managers better understand the operations and effects of various attributes. If possible, collect sample data and analyze it.

Chapter 10

Database Administration

Chapter Outline

Database Administration Tasks, 251

Case: All Powder Board and Ski Shop, 252

Lab Exercise, 253

All Powder Board and Ski Shop, 253

Security and Privacy, 261

Exercises, 268

Final Project, 269

Objectives

- Evaluate and improve the application performance.
- Establish backup and recovery methods and plans.
- Install simple security controls to provide basic protection of the data.
- Protect the forms, reports, and code from unauthorized changes.
- Protect the data with user-level security controls.

Database Administration Tasks

One of the powerful features heavily pushed by Oracle is its performance under a heavy load of users. However, obtaining this performance often requires detailed work by the database administrator. Oracle provides a variety of options to tune the storage, query execution, and other control features of the database. Additionally, these options change over time as Oracle finds new ways to improve performance. The job of an Oracle DBA is not easy and requires constant learning. However, a good DBA can make a tremendous difference in the database performance. Fortunately, Oracle is beginning to include more automated tools to help analyze the database and queries and recommend improvements.

Every DBMS maintains an internal list of all of the database objects, such as table, query, and report names. The SQL standard proposes a common method to obtain these names from the `Information_Schema`. However, Oracle has never implemented this interface. Instead, Oracle has many internal views to retrieve metadata. For example, you can use the `describe` command to list the columns in a table (such as: `desc Sale`). You can also use the Web-based administration tool to graphically explore the database. As a side note, an interesting open-source project has been developed to create the standard `Information_Schema` within Oracle. If you want to use these standardized views, you can download them from <https://sourceforge.net/projects/ora-info-schema/>.

Oracle stores a considerable amount of metadata within system tables and system views. To make them easier to use, Oracle defines several synonyms that retrieve data from these static data dictionary views. Figure 10.1 shows some of the commonly used synonyms. Any of the three prefixes can be used with the `command synonym` to specify which level of objects you want to see. The `All` prefix lists all tables in any schema that you have rights to read. The `DBA` prefix specifies objects with permissions for the DBA. The `USER` prefix lists objects within the current schema. Each schema returns different columns. You can read the Oracle documentation to identify the columns, or simply run a short query: `SELECT * FROM USER_TAB_COLUMNS WHERE rownum<5`; The `rownum` constraint reduces the amount of data displayed to a small number of rows so you do not have to wait for thousands of rows of data. The sample query in Figure 10.1 provides a list of the tables in the schema along with the percentage of space remaining that is allocated to each table.

Figure 10.1

Prefixes	Synonym	Description
ALL_	CONSTRAINTS	Table constraints and keys
	IND_COLS	Indexed columns
	MVIEWS	Materialized views
	SEQUENCES	Sequences
DBA_	SYNONYMS	Synonyms
	TAB_COLUMNS	Table columns
USER_	TABLES	Tables
	TRIGGER_COLS	Trigger columns
	TRIGGERS	Triggers
	TYPES	User-defined data types
	USERS	Users
	VIEWS	Views (saved queries)
SELECT Table_Name, Pct_Free FROM USER_TABLES		

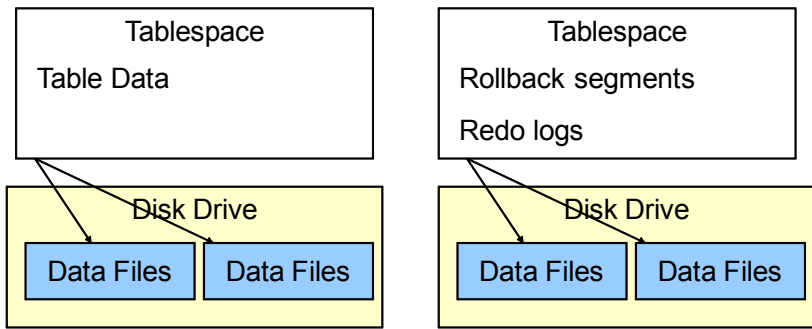


Figure 10.2

Performance is always a tricky issue in a DBMS. Small tables with a limited number of joins and a handful of simultaneous users rarely encounter performance problems. Also, with hardware improvements, performance improvements simply come down to “buy more processors and disk drives.” However, since one of Oracle’s strengths is its ability to handle huge amounts of data, you will encounter some databases that will need changes to improve performance. To understand some of the performance controls in Oracle, you need to be aware of how Oracle stores data on the file system. At the base level, the DBA allocates data files on disk drives. If you are not using a RAID system to automatically store data on multiple drives, you can accomplish a similar effect by creating separate storage files on different disk drives. Tablespaces are logical folders that can utilize multiple data files. Tables are assigned a specific tablespace to store the data. Oracle also uses rollback segments and redo logs to handle transactions and other situations where device failure might cause serious problems. As indicated in Figure 10.2, you get a substantial gain in performance if you store the table data and rollback segments in separate tablespaces on different drives. Two drives spinning independently means (1) the computer can write the data simultaneously, and (2) there is less chance of a loss in the event of a hardware failure. Oracle provides a tablespace map tool to help DBAs monitor the current storage allocation.

Backup and recovery are critical aspects to a database designed to handle thousands of users and processes running at once. In many cases, the database must run 24-7, so you cannot stop it to make a backup copy. Consequently, even while you are backing up data, new rows are being added and data is changing. Oracle has systems to protect all of this data, but if there is a hardware crash, you need to be careful about putting everything back together.

In some ways, security in Oracle is straightforward. Security and user identification are an integral component of the DBMS. By default, users have minimal access to any data in the database. Consequently, the major security efforts consist of identifying the access that people need and then enabling it with an SQL command. Of course, the security team will want to monitor system and database activity for potential breaches. Database triggers can be used to provide additional security controls by logging changes to sensitive tables.

Case: All Powder Board and Ski Shop

Ultimately, the owners of All Powder want to assign individual user permissions. Although the shop trusts its employees, it often hires students to work as clerks, and the owners would like to limit what the clerks can do with the application. The issue is only partly a matter of trust. It is also useful to protect the database

so clerks and other users cannot start changing form layouts or accidentally delete items.

The managers are also somewhat concerned about performance, particularly at the checkout machines. Sometimes the checkout lines get hectic, and the application has to be fast. Some of the issues can be handled by installing more computers, that way the salesperson can enter the basic customer data immediately, and the checkout clerk simply selects the customer and enters the product numbers. Of course, more computers mean that the company will need a network, and it means that more people will be simultaneously accessing the data, so the risk of collisions and locks increases.

Lab Exercise

All Powder Board and Ski Shop

DBMS developers learned early that indexes can significantly improve the performance of a relational DBMS. Primary key columns are almost always indexed because they often represent single-item lookups. Without an index, the computer has to search each row sequentially to find a match. Oracle automatically builds indexes on primary keys. However, you also need to think about building indexes on foreign keys to provide performance gains for joining tables. Most of the exercises in this chapter require that you have DBA permissions. Occasionally, you might need system DBA privileges. Usually, only the SYS user is given system DBA privileges—and you will only be able to use that account if you are running the database. It is a little tricky to log into SQL Plus as a system DBA. Usually you log in as yourself first, and then issue the connect command:

```
CONNECT sys/password@server as sysdba
```

Be sure to use the correct password and server name.



Activity: Monitor the Application Performance

Most Oracle DBAs use SQL to monitor and manage the database. However, this approach requires experience because it usually means you have to memorize several system views as well as create custom views to keep track of the database. Oracle has developed several versions of an Enterprise Manager tool to help less experienced DBAs perform common tasks. The 11g version runs as a separate service and can be accessed as a Web site: <http://server:1158/em/console>. You need DBA privileges to run the system.

Although you can run most of Oracle's administration tools through SQL, many of the more useful tools have been consolidated in the graphical enterprise manager. The Home page of the Enterprise Manager provides an overview of the database activity. It also contains tabs to three primary areas: Performance, Availability, and Server. Begin by selecting the Performance tab. Figure 10.3 shows the basic performance activity charts. These charts are relatively boring because they show minimal activity on a development machine. In a production setting, where

Action

- Log into the enterprise manager and click the Performance tab.
- Select the Advisor Central link under the Related Links.
- Select the Top Activity link under the list of monitoring links.
- Open a PL SQL session and issue several queries while you monitor the database performance.
- Run some reports or get several people to alter data at the same time.



Figure 10.3

hundreds or thousands of people are using the DBMS, you will see considerably more active charts. When you see peaks in usage, you can click the links to obtain more details information about which users, SQL statements, or applications are creating the loads.

The Automatic Database Diagnostic Monitor (ADDM) is a powerful management advisory tool. The system runs in the background and observes several performance aspects. To view the ADDM results, click the Advisor Central item under the Related Links section of the Performance page. The ADDM link is the first item in the Advisors list. The DBMS has to run for a while—preferably a few days—before the system gets enough data to make good recommendations. Figure 10.4 shows some of the performance data from a session—with minimal activity. With higher usage rates, the system can provide better recommendations, such as hardware recommendations (I/O subsystem) that suggest moving the data to a RAID drive. In terms of SQL recommendations, the ADDM advisor just makes overall recommendations to help you pinpoint areas that can be improved. To improve SQL queries, you need to look at each one.

You can use similar charts in the Performance monitor to locate other potential problems. If you identify a problem (say CPU usage has jumped), you can drill down or open related charts to identify the source of the problem.



Activity: Analyze Query Performance

SQL statements can be difficult for humans to improve. Developers and users look at the query to make sure it correctly answers the business question. However, queries often can be written in different ways. Long queries involving mul-

The screenshot shows the Oracle Enterprise Manager interface displaying performance metrics for a database instance. The metrics are organized into two tables.

	Per Second	Per Transaction	Per Exec	Per Call
DB Time(s):	0.1	0.2	0.01	0.02
DB CPU(s):	0.0	0.0	0.00	0.00
Redo size:	8,665.2	21,414.5		
Logical reads:	101.3	250.4		
Block changes:	41.8	103.3		
Physical reads:	0.4	1.0		
Physical writes:	2.5	6.2		
User calls:	4.8	12.0		
Parses:	5.7	14.1		
Hard parses:	0.2	0.5		
W/A MB processed:	309,455.0	764,765.6		
Logons:	0.1	0.1		
Executes:	15.7	38.7		
Rollbacks:	0.1	0.2		
Transactions:	0.4			

Instance Efficiency Percentages (Target 100%)			
Buffer Nowait %:	100.00	Redo NoWait %:	100.00
Buffer Hit %:	99.60	In-memory Sort %:	100.00
Library Hit %:	98.03	Soft Parse %:	96.53
Execute to Parse %:	63.62	Latch Hit %:	100.00

Figure 10.4

multiple tables and subqueries are even more complex. Throw in the fact that few people understand exactly how the Oracle query optimizer works, and you start to see that improving the efficiency of a query is difficult. The SQL Tuning Advisor was designed to find the queries that need the most work, and help you improve the query with minimal effort on your part.

Oracle uses two major concepts to improve query performance: statistics and indexes. Internal statistics help the query optimizer find the smallest set of rows, which reduces search time. But, generating statistics takes time and requires additional storage, so you again need to evaluate the tradeoffs.

To drive the optimization process, you must tell Oracle to analyze the database and collect statistics about each table. You can use the older Analyze Table command, but Oracle now recommends using a special procedure to analyze the entire database with one command. Figure 10.5 shows the three main commands in the DBMS_STATS package that gather statistics. Generally, you want to use the first version because it applies to the entire database. However, the first command requires SYS DBA permissions and if your database has many schemas

Figure 10.5

```
Exec DBMS_STATS.Gather_Database_Stats
Exec DBMS_STATS.Gather_Schema_Stats('powder')
Exec DBMS_STATS.Gather_Table_Stats('powder', 'Customer')
```

You might have to run the catproc.sql script first.

Action

- Find a SELECT query in the Top SQL list—preferably one from Chapter 5.
- Select the query to see the SQL statement and Oracle's plan.
- Run the SQL Tuning Wizard.
- Click the button to implement any suggestions.
- Run the Gather_Statistics command.

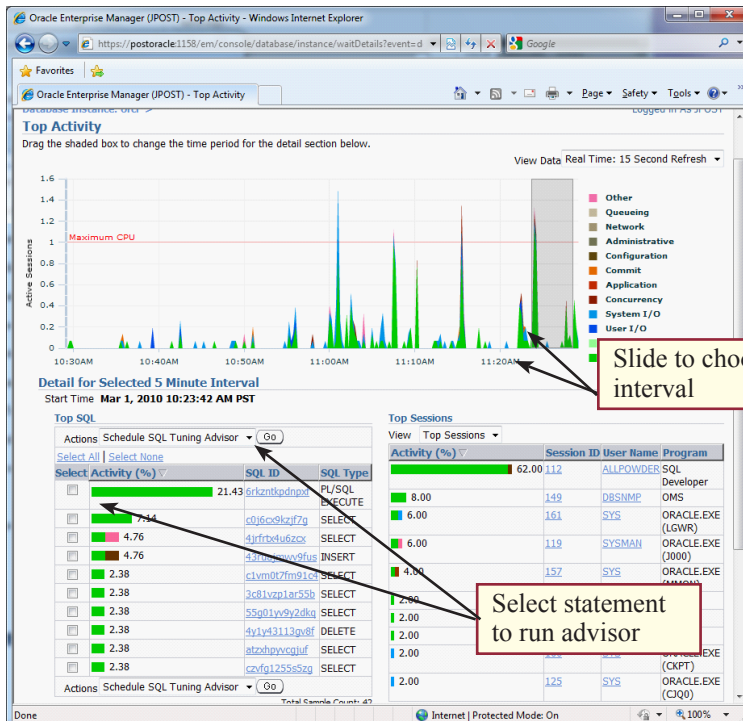
with relatively static data, you might want to update one schema at a time. Note that you might have to run the special `catproc.sql` script that installs the `DBMS_STATS` package. This script takes a while to run but only has to be done once. The `DBMS_STATS` commands should be run by a user with `DBA` privileges. Other procedures within the `DBMS_STATS` package will return the statistics to special tables that you can analyze to understand the structure of the database and manually tune the system. Hold off on running the data gathering commands for a couple of minutes.

Before starting the SQL Tuning Advisor, you might want to start SQL Developer and run some queries. You might try running some of the more complex queries from Chapter 5. To run the SQL Tuning Advisor, start the Enterprise Manager screen, select the Performance tab, and scroll down to the bottom to find the section of Related Links. Click the Top Activity link to see charts of the main activities. From the Top SQL list, select the top entry and click the Go button to analyze that query.

Figure 10.6 shows how you can choose a time interval to see the statements executed over that period. You can see the actual SQL statement by clicking on the SQL ID. You can also select several of the statements and ask the SQL Tuning Advisor to examine them and make recommendations. In this example, you would find that almost all of the statements are background housekeeping processes, many of them are run simply to provide the data for the Enterprise Manager display. However, if you run a relatively complex query, you should be able to find it in the list. From a DBA perspective, this list makes it easy to see which queries are taking the most resources and therefore need the most attention.

Figure 10.7 shows one of the queries from Chapter 4. Run the query in SQL Developer. Check the performance monitor to see if the query shows up in the

Figure 10.6



top SQL list. If not, use the search page to find it (search for %Customer.CustomerID%). The initial screen shows the basic statistics for the query. Select the Plan tab to see how Oracle intends to retrieve data to answer the query. Steps with high cost numbers take longer to execute. In this example, the ORDER BY statement is going to take more resources because it involves a few thousand rows of data. In a real-world situation, you should examine the use of the query to ensure that it truly does need to be sorted, and make sure that people really need to see all of the thousands of rows. Adding a WHERE clause back to this statement to limit the number of rows would improve the efficiency of the query. You can run the SQL Tuning Advisor on the query but it will probably not return any useful advice because the query is relatively simple. For complex nested queries, the Tuning Advisor can provide useful hints on how to improve the query performance.

Figure 10.8 shows a common set of recommendations. The SQL Advisor notices that the query optimizer does not have statistical data on the composition of tables. If it knows more about the data in each column, it can fine tune the query process. For example, if it knew that only a few skis are black, the optimizer would know to apply that condition first when a user specifies the color in a WHERE clause, because this approach would significantly reduce the number of rows that need to be searched for other conditions. If you trust the Tuning Advisor's recommendations, you can click the Implement button. You can then click the Show SQL button to see exactly how to implement the suggestions. You can also run the commands immediately. You can certainly trust the advisor in this example. However, if you look at the SQL, you will see that it creates three separate SQL statements—one to analyze each table. Since all of the tables in the schema need to be analyzed, you might as well issue the command to gather statistics for the entire schema, instead of doing it one table at a time:

```
Exec DBMS_STATS.Gather_Schema_Stats('AllPowder)
```

If you have time, you can rerun the query to see if the statistics make a difference. In most cases, the difference is great enough that the query will probably

Figure 10.7

The screenshot shows the Oracle Enterprise Manager (OEM) interface for SQL Tuning Advisor. The main window displays the SQL statement:

```
SELECT Customer.CustomerID, LastName, FirstName, Sum(QuantitySold*SalePrice) As SalesValue
FROM Customer INNER JOIN SALE ON Customer.CustomerID=Sale.CustomerID INNER JOIN SaleItem ON
Sale.SaleID=SaleItem.SaleID...
```

Below the SQL, the 'Plan' tab is selected, showing a detailed execution plan table with columns for Operation, Object, Order Rows, Bytes Cost, CPU Time, and Projection. The plan includes operations like SORT STATEMENT, HASH GROUP BY, HASH JOIN, and TABLE ACCESS FULL for tables CUSTOMER and SALEITEM.

Operation	Object	Order Rows	Bytes Cost	CPU Time (%)	Query Block Name/Object	Predicate	Projection
SELECT STATEMENT		8		36 100			
SORT ORDER BY		7	4,182,204,199K	36 8 0:0:1	SEL9E43CB6E		(#keys=1) SUM("SALEITEM"."QUAN...
HASH GROUP BY		6	4,182,204,199K	36 8 0:0:1			"CUSTOMER"."CUSTOMERID"/"NUMBER...
HASH JOIN		5	4,182,204,199K	34 3 0:0:1		"SALEITEM"."SALEID"="SALE"."SA...	(#keys=1) "CUSTOMER"."CUSTOMER...
HASH JOIN		3	1,400 53,32K	27 4 0:0:1		"SALE"."CUSTOMERID"="CUSTOMER..."	(#keys=1) "CUSTOMER"."CUSTOMER...
TABLE ACCESS FULL	SALE	1	1,400 10,938K	9 0 0:0:1	SEL9E43CB6E / SALE@SEL\$1		"SALE"."SALEID"/"NUMBER,22]","S...
TABLE ACCESS FULL	CUSTOMER	2	2,005 60,698K	17 0 0:0:1	SEL9E43CB6E / CUSTOMER@SEL\$1		"CUSTOMER"."CUSTOMERID"/"NUMBER...
TABLE ACCESS FULL	SALEITEM	4	4,182 44,924K	7 0 0:0:1	SEL9E43CB6E / SALEITEM@SEL\$2		"SALEITEM"."SALEID"/"NUMBER,22]..."

Oracle Enterprise Manager 10g
Database Control

Database: PostDB > Advisor Central > SQL Tuning Results: SQL_TUNING_1119292531067 >
Recommendations for SQL ID:279g31g8stbmn

Logged in As JPOST

Recommendations for SQL ID:279g31g8stbmn

Only one recommendation should be implemented.

SQL Text
SELECT Customer.CustomerID, LastName, FirstName, Sum(QuantitySold*SalePrice) As SalesValue FROM Customer INNER JOIN Sale ON Customer.CustomerID=Sale.CustomerID INNER JOIN SaleItem ON Sale.SaleID=Sale...

Select Recommendation

Select	Type	Findings	Recommendations	Rationale	New Benefit Explain (%)
<input checked="" type="radio"/>	Statistics	Table "POWDER"."SALEITEM" and its indices were not analyzed.	Consider collecting optimizer statistics for this table and its indices.	The optimizer requires up-to-date statistics for the table and its indices in order to select a good execution plan.	
<input type="radio"/>	Statistics	Table "POWDER"."SALE" and its indices were not analyzed.	Consider collecting optimizer statistics for this table and its indices.	The optimizer requires up-to-date statistics for the table and its indices in order to select a good execution plan.	
<input type="radio"/>	Statistics	Table "POWDER"."CUSTOMER" and its indices were not analyzed.	Consider collecting optimizer statistics for this table and its indices.	The optimizer requires up-to-date statistics for the table and its indices in order to select a good execution plan.	

Copyright © 1996, 2004, Oracle. All rights reserved.
About Oracle Enterprise Manager 10g Database Control

Database | Help | Logout

Main recommendation

Figure 10.8

not make the Top SQL list, and you will have trouble finding the details. If you include multiple SELECT statements, or a larger portion of database activity, the query analyzer can make more sophisticated recommendations.



Activity: Evaluate Storage Options and Indexes

Oracle has several other methods to improve query performance—related to how the data is stored. The two main tools are indexes and materialized views. An index can exponentially reduce the number of lookups in a search. On the other hand, indexes have to be updated whenever data is changed, deleted, or added. A materialized view is a snapshot of the data that is periodically updated. Remember that joins can be expensive in terms of resources. Indexes help by providing faster search capabilities in a sorted list. For instance, to find a name in a phone book, you would not want to start at the beginning and read every name. Instead, you know the list is sorted, so you can start in the middle, and move forward or backward in large jumps to find a specific name. A sorted index works the same way, by exponentially decreasing the time required to find an entry—as few as 20 lookups for a binary search on one million entries.

A materialized view goes even further. It creates a temporary table structure that holds non-normalized data. It builds all of the joins across the tables and moves everything into the new view. The DBMS no longer has to perform the joins when retrieving data. But, queries will not have completely up-to-date data since the materialized view is only recomputed at certain intervals. Many decision support systems can live with

Action

- Run several queries from Chapters 4 and 5 to establish base data.
- Run the SQL Access Advisor.
- Pick the option for Both Indexes and Materialized Views.
- View the recommendations and examine the proposed SQL.

this slightly out-of-date data. When a manager runs a query asking for total sales, it rarely matters if the query missed a couple of sales that took place in the last five minutes. But, as DBA, you still need to make the final decisions—both in terms of indexes and materialized views.

Placing too many indexes on a table can result in even worse performance. Your job is to find the balance with enough indexes to improve performance for key tasks, but not so many that other portions become too slow. This balance is unique to each application and can be difficult to find. Ultimately, you will have to fine-tune the application over time. A few simple rules help you begin: (1) All primary keys should be indexed, (2) Join columns should be indexed—particularly in large tables, (3) Heavily searched or sorted columns should be indexed, and (4) Transaction tables that are constantly changed (such as SaleItem) should have few indexes.

An important step for the query processor is to select the best approach for joining tables and restricting rows to retrieve data as quickly as possible. Part of this decision depends on the availability of indexes. Ultimately, the best performance depends on the amount and distribution of the data in the table. For these reasons, Oracle has implemented a cost-based optimizer that examines some statistics about the data to determine how to execute a query. This same optimizer can be used to help tune a query and identify which columns should be indexed.

You can use the SQL Access Advisor to obtain recommendations on both indexes and materialized views. The most reliable way to run the advisor is to first execute several queries that use JOIN statements on your database. You should consider running several of the queries from Chapters 4 and 5 to provide a set of data for the advisor. (The advisor does contain an option to create a hypothetical workspace where you can pick all tables in a schema, and maybe it will work someday.) After you have run a few queries, go to the Advisor Central page and

Figure 10.9



select the SQL Access Advisor link. For the most part, you can use the default choices. As shown in Figure 10.9, you usually select the option to evaluate Both Indexes and Materialized Views on the second step. On the third step (not shown), make sure you change the Window to Standard and verify Start Immediately is selected.

The advisor will schedule the job and you can click the Refresh button on the task page to see when it finishes. Click the option to View Recommendations, and click the Show SQL link to see the detailed proposal.

Figure 10.10 shows a portion of the recommendations. Your results will vary. With small workloads, the Advisor might not find any useful recommendations. The tool is designed to analyze heavy workloads and complex queries. Remember that you already gathered statistics for the underlying tables in the prior exercise. Also, Oracle automatically builds indexes for primary key columns. Consequently, the main recommendation is to create a materialized view between the Sale and Customer tables and then gather statistics for it. Oracle even makes it easy to run the SQL statement to accept the recommendations. However, you should think about the implications of the materialized view for a few minutes. In a retail store, the Sale table is a transactions table where rows are constantly inserted into the table. You might consider using the materialized view if you limit the times it is updated to once or twice a day so that it does not bog down the system during the major selling times. You should also make sure users understand that the data from the queries is likely to be dated. Also, make sure that your application forms use the raw underlying tables instead of the materialized view.

If you have problems running the Access Advisor, you can obtain the same results by running a SQL procedure directly. You need the Advisor role to run the scripts. You can find the scripts in the DBMS_ADVISOR package. The Oracle

Figure 10.10

```
CREATE MATERIALIZED VIEW "JPOST"."MV$$_00160000"
REFRESH FAST WITH ROWID
ENABLE QUERY REWRITE
AS SELECT POWDER.SALE.ROWID C1, POWDER.CUSTOMER.ROWID C2,
"POWDER"."CUSTOMER"."ADDRESS"
M1, "POWDER"."CUSTOMER"."CITY" M2, "POWDER"."CUSTOMER"."CUSTOMERID" M3,
"POWDER"."CUSTOMER"."DATEOFBIRTH" M4, "POWDER"."CUSTOMER"."EMAIL" M5,
"POWDER"."CUSTOMER"."FIRSTNAME" M6, "POWDER"."CUSTOMER"."GENDER" M7,
"POWDER"."CUSTOMER"."LASTNAME"
M8, "POWDER"."CUSTOMER"."PHONE" M9, "POWDER"."CUSTOMER"."STATE" M10,
"POWDER"."CUSTOMER"."ZIP"
M11, "POWDER"."SALE"."CUSTOMERID" M12, "POWDER"."SALE"."EMPLOYEEID" M13,
"POWDER"."SALE"."PAYMENTMETHOD" M14, "POWDER"."SALE"."SALEDATE" M15,
"POWDER"."SALE"."SALEID"
M16, "POWDER"."SALE"."SALESTAX" M17, "POWDER"."SALE"."SHIPADDRESS" M18,
"POWDER"."SALE"."SHIPCITY" M19, "POWDER"."SALE"."SHIPSTATE" M20,
"POWDER"."SALE"."SHIPZIP"
M21 FROM POWDER.SALE, POWDER.CUSTOMER WHERE
POWDER.CUSTOMER.CUSTOMERID = POWDER.SALE.CUSTOMERID;

begin
dbms_stats.gather_table_stats("JPOST","MV$$_00160000",NULL,
dbms_stats.auto_sample_size);
end;
```

Primary materialized view
joining Sale and Customer

Gather statistics for the new view

documentation explains the process and provides sample scripts. The basic steps are:

```
DBMS_ADVISOR.CREATE_TASK(...);
DBMS_ADVISOR.CREATE_SQLWKLD(...);
DBMS_ADVISOR.ADD_SQLWKLD_REF(...);
Sql_text := 'SELECT ...';
DBMS_ADVISOR.ADD_SQLWKLD_STATEMENT(...);
DBMS_ADVISOR.EXECUTE_TASK(task_name);
```

A copy of the script is included with the files for this chapter (SetupAccessAdvisor.sql). You can define multiple SQL statements and add them to the work load. Including more queries gives the advisor more information to work with and it can design more sophisticated recommendations. After you have edited the script, run it (Exec SetupAccessAdvisor). You have to retrieve the results into a variable to see them:

```
VARIABLE buf CLOB;
Set long 50000;
Execute :buf := DBMS_ADVISOR.GET_TASK_SCRIPT('First_
Task');
Print :buf;
```

Oracle has additional options for controlling how data is physically stored. These options are usually established when a table is created, so you can read the Oracle documentation on CREATE TABLE to see the list of options. Common examples include partitions, clusters, and hashed tables. Partitions are used to split tables so portions are stored on different physical devices. Clusters are used to group items together. For example, storing the sales data along with each customer so all of the data is retrieved in one pass. Hashed tables are similar to specialized indexes, where a single row can be physically retrieved based solely on the key value. Although these tools have been useful in the past, hardware solutions are generally preferable today. In particular, you can get substantially greater performance gains by storing data on RAID controllers.

The expert recommendations are an easy way to quickly identify good index candidates. However, you must still be cautious. Remember that adding indexes to a table speeds up queries but slows down deletes, inserts, and updates. Every time a row of data is changed, all of the indexes on the table have to be rebuilt. The expert analyzer tries to evaluate how the tables are used—based on recent queries, particularly the slow-running queries. However, you should still carefully evaluate its recommendations. Once the database is in production mode, you should keep good records of any changes you make and then observe the processing times carefully for a couple of weeks to see how your changes affected the overall database performance.

Security and Privacy



Activity: Backup and Recovery

Backup and recovery of an Oracle database can be straightforward, or it can be complex. If you are able to shut down the entire database, you could simply use the operating system utilities to copy the underlying data files and the system control file. More realistically, the business will want to run the database without interruption. Oracle uses its redo logs to handle this situation. The underlying data

is backed up at a certain point in time. Data that is being changed as the backup is taking place is written to the redo logs. These logs are also backed up so the database can recover everything up to the point of the backup, and then roll forward the additional changes from the redo logs.

In the Oracle Management Server, Oracle provides the RMAN (recovery manager) tool to automatically back up a database, as well as initiate the recovery steps if something happens to the database.

You run this tool from the command line on the server (`rman target sys/password@database`). Use the `SHOW ALL` command to get a status listing of the current configuration. You will have to read the Oracle documentation for a complete explanation of the commands. Once you have identified and configured the backup tape or disk drives, you can issue the backup command to make a copy of the database.

In order to back up the database while it is running, Oracle requires that the database be running in Archive Log Mode. By default, the initial database is usually running in No Archive Log mode. The easiest way to change this setting is to use the enterprise manager, select the database, the instance, and the configuration options in the tree view. Under the Recovery property tab, you can check the box to select Archive Log Mode. This action does mean that your database will continually use more disk space. As changes are made to the database, they will be permanently saved to a set of archive files. You will generally have to stop and restart the database for this change to take effect.

You can use the Maintenance tab in the Enterprise Manager to configure most backup and recovery actions. Figure 10.11 shows the main options under the Availability tab. You should first check the Recovery Settings link. Most likely the `NoArchiveLog` option is set, so you will not be able to perform a hot backup of the database. By default, the only way to backup the database is to shut it down and make copies using the operating system utilities. This option is set when you installed Oracle and chose the default database. For now, you do not need to change the settings because you might have to rebuild your database. In a production environment, you would alter the setting when you build a customized initial database.

You can also check the Backup Settings link to see how to specify the backup device. If you have a tape system attached to the server, you can tell Oracle to dump the entire database to tape. The backup process will automatically write everything to multiple tapes if necessary. The other option is to write the files to removable hard drives. Drives are considerably faster than tapes for both writing and retrieval.

Ultimately, the actual backup options within Oracle are becoming less useful as the underlying hardware changes. In most cases, you will want to run a RAID system of drives and configure the system so that it automatically writes a backup copy of every item stored to a drive. If one drive fails, you can simply replace the drive and they system will automatically recover—in many cases while leaving the entire database online. RAID systems generally require their own backup systems, so you will use the operating system tools to backup the drives. In a situ-

Action

Start the Enterprise Manager as SYSDBA and switch to the Availability tab.

Click the Recovery Settings option to check the `NoArchiveLog` status.

Return to Availability and check the Backup Settings link.

If you have permission, and if you have drive or tape facilities you can try to create a backup.

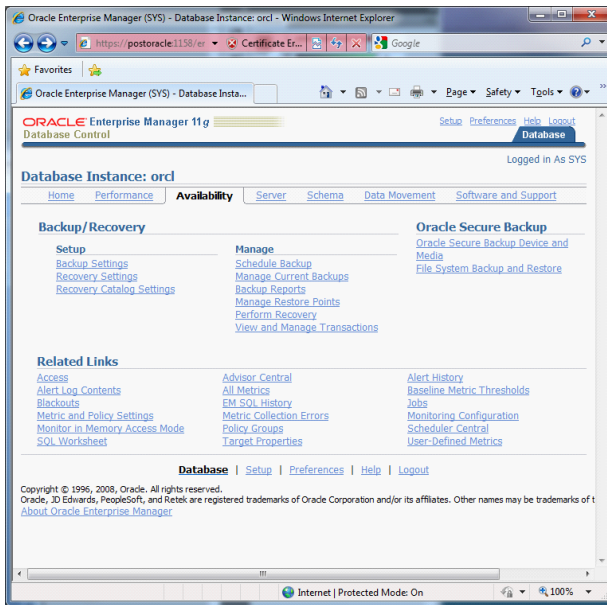


Figure 10.11

ation where you absolutely need to guarantee recoverability of the data, you could run both the operating system and Oracle backup tools; essentially making two copies of the data. But, you need enough time and storage space available to copy that much data. In any case, it is always easiest to recover a cold database copy than a hot copy. If at all possible, you should try to schedule downtime so that you can make complete copies of the database at least once in a while.



Activity: Setting User-Level Security Controls

The Oracle database system is built and distributed with a complete security system. Users must log in to the system to see any of the data. The DBA can create new users and assign rights to the users or to groups of users. Initially, users have no permissions. Users and security rights can be created through SQL commands or by using the enterprise manager. The enterprise manager provides a relatively easy-to-use graphical interface, and is useful when you need to make simple changes or check on a particular item. However, if you need to set several security permissions at one time, it is often easier to write the SQL commands into a text file and execute the file in SQL Plus. If you do not happen to remember the exact SQL syntax, it is sometimes helpful to set up a test example using the enterprise manager interface, and copy the SQL command that it writes.

The first issue to face is that Oracle needs to be able to identify the individual users. Figure 10.12 outlines the basic process. The main database application contains forms, reports, and tables. As the DBA, you want to assign individual permissions to separate users for each object. For instance, sales clerks would be able to read some supplier data, but not change it, and probably would not need access to the main supplier form.

Action

Identify the SalesClerk and SalesManagers roles and determine what permissions are needed on the basic Sale, SaleItem, Customer, and Inventory tables.

Create three new users and assign them simple passwords.

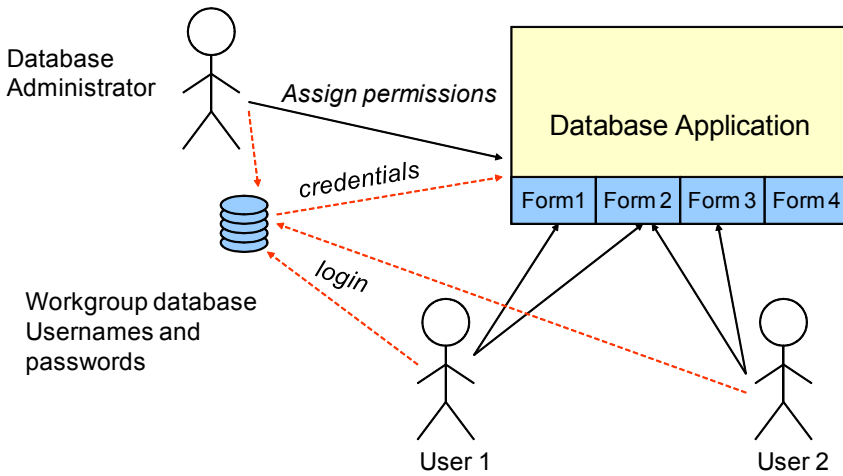


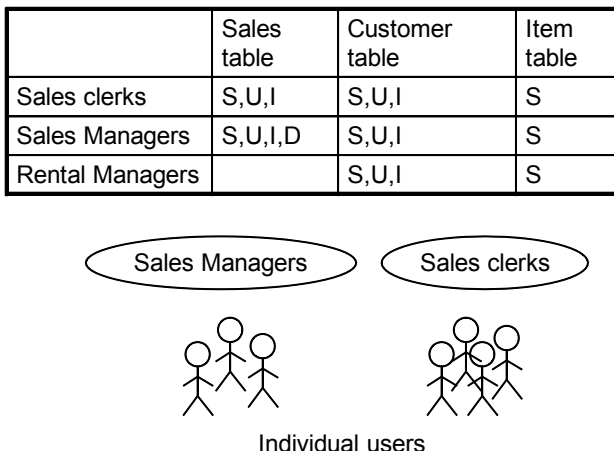
Figure 10.12

But, before you can assign any permissions, the database application needs to be able to identify the user.

Identifying a user is an important step in securing a database or a computer system. Oracle has two primary means of identifying users: (1) Individual accounts can be created within the database, where users are assigned a unique username and a password, or (2) User accounts can be created on the computer that is responsible for handling the login and passing the username to Oracle. Each organization must balance the costs and benefits of the two methods. It is relatively easy to set up a new user account within Oracle. The main drawback to this approach is that users need to remember yet another username and password. Firms are increasingly looking for single sign-on systems where users log into a central directory and all computers and applications pull the user identity from this central server.

Before attempting to create users and assign security, you should write down a list of usernames and initial passwords that will be asked to enter into the workgroup database. While you are identifying users, you should also classify them in terms of tasks or groups. You almost never want to assign permissions to in-

Figure 10.13



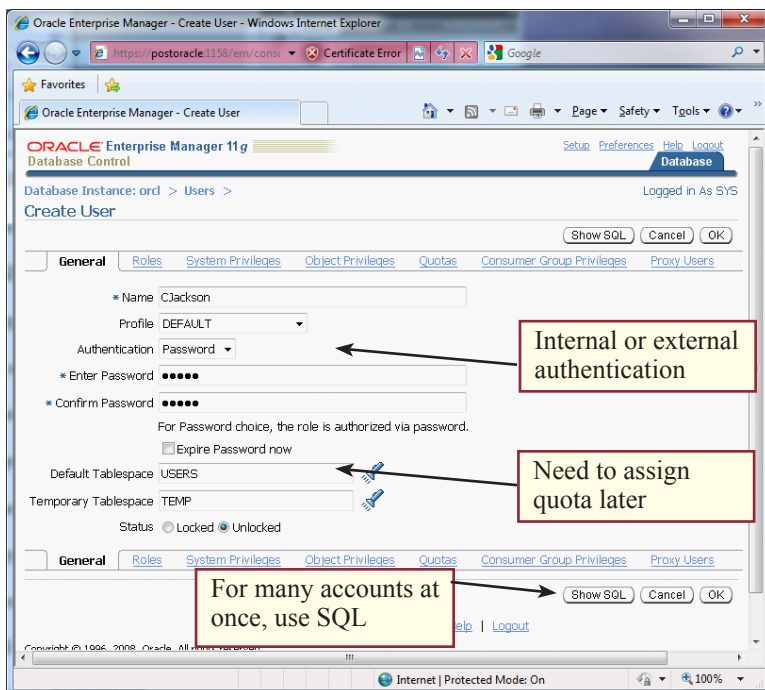
dividual users. Instead, you place users into groups and assign database permissions to the roles of these groups. Figure 10.13 illustrates the main concept. By assigning permissions to the role, you should only have to set permissions once. As individual roles are added to or removed from users, their permissions automatically change.

Creating a new user account is relatively easy with the enterprise manager tool. With DBA permissions, click the Server tab and then the Users link under the Security listing. Click the Create button to open a form for new users. As shown in Figure 10.14, you really only need to enter the name of the new user account and the password. If necessary, you can specify that the user is to be authenticated by using the operating system account (external) or an Oracle global directory (global). You can also specify the main tablespaces for this user, but it is easier to change those later. Remember that all tasks in Oracle are really handled by SQL. Click the Show SQL button to see the commands used to create a new user. The command is usually straightforward. In fact, most DBAs keep a simple script handy to generate user accounts since it is faster than opening the Enterprise Manager. You can also write a PL/SQL program that would read the list of names and passwords from a file or table and execute the statement to create each account automatically.

The next step is to create the roles of SalesClerk and SalesManager. Return to the main Administration page and click the Roles link under the Security listing.

Action
Create the SalesClerk and SalesManager roles.
Assign appropriate table permissions to the new roles.
Assign one of the roles to each of the new users.
Use the Sales form to test the accounts and roles.
Test the roles by using SQL statements.

Figure 10.14



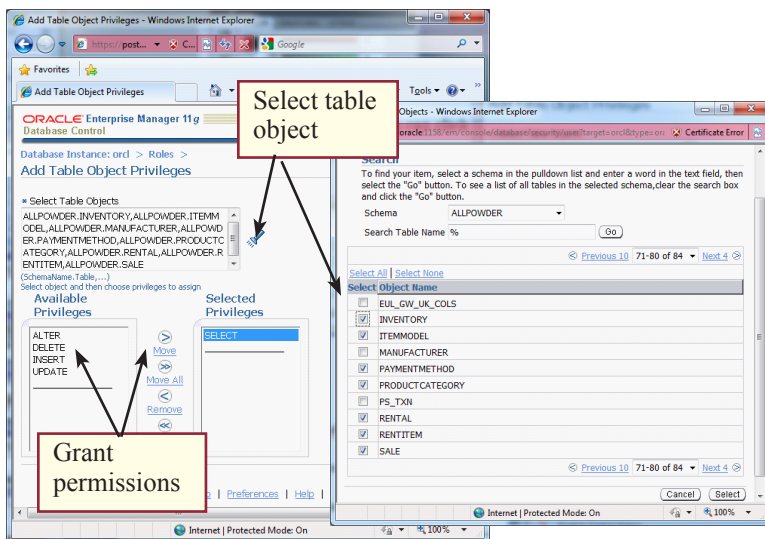


Figure 10.15

The list of system roles already created is extensive, and someday you should explore those choices to see what actions are supported. For now, you simply click the Create button to build a new role. Enter SalesClerk as the name of the new role. Double-check Figure 10.13 to see what permissions are needed by users in this group. You will also need to assign a quota on the main tablespace (Users).

Note that all of the permissions are object privileges, so click the Object Privileges link in the Role screen. Select the Table option and click the Add button. The form is configured to assign permissions to a group of tables. Click the flashlight icon to open the table browser. Pick the tables where the sales clerks will need the SELECT permission—which is most of the tables. When you click the Select button on the table form, the names of the selected items will be entered in to the box on the first form. Choose the SELECT option and click the arrow button to move it to the right-side box. When you click the OK button, you will be returned to the main assignment screen. Repeat the process for tables that the sales clerks can update, insert, and delete. This list is shorter, and probably includes Sale, SaleItem, Rental, RentalItem, and possibly Customer. More importantly, it does not include the Inventory or Model tables because clerks should not have the ability to change prices or delete items. When all of the permissions are correct, you can click the Show SQL button to see the assignment command. As a DBA, you generally want to copy this list and store it in a text file to make it easier to modify later and to provide an easy record of the permissions granted to this role. You can click the OK button to run the script and assign the permissions to the SalesClerk role.

Once the users and the roles have been created, it is relatively easy to assign the roles to each user. Figure 10.16 shows the process using the enterprise manager. Return to the Users administration page and select the CJackson user you created earlier. Click the Roles link at the top of the page. You will notice that he has been granted the CONNECT role by default. You really need to look at the permissions granted to the CONNECT role. Most DBAs are reluctant to assign this role to general business users—because it gives them permission to create tables, views, and sequences. Most DBAs would only grant users the CREATE SESSION permission—which is required to enable the user to log in. So, you really should re-

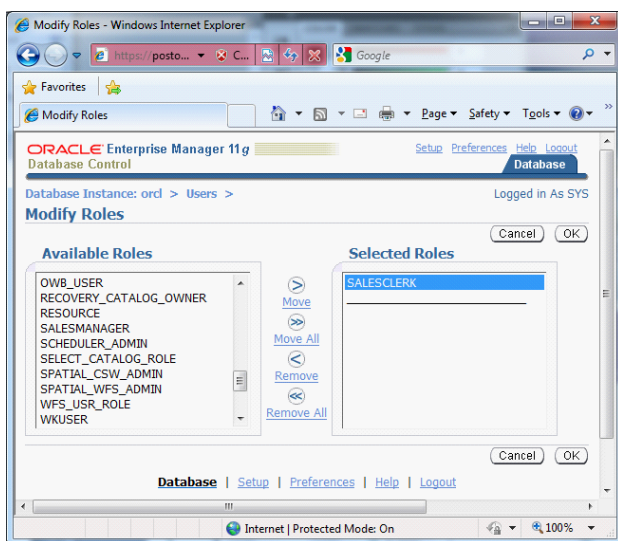


Figure 10.16

move this role, then return to your SalesClerk role and add CREATE SESSION as a System Privilege. As shown in Figure 10.16, it is straightforward to add roles or remove them from a user account. Again, once you click the OK button, you can check the SQL statement to see how the simple commands. Although the Enterprise Manager seems easy because of its ability to pick items from lists, it is much simpler to write SQL scripts when you have to perform a process several times. How often are you going to have to create users and assign roles? If you use a script, you can test it and verify that the permissions are correct. In the future, when you create a new user, you simply run the script and you know that it will be accurate. If you have to configure each new account person individually with the Enterprise Manager, you are likely to make mistakes.

When defining roles and assigning them to users, it is important to remember that users are often assigned multiple roles. Security is more effective when the roles are assigned with relatively small granularity. That is, instead of creating two or three all-encompassing roles and assigning one to a person, it is better to break roles into smaller pieces and assign multiple roles to each person. In the All Powder case, you should consider separate roles for Sales, Rentals, Receiving, Adding Customers, and so on. Then sales clerks would be granted the roles for sales, adding customers, and perhaps one or two other tasks. If a person is promoted or moved to a different position, you simply have to change the role assignment to match the new job. It is important that the roles and their names closely match the business jobs.

Of course, you need to test the security assignments. Try the test first using the forms—which is how the sales clerks will generally use the application. Notice that the forms themselves are stored outside the database, so they are not directly subject to the security conditions. However, as soon as the form tries to retrieve data, the security conditions are imposed, so unauthorized users will not be able to see any data. In fact, the first time you try to run a form as a sales clerk, you will probably receive an error message. You often need to test and retest each form until you get the security permissions right.

It is also possible to add security roles and conditions to the JDeveloper forms themselves. The roles created within the database in this lab are the best place to start. You must always protect the data at its source, so that if someone bypasses the forms the database controls will still limit the access and changes. However, as briefly explored in Chapter 9, you can create separate logins for the forms. In fact, for the employees, JDeveloper has tools that enable you to use the logins within Oracle. That way, employees can be given separate Oracle logins that work for both the database and the applications. This approach is a bad idea for customers—because you do not want to create accounts within your database for each external customer. Instead, you add a username and password column to the Customer file and use those credentials to validate the user. Then the application can use the information to customize menus and block access to the forms.

Exercises



Many Charms

Samantha and Madison do not believe that security will be a critical issue at Many Charms. The database will run on one machine and rarely be used by anyone except the two of them. On the other hand, they do need a system on which it is easy to create backup copies. And, for some security, they are willing to use the single database password. On the other hand, they are concerned about performance. Although they do not expect too many orders arriving at one time, they do want to examine some lengthy reports to evaluate sales trends.

1. Run the performance analyzer to improve the performance of the database and identify indexes needed. Also check the performance for the report queries.
2. Create a backup option that makes it easy for the managers to create a backup copy. As much as possible, keep it down to one button. But provide some notices about moving the backup copy offsite in case of fire.
3. Add the security provisions needed by Samantha and Madison.



Standup Foods

Security is a serious concern for Laura. The database contains a large amount of data about employees—and celebrity preferences. Managerial employees will need access to the database to enter a considerable amount of information regarding other employees and the status of the event. Consequently, employee access has to be carefully thought out. Managers should have the ability to enter data on employees who report to them, but should not be able to even see most data on other employees. You will have to use queries to provide this level of security. Assigning access to the entire employee table would give managers too much permission. Instead, you will have to set up queries that retrieve the data for specific approved managers and then give the managers access to the data through that query.

1. Run the performance analyzer to improve the performance of the database and identify indexes needed. Also, check the performance for the report queries.
2. Create a backup option and a written set of procedures that Laura can follow to ensure the data is protected.

3. Create the security provisions needed by Laura. Concentrate on the permissions needed to handle evaluation of employees by a manager—without allowing the manager full access to data for all employees.



EnviroSpeed

The knowledge in the EnviroSpeed database is a major strategic asset to the company. This data represents experience gained over several years and enables the company to be considerably more productive and profitable than its competitors. Tyler and Brennan believe it is critical to protect this asset. On the other hand, it is also critical that employees and hired experts have immediate access to all of the knowledge during a disaster cleanup. Security controls need to be set carefully to protect the database from outside hackers. Fortunately, Brennan and Tyler can trust all of the employees and experts and do not believe it is necessary to track the exact usage by each person to prevent theft.

1. Run the performance analyzer to improve the performance of the database and identify indexes needed. Also, check the performance for the report queries.
2. Create a backup option and a written set of procedures to follow to protect the database.
3. Create the security provisions needed. Concentrate on protecting the data from external attacks.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you pick one or your instructor picks one, perform the following tasks.

1. Run the performance analyzer to improve the performance of the database and identify the indexes needed. Also check the performance for the report queries. Identify the main areas that will be stressed as loads increase.
2. Create a backup option and a written set of procedures to protect the database.
3. Identify the main risk factors and implement the security provisions needed to protect the data, but still ensure users have the access needed to perform their jobs efficiently.

Distributed Databases

Chapter Outline

Location, Location, Location, 271

Case: All Powder Board and Ski Shop, 272

Lab Exercise, 272

All Powder Board and Ski Shop, 272

The Internet, 276

Exercises, 279

Final Project, 280

Objectives

- Create database links to connect data from multiple sources.
- Replicate a database and synchronize the changes.
- Create Web pages to edit data over the Internet.
- Export and import data as XML files.

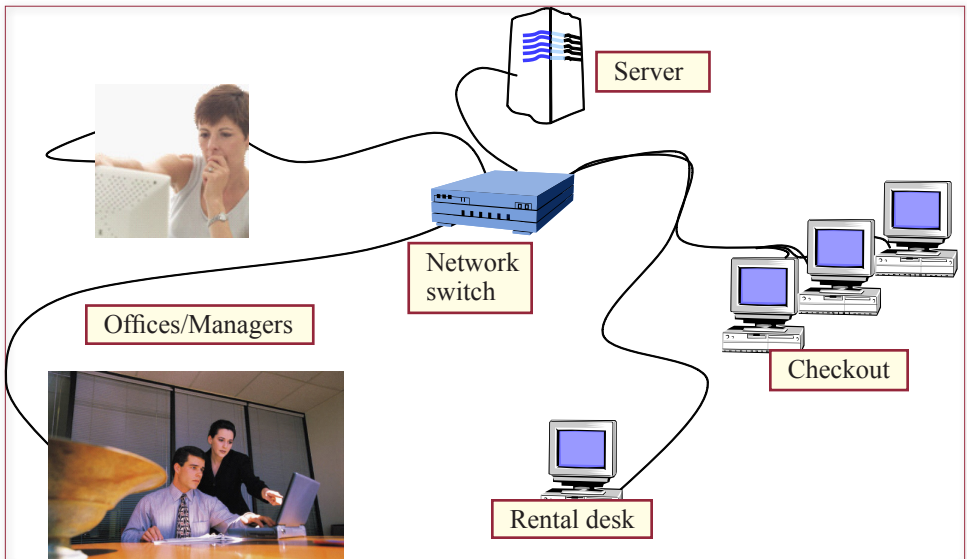
Location, Location, Location

Even small companies often need to access data in multiple locations. This distributed access generates several issues in database management. The most important question you will face is where to store the data. The answer depends on how the database is used, how fast the connections are, and whether everyone needs 24-hour access to immediately current data. The first step in designing a distributed system is to answer these questions and determine the most efficient method for handling data updates in the various locations. Note that efficiency also includes cost issues.

Oracle provides several tools to support distributed access to data. The three primary approaches are (1) Internet access, (2) linked databases, and (3) data replication. You could also make the argument that the cluster system is a distributed system on a local scale. The primary purpose of the cluster system is to improve performance and reliability. You can use storage area networks to separate the data files from the processors. The grid system enables multiple computers to work on the same data at the same time. At some level, Oracle treats all of this hardware as a single (really fast) system. On the physical side, you gain flexibility by being able to move, change, and add hardware without altering the database design.

In terms of distributed access, the use of Web-based forms and reports provides considerable flexibility in terms of client access. The database itself has become more centralized, which makes it easier to manage. Yet, managers can access the information from anyplace with an Internet connection. As wireless devices, including cell phones, gain more Internet features, managers will have almost continuous access to the data regardless of location. And this power comes almost automatically when you build Web-based forms and reports. Since the other labs cover these tools, this chapter will focus on database links and replication. Just remember that whenever you encounter the need for a distributed system, you should first ask whether the problem can be solved using the Internet.

Figure 11.1



Case: All Powder Board and Ski Shop

Initially, you might think that All Powder with only one store would not care much about distributed databases. Certainly, if the owners consider adding a second store, the issues become more complex. This situation will be examined in a second lab exercise. In the meantime, even with one store some basic distributed issues arise. The distributed aspect arises because there will be several locations within the store that need access to the database—the checkout stations, the rental desk, and a couple of offices. Figure 11.1 shows that each of these locations will have a computer that needs to run the forms and share the data.

Distributed questions within a single building are much easier to solve than those spreading across wide geographic areas. The reason is because of the speed of local area networks. Within the store, it is relatively easy to install a high-speed LAN that can transfer data as quickly as a typical computer can transfer data to an internal hard drive. Consequently, it is possible to store the database in one location and share it with all of the other computers—with no noticeable delays. You have already built the system so that all of the data files, forms, and reports run on the server. To provide access from multiple locations, all you need to do is ensure that each station has a machine with a Java-enabled Web browser and a network connection back to the server. You might even consider portable wireless devices for some of the employees so they can help customers throughout the store. The key point is that the database will run without any changes.

Lab Exercise

All Powder Board and Ski Shop

The existing single server with network access will work well as long as most of the operations occur in one location. How well would this system work if the company acquires an inventory warehouse or opens another store? The answer depends on how fast of a connection the company is willing to lease between the other locations and the database server. With a relatively high-speed connection, the Web-based approach will work fine. There might be slight delays if everyone opens major reports at exactly the same time, but most of the time, the connection is simply transferring small amounts of transaction data. With only a few users, even a fractional T1 line or frame relay might be sufficient to handle the typical loads. In a real-life situation, you could monitor the amount of traffic and network usage within the existing store to get a better idea of how much bandwidth would be needed to connect to a second store.

On the other hand, you could eventually reach a situation where you need faster response times at each location. In this case, you might split the database and run two or more servers. The servers would support local operations, but some reports would need to retrieve data from both databases. As long as you have a network connection, you can create a database link that enables forms, reports, and SQL to access data from any connected database. The process is relatively easy, but you will need to think about security issues.



Activity: Create Database Links

The first step is to find or create a second database. It is even better if you happen to have two machines running as Oracle servers. As an example, you can create multiple databases that run on the same server. The easiest way to create

a new database is to start the Database Configuration Assistant from the main Windows menu Oracle Home/Configuration and Migration Tools. Create a unique SID for the new database (such as Test). The tool creates a new administration link in the main menu that you can use to connect to the database. Use it to create a user account or use one of the system accounts. Start SQL Developer and add a new connection. The basic connection is the same as before: computer=localhost, port=1521, but use the new SID=test. Now you can log on and create a table. To illustrate a database link, you only need a small table with a couple of rows of data. Figure 11.2 shows a small Customer table with a couple of rows of sample data. This new database and table will be the target link.

Open a new Worksheet in your original AllPowder connection within SQL Developer. You must have DBA permissions in this account. In this main database, you want to create a database link to the target you just created. Figure 11.3 shows the SQL statement needed to create the database link. It needs to be issued only one time. Now you can access tables in the other database by adding the name of the link to any SQL commands as shown in the SELECT statement. You should create a descriptive name for the link so you remember which server you are using. In this example, the server would be for the New York store. In general, avoid being too specific about a location, because you might want to move the hardware later. The link name should reflect the business operation. The other tricky part of the link is that it specifies a remote username and password. If you do not specify the CONNECT TO clause, the system will attempt to connect with the current username/password from the local system. This approach would require that you establish identical user accounts on both machines. From a security perspective, it might be slightly safer to create identical accounts, but it takes time and effort to continually synchronize the accounts and passwords. The USING clause references the hostname of the computer that you entered when you created the network connection to the database. The link needs to be created only one time. From this point, you can reference the table much like any other table in your schema.

Action

- If necessary, create a second database, preferably on a different machine.
- Create a small Customer table and load it with four or five rows of data.
- Return to your main database and create a database link to the target.
- Run an SQL statement that retrieves data across the link.

Figure 11.2

```
CREATE TABLE Customer
( CustomerID INTEGER,
  LastName   VARCHAR2(15),
  FirstName  VARCHAR2(15),
  Constraint pk_Customer Primary Key (CustomerID)
);
INSERT INTO Customer (CustomerID, LastName, FirstName)
Values (1,'Smith', 'Adam');
INSERT INTO Customer (CustomerID, LastName, FirstName)
Values (2,'Keynes', 'John');
INSERT INTO Customer (CustomerID, LastName, FirstName)
Values (3,'Samuelson', 'Paul');
INSERT INTO Customer (CustomerID, LastName, FirstName)
Values (4,'Robinson', 'Joan');
Commit;
```

```
CREATE DATABASE LINK NewYork
CONNECT TO RemoteUser IDENTIFIED BY t1
USING 'dbhostname';

SELECT * FROM Customer@NewYork;
```

Figure 11.3

Of course, if the actual network connect is slow and you try to retrieve thousands of rows of data, you will have to wait quite a while. So, be careful and think about how much data your query might attempt to return over a database link.

If you create a couple of queries in SQL Developer using database links, you will quickly grow tired of having to specify the name of the link, the database schema, and the name of the table. Even if you do not use a distributed database, you can quickly grow tired of specifying the name of a schema to identify tables that are commonly used. Oracle provides a shortcut to make it easier to reference tables anywhere in the database. You simply create a public synonym. Figure 11.4 shows the basic syntax for creating a synonym that points to the standard emp (employee) table for the common Oracle scott database schema. Notice how the synonym simplifies the following SELECT statement, since the synonym emp replaces the full name of the table. In fact, synonyms and database links create location transparency. Users and applications use the synonym, and you can change the physical location of the data table simply by changing the value of the synonym. Users never need to know or care where the data table is located.

```
CREATE PUBLIC SYNONYM emp
FOR scott.emp@sales.us.mycompany.com;
SELECT LastName, FirstName from emp;
```

Figure 11.4



Activity: Replicate and Synchronize a Database

By themselves, database links do not provide replication of data. In fact, you generally separate the data and each database holds only the data needed in each location. Oracle's query processor automatically takes the distributed data into account and attempts to minimize the transfer of data across linked databases. Linked databases are often connected via slower network lines, so transferring large amounts of data is usually bad. Oracle uses its cost analysis system to find the best way to retrieve the data.

However, sometimes you need to replicate portions of the database, so that the same data exists at multiple sites. For example, you might want to keep a single product catalog that is shared by all stores. As you make changes to the catalog, you want the changes pushed to the replicas stored on the remote databases. With Oracle 11g, the most common (possibly the only) solution is to use materialized views. Recall that a materialized view is a snapshot of the data. It is often used to improve query performance by joining tables ahead of time. However, a material-

Action

- In the Enterprise Manager select the Schema tab.
- Click the Materialized Views link.
- Click the Create button.
- Create the EmployeeMV.

ized view can also be used on a single table or even a portion of a table (using a query with a WHERE clause and one table).

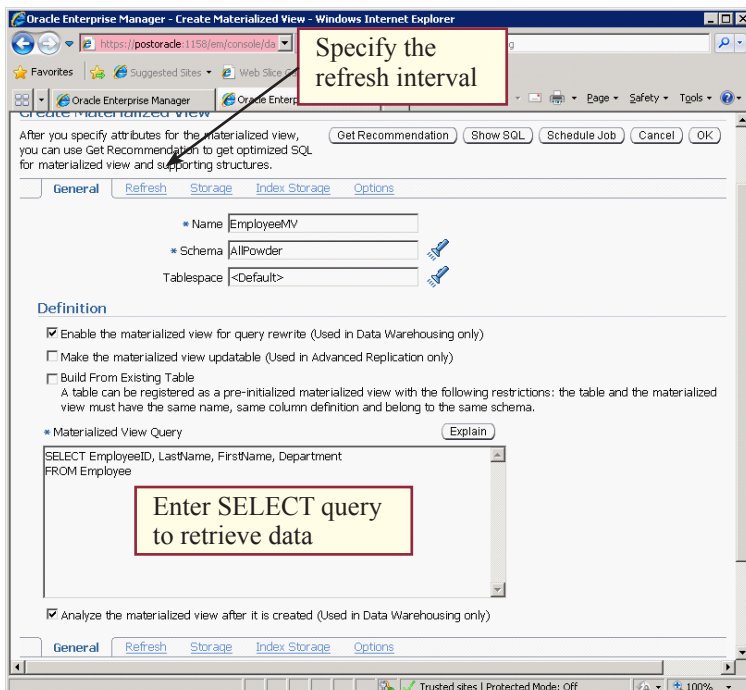
To create a replica, you simply create a query that extracts the data you need and store it as a materialized view in the replicated database. You then assign the materialized view to a refresh group to specify how often it should be updated. Figure 11.5 shows the basic process using the Enterprise Manager. However, it is straightforward to create the initial view using SQL. You can click the Show SQL button to see the options. If you do not have DBA permissions, you can simply type the SQL statements using SQL Plus. However, note that you will need the CREATE MATERIALIZED VIEW system privilege. The basic command is simply:

```
CREATE MATERIALIZED VIEW EmployeeMV USING INDEX REFRESH
FORCE ENABLE QUERY REWRITE AS SELECT...
```

You can select different options, but these are commonly used to speed up the refresh process.

After the materialized view has been defined, you can use the Enterprise Manager to check and modify the refresh timing. Figure 11.6 shows the basic options. For a replica, you will generally want to schedule the updates during a slow business time—usually overnight when the data transfers will not interfere with other activity. The frequency of the updates depends largely on the amount of data to be transferred. If it is going to take several hours to update the view, you will probably have to limit when the data can be transferred. On average, most people would probably prefer to update data at least once a day, but sometimes you can get by with out of date data. Other times, managers need relatively current data, so you might have to schedule updates more than once a day. Just make sure you test the system and monitor network traffic to ensure you do not interfere with other critical business activities.

Figure 11.5



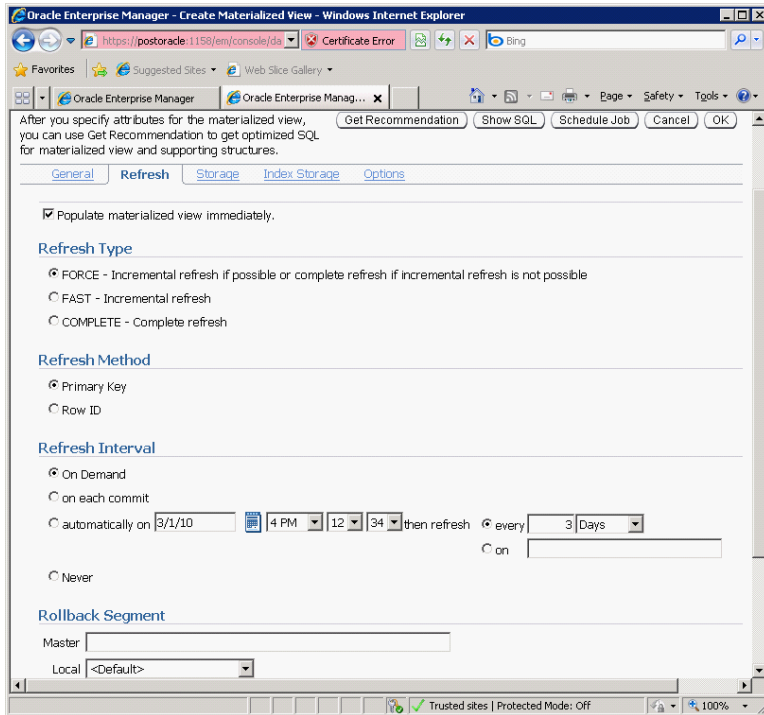


Figure 11.6

Note that with materialized views, you do not want users making changes to the distributed copies. If anyone does need to make a change to the data, it should be forwarded back to the original database. This way changes are centralized and they will all be distributed to the replicas on schedule. If you really do need to let people in the outer offices make changes to data, you should split the tables and make each group responsible for its own set of data. You can use materialized views to pull the local data back to the home office if necessary. Allowing changes to the same piece of data in different locations is challenging. What happens when two people alter the same piece of information? The DBMS will not recognize the collision until the materialized views are updated, and then it will not know which version to accept. Consequently, a person would have to be notified to make the decision about which change to keep. If a system needs human intervention, you might as well get them involved right at the start so they can control the process at the beginning—where they are less likely to make mistakes.

The Internet



Activity: Building Web Pages with JDeveloper

Today, the best way to handle distributed databases is to avoid them. Instead, get a good Web server with a high-speed connection. Build applications in JDeveloper as you did throughout this workbook. Give users simple PCs or tablets or even cell-phone Web browsers. The data and the applications stay in a single location making them easy to update, backup, and monitor. Users get mostly unlimited access from any location.

The only challenge to the Web-based approach is that you often have to deal with “old” applications. It is easy to move the database itself to a central server.

But it might take huge amounts of time and money to rewrite all of the existing applications as Web forms and reports. Still, it needs to be done eventually, and you can start with the applications that are used the most often by people who travel outside the main company.



Activity: Transferring Data with XML

One issue you will face with distributed databases is the need to transfer data among differing database systems. Note that SQL Developer can connect to other types of database systems, such as SQL Server. But you would need network access and permissions on that database. More likely, you need to import or export data to external databases. For example, a supplier might

send you product information electronically. Since the supplier does not know what type of database system you have or how your database is organized, it can be difficult to provide the data in a format that your system can read. The process is complicated when suppliers have thousands of customers like your shop. Suppliers have no desire to create thousands of different electronic files. Instead, they should be able to send one file in a standard format, and your system should be able to identify the necessary data, select it, and import it into your database. This dream is not quite reality, but XML (eXtensible Markup Language) was created to make it easier to exchange data among disparate systems.

Exporting data in XML format is relatively easy with Oracle. Oracle has several tools to scan a relational table and produce an XML format. The DBMS_XMLGen package will read or write an entire table and create the output file in one piece. It also offers some options to control the layout of the file. It can create nested subsections, such as an items order on an order form. Figure 11.7 shows the basic syntax for creating an XML file using four columns in the Employee table. One catch is that the SELECT command prints a heading at the top of the file, so you have to edit the output file and delete everything before the `<? xml ... ?>` line. The DBMS_XMLGen package can also be used to import data from a file.

SQL Developer has easier ways to export data in different formats. Whenever you run a query, you can right-click the results grid and choose the Export option. From there, you can select XML or CSV or Text among other options. You might have to edit the resulting file to clean it up, but that step is usually optional.

Action

Run the DBMS_xmlgen command to export some employee data.

Edit the file to remove the header.

Verify that it works using the browser.

Write an SQL query in SQL Developer and right-click the results to Export the data to an XML file.

Figure 11.7

```
spool C:\Database\EmployeeList.xml
SELECT DBMS_xmlgen.getXml(
'SELECT EmployeeID "EID",
  LastName "LastName",
  FirstName "FirstName",
  Department "Department"
FROM Employee'
,0 ) from dual;
spool off
```

```

<?xml version="1.0"?>
<ROWSET>
  <ROW>
    <EID>0</EID>
    <LastName>Staff</LastName>
  </ROW>
  <ROW>
    <EID>1</EID>
    <LastName>Killy</LastName>
    <FirstName>Jean-Claude</FirstName>
    <Department>Ski-Alpine</Department>
  </ROW>
  <ROW>
    <EID>2</EID>
    <LastName>Miyahira</LastName>
    <FirstName>Hideharu</FirstName>
    <Department>Ski-Alpine</Department>
  </ROW>
  ...
</ROWSET>

```

Figure 11.8

Figure 11.8 shows part of the resulting XML file for the small employee example. You might want to edit the file and change the names of some of the tags. But, the file can be read in its current form by any XML parser. It is probably worthwhile to change the starting and ending tag from `<ROWSET>` to `<EMPLOYEES>` to better indicate the data that is being transferred.

Figure 11.9

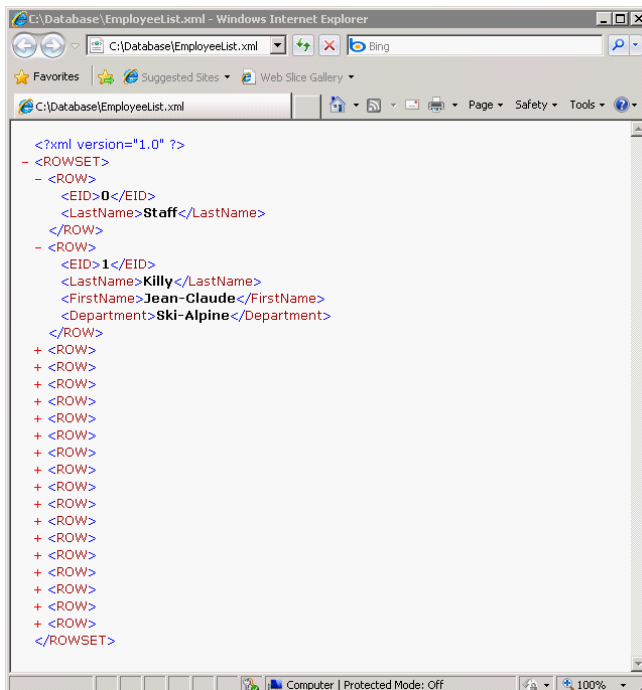


Figure 11.9 shows that you can open XML files using the Internet Explorer browser. This approach highlights the individual data records and makes it easy to see the structure of the data. You can expand or contract individual segments to focus on individual areas. It is a useful way to quickly check a file to ensure that it is consistent. It is also useful for browsing data sent from an external source so you can check the contents.

Oracle also has several methods to import data from XML files and place it into the database. In particular, you can create a column in a table using the XML data type and store the raw XML data directly in the table.

One of the biggest questions you will face in this situation is whether to store the data in raw XML form or to retrieve the data items from the XML file and store them in relational tables. If you are importing data to be used within your existing database, then you will generally want to extract the data and store it in the relational tables. Searching for data within an XML column requires the use of XQuery. It works, but it can be slow. If you need the data often, it is better to extract it from the XML and store it in relational tables.

On the other hand, if you are going to re-export the XML file, or simply need to extract a few items from it, or if it is needed for some other application, you will want to leave it in a special XML table. Oracle also supports XML as a data type for a column. You can load an XML table from a data file. Oracle then provides the `Extract` and `ExtractValue` commands to retrieve individual items from the XML structure.

Exercises



Crystal Tigers

Most of the information for the Crystal Tigers club can be maintained on one computer run by the club secretary. However, the secretary sometimes needs assistance entering all of the data during special events. Although he brings the database on his laptop, it would probably be easier if two or three people brought laptops and handled specific tasks. At the end of the day, the data could be synchronized and available for analysis. It would at least speed up the data entry and give more people access to the critical information needed during the day.

1. Replicate the database and test it on three separate computers, then synchronize the changes a few times to see if this approach will work for the club.
2. The club has talked about making some data available to members over the Internet. Although many of the members do not have Microsoft Office installed, the club would prefer to provide read-only access. Set up a page that generates activity lists for an upcoming event so members can check the schedule.
3. One of the charitable organizations the club works with is impressed with the database and would like some of the data. Create a query and export an XML file that lists the members and the hours worked for a particular event.



Capitol Artists

Because the system for Capitol Artists collects data from many employees at the same time, the main database needs to run on a central server. All of the computers

are connected by a high-speed LAN and, based on the company growth rates. The company is unlikely to open a second office; however, many of the employees have suggested that they would be more productive if they worked from home. The managers have suggested testing this idea by using the database work tracking system. Employees would connect to the database using the Web interface. As they completed client tasks, they would fill out the work table as usual. This data could then be synchronized with the company database at the end of the day. After a month, the managers could see if employee productivity declined or improved.

1. Check the performance of the database using an Internet connection from off-site. If possible, try it with a cable-modem connection and with a dial-up connection. Is the performance fast enough?
2. Outline the security issues involved in enabling employees to access the database from home over the Internet.
3. One of the owners travels often and wants to check on daily progress reports over the Internet using her laptop. Create a Web page that displays the work done for the current day and lists the hours and expense of the employees for each project.



Offshore Speed

The Offshore Speed company has some aspects in common with All Powder. In particular, the store needs several computers to access the application that handles sales, orders, and management reports. However, with the Web-based forms, the process is straightforward. On the other hand, the company deals with a huge number of parts, and it seems like vendors constantly change descriptions and prices. The company is trying to work with the vendors to connect to their databases and at least be able to retrieve replicated materialized views.

1. Set up a small new database that would be created by a vendor to hold information on parts. Replicate the table as read only so the Offshore Speed company can subscribe to it to automatically receive changes on a regular basis.
2. Some of the company's partner firms would like to receive files that they can read into their databases or into Excel. Set up a procedure that will create text files with basic order data for a selected partner.
3. Create a Web page that customers can use to check on the status of their orders. You should create a separate password for the customers that will be stored within the Customer table. Verify that the password and order number are correct before displaying the data.

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you or your instructor picks one, perform the following tasks.

1. Describe any distributed features or database links that will be useful to the project and list any problems you might encounter.
2. Create a replica and test all of the forms and reports on both copies. Test the synchronization.

3. Export at least one table into an XML file that could be sent to an outside firm such as a customer or supplier.
4. Create a basic Web form and response page that enables customers (or employees) to enter some identifier and receive additional information. For example, a customer might select a product category and receive a list of products in that category.
5. Create a second database and build a link to that database, so at least one form operates using data in the second database.

Chapter 12

Physical Database Design

Chapter Outline

Storing Data, 283

Case: All Powder Board and Ski Shop, 284

Lab Exercise, 285

All Powder Board and Ski Shop, 285

Data Clusters, 288

Exercises, 291

Final Project, 292

Objectives

- Configure tablespaces.
- Configure the various data storage methods.
- Establish data clusters and partitions.

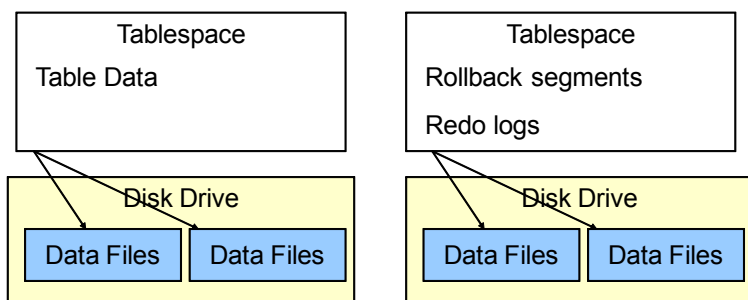
Storing Data

To improve performance, Oracle provides several methods to control the physical storage of data. Most tables can be created with default options that function reasonably well for typical data. The default methods use B+tree storage, which is the best overall storage method. However, you sometimes need to configure the actual storage—particularly through choosing the location and expansion characteristics of tablespaces. Oracle also provides alternate storage methods such as indexed tables and hashed access. You can also assign clustering to indexes, or create partitions to separate data to improve performance. These options are generally used for very large databases, so it is difficult to gauge their effect using the small sample sets of data. Nonetheless, you should understand how to create and use the various tools so you know the options are available as your database grows. Indexed tables extend the concept of B+trees by storing the data in the index itself—reducing the time needed to retrieve the actual data.

Recall from Chapter 10 that Oracle uses tablespaces as logical folders that can utilize multiple data files. Figure 12.1 shows that tables are assigned a specific tablespace to store the data. Oracle also uses rollback segments and redo logs to handle transactions and other situations where device failure might cause serious problems. You get a substantial gain in performance and safety if you store the table data and rollback segments in separate tablespaces on different drives. Two drives spinning independently means (1) the computer can write the data simultaneously, and (2) there is less chance of a loss in the event of a hardware failure. Oracle provides a tablespace map tool to help DBAs monitor the current storage allocation.

As shown in Figure 12.2, within a tablespace, Oracle stores data in data blocks. You can specify the fixed data block size when you create a database (`DB_BLOCK_SIZE=8192`), but generally, you use the default value (typically 8K bytes). A collection of data blocks is called an extent, which is internally managed by Oracle. A segment consists of a collection of extents, and segments always contain related data. For instance, a data segment specifically contains row data. These concepts are useful to know when you evaluate performance. For instance, all data within a data block is retrieved in one pass from the disk drive; which is why its size is tied to the operating system parameters. As rows are inserted into a table, they are added to a specific data block. However, the DBMS plans ahead for potential changes due to updates. If a data block runs out of space while updating a row, the data has to be move moved to a new data block, which takes time. Consequently, as the DBMS inserts rows, it will switch to a new data block when the existing one reaches the `PCTFREE` parameter. For instance, if `PCTFREE` is

Figure 12.1



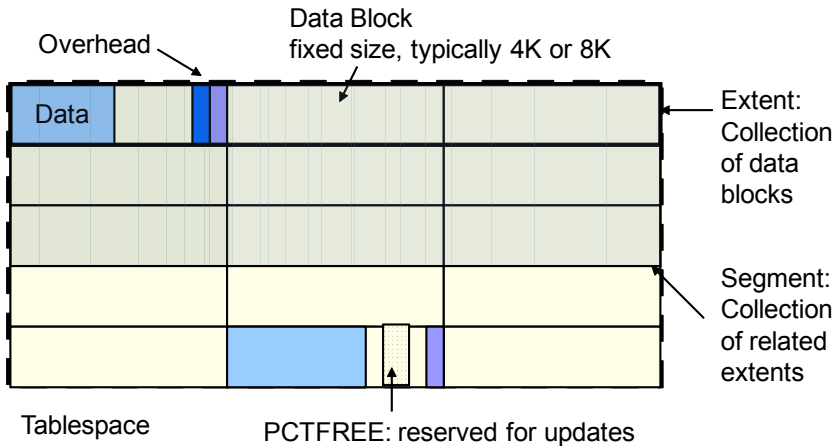


Figure 12.2

set to 20 percent and a data block has only 18 percent free space, no more rows will be added to that data block. Going the other direction, when rows are deleted from a table, space becomes freed up in the data block. Should the DBMS begin adding new rows to the block as soon as the PCTFREE threshold is opened? If the DBMS followed this policy, it would constantly be deleting one row and adding one row to a data block. Instead, the PCTUSED threshold is the indicator for when it is safe to begin adding rows again. When enough rows have been deleted to drop below the PCTUSED value, the DBMS will switch to adding new rows to the data block. The DBMS uses default values for these two parameters, but you can override them when you create a table. For instance, if you know that a table typically gets many inserts, but few deletes or updates, you can specify a low value for PCTFREE to provide more efficient use of the storage space along with faster retrievals.

Case: All Powder Board and Ski Shop

Although the store has hundreds of different products and sells or rents dozens of items a day, the standard storage defaults should work well for several years. On the other hand, it is always a good idea to be prepared and practice with configuring different storage options now. If a problem arises in the future, you will have notes on how to handle the issue or at least be able to set up the configuration quickly.

The storage parameters for tablespaces and data blocks are the easiest to configure. If you have administrator rights, you can use the Enterprise Manager tool to design new tablespaces and tables. It contains screens to select the options from lists and automatically generates the correct syntax. If you want to read more about the storage options and their effects, you can find the syntax in the Oracle SQL Reference Guide. The Oracle Database Administrator's Guide contains detailed descriptions of how to configure the storage parameters, including when you should and should not use various settings. Figure 12.3 shows the basic values recommended in the Oracle documentation.

Settings	Table Attributes
PCTFREE=10 PCTUSED=40	Default.
PCTFREE=20 PCTUSED=40	UPDATE statements that tend to increase row size.
PCTFREE=5 PCTUSED=60	Row size is constant.
PCTFREE=5 PCTUSED=40	Most activity is INSERT or read only.

Figure 12.3

Lab Exercise

All Powder Board and Ski Shop

The main text explains that data can be stored using several methods, including sequential, B+trees, and direct or hashed storage. For the most part, the Oracle DMBS stores data similar to linked lists using the data blocks. By default, Oracle indexes any primary key columns, and the indexes are stored and searched using B+trees. This approach is the best general storage method, but sometimes you might need more control over how the data is stored and retrieved.

Oracle 11g handles the other storage methods through partitions and clusters. Partitions enable you to store different parts of a table in different locations. This trick can be useful even for relatively small databases. In the All Powder case, the inventory data tends to change every year as new products are introduced. You do not want to delete the old data because the managers want to go back and look at sales in various categories. On the other hand, you do not need it taking up space on the main disk drives—because the old models are no longer available for sale or rental. A partition enables you to move the older data to a different disk drive.



Activity: Create Data Partitions

Partitioning enables you to split a data table into multiple pieces. Each piece contains the same types of data (same column names and same data types), but can have different physical parameters (tablespace, PCTFREE, and so on). The tablespace option is the most powerful choice because it enables you to store one part of the table in one location (disk drive) and the rest in other locations. Placing data in different tablespaces also improves query performance because the DBMS can restrict the search to a single partition. It also improves backup and recovery operations since you can tell the DBMS to operate on a tablespace or partition at a time. Finally, the partition can be invisible to the query system and the users. Existing queries and applications will continue to work correctly with no changes. However, Oracle does place one important restriction on partitioning tables: You cannot parti-

Action

- Create the p2007 tablespace.
- Create the new Sale2 table with a range interval partition.
- Transfer the existing data into the new table and check the tablespace usage.
- Use the USER_PART_TABLES view to test your work.

tion a table that contains a LONG or LONG RAW data type column. Instead, you would need to convert these data types to CLOB or BLOB data types.

Partitions are defined based on the data contained in the row. Oracle supports three types of partitions: range, list, and hashed. A couple of composite types are also supported, but they are not covered here because they are simply combinations of the three base types. Range partitions are easy to understand and are often used for date or ID columns. You choose a data column and split the rows based on ranges of data. Oracle 11g improved the range partitions by implementing an INTERVAL partition process. You can define an interval in terms of a date based on either months or years. The power of the INTERVAL definition is that Oracle automatically creates a new partition when the date enters into a new time period. Before 11g, you had to define complete partitions for every year (or month) that might exist in the data. Figure 12.4 shows the syntax for defining the partition and interval for the Sales table.

Of course, you should create the first tablespace before trying to insert data into this new table. You can use the Enterprise Manager to create the tablespaces. You can use the EM interface to specify the options and then use the Show SQL command to see the full syntax:

```
CREATE SMALLFILE TABLESPACE " p2007"
DATAFILE 'D:\ORACLE\PRODUCT\11.1.1\ORADATA\POSTDB\p2007'
SIZE 10M
AUTOEXTEND ON NEXT 5M MAXSIZE UNLIMITED LOGGING EXTENT
MANAGEMENT
LOCAL SEGMENT SPACE MANAGEMENT AUTO
```

Figure 12.4

```
CREATE TABLE Sale2
(
SaleID      INTEGER,
SaleDate    DATE,
CustomerID  INTEGER DEFAULT 0,
EmployeeID  INTEGER DEFAULT 0,
ShipAddress NVARCHAR2(50),
ShipCity    NVARCHAR2(50),
ShipState   NVARCHAR2(50),
ShipZIP     NVARCHAR2(50),
SalesTax    NUMBER(38,4) DEFAULT 0,
PaymentMethod NVARCHAR2(50),
    CONSTRAINT pk_Sale2 PRIMARY KEY (SaleID),
    CONSTRAINT fk_CustomerSale2 FOREIGN KEY (CustomerID)
        REFERENCES Customer(CustomerID)
        ON DELETE CASCADE,
    CONSTRAINT fk_PaymentMethodSale2 FOREIGN KEY (PaymentMethod)
        REFERENCES PaymentMethod(PaymentMethod)
        ON DELETE CASCADE
)
PARTITION BY RANGE (SaleDate)
INTERVAL (NUMTOYMINTERVAL(1,'year'))
(
    PARTITION p2007 VALUES LESS THAN (to_date('2008/01/01','yyyy/mm/dd'))
);
```

Observe that the command also defines a new operating system file to hold the tablespace. In most cases, each of the tablespaces will be assigned to different folders and probably to different physical disk drives. The physical file parameter is the mechanism you use to control the location of the data. The most recent data would be stored on a high-speed drive—probably a RAID system. The other files can be stored on slower, less expensive drives because the data will be accessed less often and rarely altered.

Once you have the initial tablespace defined, you can run the CREATE TABLE command to build the Sale2 table. To save time, you can ignore the referential integrity constraints. You can use an INSERT INTO command to transfer the data from the existing table into the new one. Once the data has been inserted, you can use the EM to check that data has been inserted into each of the tablespaces. You can use the ALTER TABLE ADD PARTITION command to add more partitions later. The DBA_PART_TABLES or USER_PART_TABLES system views will display information about partitioned tables so you can check your work. Note that range partitions can include multiple columns, and you can find the syntax in the Oracle Administrators Guide.

List partitions are similar to range partitions, but you use a list when no continuous range of values exists. For example, a list partition is commonly used to separate data by geographical region such as state or province. Figure 12.5 shows that the syntax is similar to that used for the range partition. You have to individually specify the list items that apply to each partition. Eventually, you would have to add the other state codes. Also, remember that you need to create the tablespaces before you try to run the CREATE TABLE command.

Figure 12.5

```
CREATE TABLE Customer2
(
    CustomerID    INTEGER,
    LastName      NVARCHAR2(50),
    FirstName     NVARCHAR2(50),
    Phone         NVARCHAR2(50),
    Address       NVARCHAR2(50),
    Email        NVARCHAR2(50),
    City         NVARCHAR2(50),
    State        NVARCHAR2(50),
    ZIP          NVARCHAR2(50),
    Gender       NVARCHAR2(50),
    DateOfBirth  DATE,
    CONSTRAINT pk_Customer2 PRIMARY KEY (CustomerID)
)
PARTITION BY LIST (State)
(
    PARTITION WestSales VALUES ('CA', 'HI', 'OR', 'AZ')
        TABLESPACE CustomerWest,
    PARTITION EastSales VALUES ('RI', 'CT', 'NY')
        TABLESPACE CustomerEast,
    PARTITION SouthSales VALUES ('FL', 'GA', 'LA')
        TABLESPACE CustomerSouth,
    PARTITION MidSales VALUES ('WI', 'MI', 'MN', 'OH')
        TABLESPACE CustomerMid
);
```

```

CREATE TABLE Employee2
(
    EmployeeID    INTEGER,
    LastName      NVARCHAR2(50),
    FirstName     NVARCHAR2(50),
                CONSTRAINT pk_Employee2 PRIMARY KEY (EmployeeID)
)
PARTITION BY HASH (EmployeeID)
PARTITIONS 4
STORE IN (tsEmp1, tsEmp2, tsEmp3, tsEmp4)
;

```

Figure 12.6

Hash partitioning is the third basic method and it is simpler than the other two because it contains fewer options. Its purpose is to randomly distribute the items relatively equally across the specified partitions. This approach might be useful if your operating system does not support RAID striping. You could assign each tablespace to a different physical drive and let Oracle assign each row to a different drive based on the ID value. You also improve performance because Oracle uses the hash function to identify the needed partition and reduce its search time. Figure 12.6 shows the basic syntax. Essentially, you specify the number of desired partitions and then list their names. A couple of options exist, such as specifying the initial data size, but most of the work is handled automatically by Oracle.

Data Clusters



Activity: Create Data Clusters

Clusters are different from partitions—the goal is to store related data close together. Really close together. Remember that disk drives are the slowest component of the computer (not counting interfaces with people), because they rely on mechanical elements. Data that is stored in different locations on the drive take time to retrieve because the drive head has to wait for the sector to spin around. The goal of clustering is to reduce this delay by storing related data together so that it can be retrieved in one pass. Data block size in Oracle is generally tied to the operating system capabilities so that an entire data block can be read in a single call to the disk drive. In the All Powder example, look at the Sale and SaleItem tables. In almost every case, users will want to retrieve data from both the Sale and SaleItem table at the same time. You can improve performance by telling Oracle to cluster the data for each SaleID. For instance, the SaleItem data for SaleID=101 will be stored in the same data block as the base Sale data for SaleID=101.

It takes three basic steps to cluster data. (1) Create the cluster, (2) Create the two tables and assign them to the same cluster with the same key, (3) Create an index on the cluster. Once this structure has been defined, you can add data to the tables. Note that because you are controlling the way data is physically stored,

Action

- Create a cluster based on the SaleID.
- Define a new SaleC table that uses the cluster.
- Define a new SaleItemC table that uses the same cluster.
- Create the index for the cluster.
- Insert some rows into the two new tables and see if there is a difference.

```
CREATE CLUSTER Sale_SaleItem (SaleID INTEGER)
PCTUSED 80
PCTFREE 5
SIZE 130
TABLESPACE ts1;
```

Figure 12.7

you must create the cluster before you add data to the table. If you want to cluster tables that have existing data, you will have to create new tables for the cluster and transfer the data from the old tables.

Figure 12.7 shows the CREATE CLUSTER command used to define the overall cluster. Notice that it does not include the names of the tables.

The cluster definition simply specifies

the storage space needed along with the name of the tablespace. You do need to specify the data type of the key value (SaleID is an INTEGER) that will be used to cluster the data. You can use multiple columns for the key, but you cannot use the LONG data type. The SIZE parameter provides a guide to Oracle so that it knows the approximate amount of space that will be needed to store the key and the associated data rows. By default, Oracle uses an entire data block to hold data for one key value. If the data does not fit in one block, Oracle chains them together to speed the retrieval. This approach will waste considerable space if your rows are relatively small and few rows exist for each key value. By using the SIZE parameter to specify the average amount of data bytes, Oracle can allocate the blocks more efficiently.

Action

- Create a cluster based on the SaleID.
- Define a new SaleC table that uses the cluster.
- Define a new SaleItemC table that uses the same cluster.
- Create the index for the cluster.
- Insert some rows into the two new tables and see if there is a difference.

Figure 12.8

```
CREATE TABLE SaleC
(
    SaleID INTEGER,
    SaleDate DATE,
    CustomerID INTEGER,
    -- other columns
    CONSTRAINT pk_Sale PRIMARY KEY (SaleID),
    -- other constraints
)
CLUSTER Sale_SaleItem (SaleID);
CREATE TABLE SaleItemC
(
    SaleID INTEGER,
    SKU NVARCHAR2(50),
    -- other columns
    CONSTRAINT pk_SaleItem PRIMARY KEY (SaleID, SKU),
    -- other constraints
)
CLUSTER Sale_SaleItem (SaleID);
```

```
CREATE INDEX Sale_SaleItem_index
ON CLUSTER Sale_SaleItem
TABLESPACE ts1
PCTFREE 5;
```

Figure 12.9

Creating tables is straightforward—you just need to specify the cluster that will hold the table data. Figure 12.8 shows the syntax. You simply add one line at the end to indicate the name of the cluster along with the key value from the table that will be passed to the cluster key. You repeat the process for the second (SaleItem) table.

The third step is to create an index on the cluster itself. This step is required and must be processed before you try to add data to either of the tables. Figure 12.9 shows that the syntax is similar to the standard CREATE INDEX command, but you specify the name of the cluster instead of the column names. You can also set storage parameters that are not shown. You can choose a different tablespace so that the index is stored and processed on a separate disk drive to improve update performance.

When you have completed these three steps, Oracle handles everything else. Your INSERT, UPDATE, SELECT, and DELETE statements will all work the way they always have. The difference is that Oracle changes the way the data is physically stored. Bear in mind that clustering is not guaranteed to be faster. In fact, if your application routinely updates or searches the tables separately, performance will be better without clusters.

You can now insert data into the two new tables. If your database server has a really slow disk drive, you might be able to observe a difference in query performance using the clustered tables. But, in most cases, with a small amount of data, you will not perceive the difference.



Activity: Create Hashed Data Clusters

Oracle also uses clusters to handle hashed data access. You create a hash cluster to define the storage locations and add the tables to the cluster. Oracle computes a hash key from the key values to identify the specific storage area. The hash value corresponds to a specific data block that holds the desired data. Remember that hashed data storage is most useful when searches are performed with equality tests against the key value (e.g., SKU=XEN-758). The table should also be relatively static, so that you can estimate the amount of space to allocate. Hashed clusters can be used effectively with a single table—where the primary key is used to locate each individual row. Hashed-key tables are useful for situations where you will always know the primary key and require rapid access. Bar-coded product numbers are a classic example. Queries with JOINS to this table also perform faster.

All Powder wants to implement a scanner to read inventory items when they are purchased. With several scanners in a store operating at the same time, the

Action

- Create the hashed cluster for the single Inventory table.
- Create the new Inventory table and assign it to the cluster.
- Transfer data into the new table.
- Test some queries using exact key values.
- Test some queries with inequalities.


```

CREATE CLUSTER Inventory_hash (SKU NVARCHAR2(50))
SIZE 32 SINGLE TABLE HASHKEYS 5000;

CREATE TABLE InventoryH
(
    SKU      NVARCHAR2(50),
    ModelID NVARCHAR2(50),
    ItemSize NUMBER(20,2),
    QuantityOnHand INTEGER,
    CONSTRAINT pk_Inventory PRIMARY KEY (SKU),
    -- fk constraint
)
CLUSTER Inventory_hash (SKU);

```

Figure 12.10

application needs fast access to the Inventory table. Since a hash cluster does not use an index, you only need two steps. First you create the cluster then you create the table and assign it to the cluster. You have only a few options when creating the cluster. You might want to review the main text to understand the way that hashed-key access works, which helps explain the role of the main parameters. The main point is that the system needs to know the approximate number of rows to store. This value is automatically rounded up to the nearest prime number. To allocate space in the data block, the DBMS also wants an estimate of the size of an average row of data. Figure 12.10 shows the cluster and table commands to create the new InventoryH table.

Transfer data from the old Inventory table into the new InventoryH table and test some queries. Unless your computer and disk drives are incredibly slow, it will be difficult to perceive a difference when using equality constraints on the SKU. But, you might be able to perceive a difference if you select rows based on inequality tests or even LIKE statements.

Exercises



Many Charms

The database for Many Charms is likely to remain relatively small and performance should not be a serious issue. Nonetheless, you should look for possible ways to improve performance by controlling the data storage.

1. Assuming the company becomes substantially larger, what storage strategies would be useful?
2. Create a hashed cluster for the ItemList table.
3. Create a cluster to store matched data for the Customer, Sale, and SaleItem tables.
4. Partition the Production table into two sections based on the ProductionDate.



Standup Foods

Standup Foods has the potential to grow to a relatively large company over the next couple of years. It is possible that performance will become an issue with some of the tables. The client list is particularly interesting, because studios are continually creating new companies and partnerships. As a result, many of the

older companies in the list no longer exist. On the other hand, the contact list is important, since it contains data on individual people. Similarly, the Employee list changes on an almost daily basis. Laura is reluctant to delete the older employees because many of them come back for special projects every couple of years.

1. Identify the tables that could be improved using partitions or clusters. Explain your reasoning.
2. Partition the project table into three sections based on the contract date.
3. Create a cluster that stores data by EmployeeID that includes the Employee, EmployeeSpecialty, TaskSpecialty, and ProjectEmployee tables.
4. Create a hashed cluster for the ProjectGuest table.



EnviroSpeed

The database for EnviroSpeed could eventually become quite large. Because the system contains valuable knowledge, the company does not want to delete anything. The company also benefits by keeping all of the data in one large database. Although much of the data becomes dated, employees still want the ability to search through older cases. However, the older data does not change so it could be moved to different disk drives.

1. Identify the tables that could be improved using partitions or clusters. Explain your reasoning.
2. Partition the Situation and ProposedSolution tables into three segments based on the date.
3. Partition the Crew table into four regions based on Country.
4. Create a hashed cluster for the Bill and Receipts tables

Final Project

The main textbook has an online appendix with several longer case studies. You should be able to work on one of these cases throughout the term. If you pick one or your instructor picks one, perform the following tasks.

1. Identify the tables that could be improved using partitions or clusters. Explain your reasoning.
2. Create a partition on at least one table.
3. Create a cluster on at least two tables.
4. Create a hash cluster on a single table.