Database Management Systems

Sixth Edition

Designing & Building Business Applications

Gerald V. Post

Database Management Systems

Designing and Building Business Applications

Version 6.0.0

Gerald V. Post

University of the Pacific

Database Management Systems
Designing and Building Business Applications

Preface

Goals and Philosophy

Working with business information systems is exciting. You get to work with people to find ways to improve their jobs. You get to use your creativity to build new applications. You often work on teams to share ideas and techniques. In the end, you get to see an application that you built help companies and people. Database management systems are key components in almost every business application. They organize and store the data so it can be retrieved and analyzed.

The goal of this text is straightforward: At the end of the text students should be able to evaluate a business situation and build a database application.

Building an application requires four basic steps. The book is organized by these four steps: (1) Database Design, (2) SQL Queries, (3) Application forms and reports, and (4) Database administration.

The first two steps (design and SQL) are standard for almost every database management system. Normalization shows how to carefully design databases to gain the strengths of the database approach. SQL is a standard query language that is used for virtually every step of application development. These two topics must be covered carefully and thoroughly, particularly because they are both difficult topics for students.

The concepts of forms and reports are relatively standard, but every database system has different tools to build applications. Similarly, database administrators perform relatively standard tasks at every company, but the tools are different for every system. The main textbook explains the basic tasks in neutral terms that apply to every database management system.

Workbooks

Students need to do more than learn the basic concepts and theory. They need to actually build an application. A database management system (DBMS) is a complex tool with thousands of options and individual quirks. The accompanying workbooks are designed to show students how to build and application using a particular DBMS. The workbooks follow the main text chapters but focus on creating an application for a specific company.

The first step in using the Workbooks is to choose a DBMS. Be sure all of the components of the DBMS are available and installed. In particular, Oracle requires several components and Microsoft SQL Server needs Visual Studio. A nice feature of the Workbooks is that they all use the same case. So students can learn one DBMS and then later use a different workbook to learn the details of a new DBMS.

Learning Assessment

Learning assessment is important to students as well as faculty and employers. Students need to determine what aspects they are strong in and which ones need additional work. Students need to understand that if they can successfully learn this material, they will have acquired several skills that will get them jobs and help them contribute to businesses by being able to quickly build and maintain business applications.

Learning assessment in this book is straightforward: At the end of the book, students should be able to analyze a business situation and develop a database application. The complexity of the application and tools used will depend on the specific class and the background of the students. Students should develop a term-project as part of a course. Several sample projects are included as a separate project book and several more are in the workbooks. The project provides an excellent opportunity to assess overall learning. The final project can be evaluated in terms of (1) correctly meeting the business needs, (2) an efficient database structure, and (3) usability.

It is also useful to assess individual skills independently—particularly if groups are used to create the final project. In this case, assessment consists of individual exams for (1) database design and normalization, (2) SQL and creating queries from business questions, and (3) selected topics including database programming, security, data mining, and distributed systems.

Organization

The organization of the text follows the basic steps of application development: design, queries, applications, administration, and advanced topics. Some instructors might prefer to teach queries before database design, so the initial chapters are written with that flexibility.

Chapter 1: Introduction

Part 1: Systems Design
- Chapter 2: Database Design
- Appendix: Database Design System
- Chapter 3: Data Normalization
- Appendix: Formal Definitions of Normalization

Part 2: Queries
- Chapter 4: Data Queries
- Appendix: SQL Syntax
- Chapter 5: Advanced Queries and Subqueries
- Appendix: Introduction to Programming

Part 3: Applications
- Chapter 6: Forms and Reports
- Chapter 7: Database Integrity and Transactions
- Chapter 8: Applications
- Chapter 9: Data Warehouses and Data Mining

Part 4: Database Administration and New Systems
- Chapter 10: Database Administration
- Chapter 11: Distributed Databases
- Chapter 12: Physical Data Storage
- Chapter 13: Non-Relational Databases

The introduction explains the importance of databases and relates database applications to topics the students have likely seen in other classes.

The section on database design has two chapters: Chapter 2 on general design techniques (systems techniques, diagramming, and control) and Chapter 3, which details data normalization. Chapter 2 leans towards an object and graphical approach, while Chapter 3 emphasizes normalization rules. The objective is to cover design early in the term so that students can get started on their end-of-term projects. Students should use the online Database Design system to work on exercises for both chapters to obtain feedback.

Queries are covered in two chapters. Chapter 4 introduces queries and focuses on the fundamentals of converting business questions to SQL queries. Chapter 5 discusses more complex queries. including subqueries and outer joins.

Part 3 describes the development of database applications, beginning with the essentials of building forms and reports in Chapter 6. Chapter 7 examines the common problems created in a multiuser environment. It explains the techniques used to handle data integrity and transactions. Chapter 8 shows how to put everything together to build a complete application, including navigation and help files. Chapter 9 explains why analytical processing requires a different database configuration than transaction processing. It covers the main tools for analysis and data mining in a nonstatistical context.

Part 4 examines various topics in database administration and new tools. Chapter 10 examines management issues emphasizing planning, implementation, performance, and security. It explains the major tasks and controls needed by an administrator. Chapter 11 investigates the growing importance of providing distributed access to databases. It examines the impact of various network configurations. Chapter 12 leans toward computer science when it looks at how the DBMS physically stores data. The last two chapters can be difficult to cover in a single-term course, but are presented at an introductory level. Chapter 13 is new with the sixth edition and introduces contemporary non-relational databases typically used for high-scalability, massively parallel Web-based tasks. The design, queries, and tradeoffs are illustrated with the Cassandra DBMS.

Additionally, four chapters have appendixes that discuss programming concepts that are more technical. The appendix to Chapter 2 describes the online database design system that is available to instructors and students. It provides immediate feedback on database designs, making it easier for students to understand the problems and explore different designs. The appendix to Chapter 3 presents the formal definitions of normalization. They are provided for instructors and students who want to see the more formal set-theory definitions. The appendix to Chapter 4 is a convenient list of the primary SQL statements. The appendix to Chapter 5 provides an introduction to programming. It is designed as a summary or simple reminder notes.

Pedagogy

The educational goal of the text is straightforward and emphasized in every chapter: By the end of the text, students should be able to build business applications using a DBMS. Throughout the text, many examples are used to apply and illustrate the concepts. The Web site also provides several databases so students can work with data, queries, forms, and applications. Students should be encouraged to apply the knowledge from each chapter by solving the exercises and working on their final projects.

Each chapter contains several sections to assist in understanding the material and in applying it to the design and creation of business applications:

- **What You Will Learn in This Chapter.** A list of questions that are answered within the chapter. Each question is echoed at the start of a section.
- **A Developer's View**. A student's perspective of the chapter contents.
- **Chapter Summary**. A brief review of the chapter topics.
- **A Developer's View**. A short summary of how the material in the chapter applies to building applications.
- **Getting Started**. A short statement of the main goal of the chapter focused on how the chapter contributes to designing and building applications.
- **Key Words**. A list of words introduced in the chapter. A full glossary is provided at the end of the text.

- **Additional Reading**. References for more detailed investigation of the topics.
- **Website References**. Some sites provide detailed information on the topic. Some are newsgroups where developers share questions and tips.
- **Review Questions**. Designed as a study guide for the exams, with a focus on the major topics within the chapter.
- **Exercises**. Problems that apply the concepts presented in the chapter. Most require the use of a DBMS.
- **Projects**. Several longer projects are available in a separate online document. They are suitable for an end-of-term project.
- **Workbooks**. Each workbook outlines the steps to build an application using a specific DBMS. Each workbook chapter illustrates tasks that match the discussion in the textbook. The workbook also provides exercises to build six other databases for different companies.
- **Sample Databases**. Three sample databases are provided to illustrate the concepts. Sally's Pet Store illustrates a database in the early design stages, whereas Rolling Thunder Bicycles presents a more finished application, complete with realistic data. Corner Med is a database for a neighborhood medical facility that tracks patient and physician interactions. Exercises for all three databases are provided in the chapters. The sample databases can be installed in several DBMS formats.

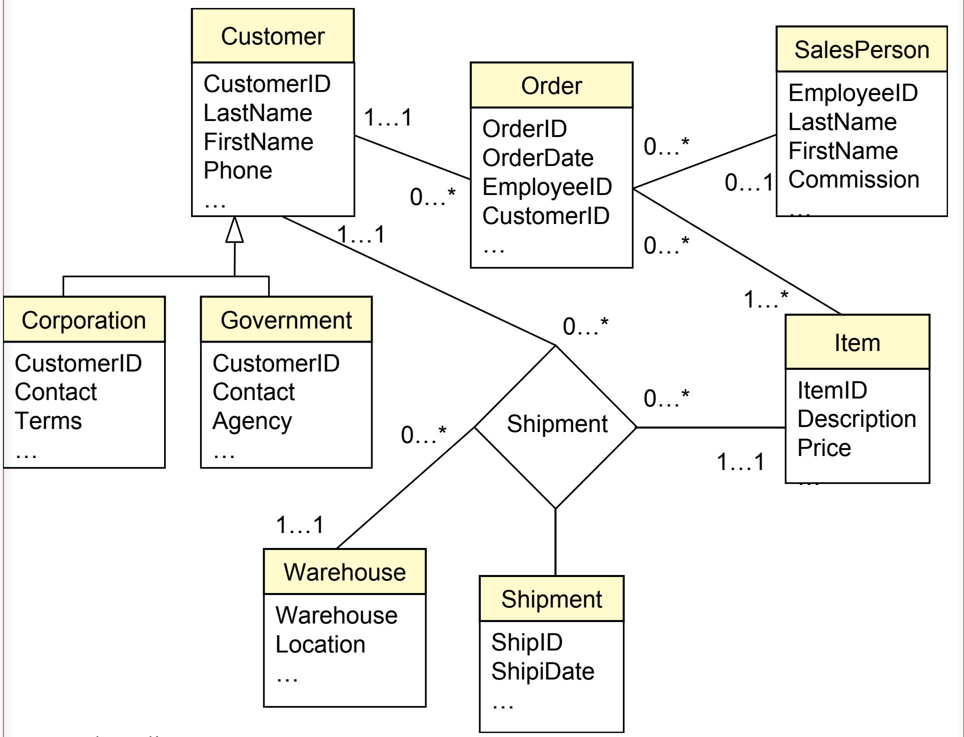
Features of the Text

1. Focus on modern business application development.
   - Database design explained in terms of business modeling.
   - Application hands-on emphasis with many examples and exercises.
   - Emphasis on modern graphical user interface applications.
   - Chapters on database programming and application development.
   - Introduction to data mining.
   - Some answers to exercises are provided to students through the online system. The exercises with answers are highlighted in the text with a check mark icon. Sometimes students need to see a solved problem.
2. Hot topics.
   - Description and use of the unified modeling language (UML) for modeling and system diagrams.
   - In-depth discussion of security topics in a database environment.
   - Development of databases for the Internet and intranets.
   - Emphasis on SQL 92, with an introduction to SQL 99 and the XML features of SQL 2003 and SQL 2008.
   - Integrated applications and objects in databases.
   - Introduction to non-relational (NoSQL) systems such as Cassandra.

3. Applied business exercises and cases.
   - Many database design problems.
   - Exercises covering all aspects of application development.
   - Sample cases suitable for end-of-term projects.
4. A complete sample database application (Rolling Thunder Bicycles).
   - Fully functional business database.
   - Sample data and data generator routines.
   - Program code to illustrate common database operations.
5. Two additional databases (Sally's Pet Store and Corner Med) for comparison and additional assignments.
6. Lecture notes as PowerPoint slide show.
7. Hundreds of database exercises and problems for students to work on.
8. Workbooks built for specific database technologies that illustrate the hands-on steps needed to build an actual application. Check the online site for versions of the workbooks for additional systems.

End-of-Term Projects

Several projects are described in the project document available online. These cases are suitable for end-of-term projects. Students should be able to build a complete application in one term. The grading focus should be on the final project. However, the instructor should evaluate at least two intermediate stages: (1) a



UML class diagram.

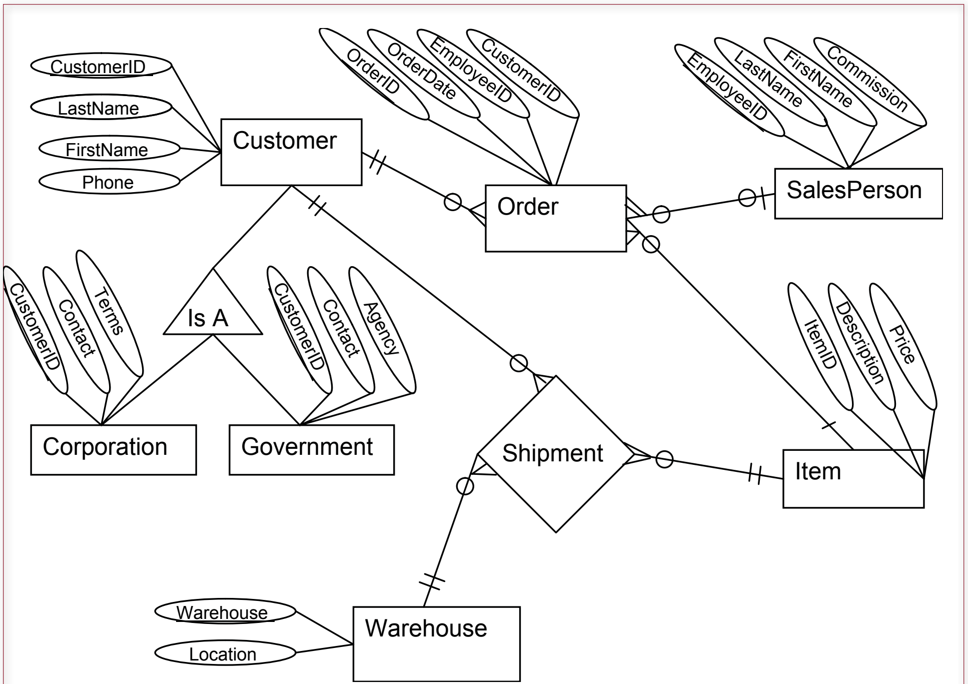
list of the normalized tables collected shortly after Chapter 3 is completed and (2) a design preview consisting of at least two major forms and two reports collected shortly after Chapter 6. The six additional cases in the workbook can also be used as an end-of-term project.

Some instructors may choose to assign the projects as group assignments. However, it is often wiser to avoid this approach and require individual work. The project is a key learning tool. If some members of the group avoid working on the project, they will lose an important learning opportunity.

Database Design and the Unified Modeling Language

For several years, entity-relationship diagrams were the predominant modeling technique for database design. However, this approach causes problems for instructors (and students) because there are several different diagramming techniques. This edition continues to help solve these problems by incorporating the Unified Modeling Language (UML) method, instead of traditional entity-relationship (ER) diagramming, as the modeling technique for database design. This change will be most apparent in the replacement of the ER diagram notation and terminology with the parallel concepts in UML class diagrams.

UML class diagrams, although very similar to ER diagrams, are superior in several ways. First, they are standardized, so students (and instructors) need learn only one set of notations. Second, they are "cleaner" in the sense that they are easier to read without the bubbles and cryptic notations of traditional ER diagrams. Third, they provide an introduction to object-oriented design, so students will be better prepared for future development issues. Fourth, with the rapid adoption of UML as a standard design methodology, students will be better prepared to move



Corresponding Entity Relationship diagram.

into future jobs. Many of the systems design organizations have adopted UML as a standard method for designing systems. UML has the support of major authors in systems design (e.g., Booch, Rumbaugh, and Jacobsen) as well as being supported by the major software development firms including IBM, Microsoft, and Oracle. Note that Microsoft Access and SQL Server both use a diagramming tool that is similar to UML. In addition, students should have little difficulty transferring their knowledge of the UML method if they need to work with older ER methods.

The basic similarities between ER and class diagrams are (1) entities (classes) are drawn as boxes, (2) binary relationships (associations) are drawn as connecting lines, and (3) N-ary associations (relationships) are drawn as diamonds. Hence, the overall structures are similar. The main differences between UML and ER diagrams occur in the details. In UML the multiplicity of an association is shown as simple numerical notation instead of as a cryptic icon. An example is shown in the accompanying figures.

UML also has provisions for n-ary associations and allows associations to be defined as classes. There are provisions for naming all associations, including directional names to assist in reading the diagram. Several situations have defined icons for the association ends, such as composition (rarely handled by ER) and subtypes (poorly handled by ER).

More details of the UML approach are shown in Chapters 2 and 3. Only a small fraction of the UML diagrams, notation, and terminology will be used in the database text. You can find the full specification on the Web at http://www.omg.org/spec/UML/ with some introductory descriptions at http://www.ibm.com/developerworks/rational/library/769.html.

Instructional Support

- **The online Database Design expert system**. It has been statistically proven to help students learn database design. It provides immediate feedback to students. It saves hours of instructor grading time. It contains over 100 design problems that can be used for teaching or testing.
- **A test bank** with multiple choice and short answer questions.
- **Lecture notes** and overheads are available as slide shows in Microsoft PowerPoint format. The slides contain all the figures and additional notes. The slides are organized into lectures and can be rearranged to suit individual preferences.
- **The sample databases** and solutions can be downloaded from the online site. The instructor can add new data, modify the exercises, or use them to expand the discussion in the text. The databases are provided for several DBMSs.
- **The Instructor's Manual** contains answers to the exercises.

The Online System

All of the instructional material is available online. The main reason for this move is cost. The main textbook and the workbooks were rewritten and expanded. The costs and student prices for print books are out of line. Electronically, it is possible to make the entire set available for an almost trivial fee. It is impossible to illustrate multiple DBMSs within a single textbook. Splitting the workbooks from the main text makes it possible to cover a variety of DBMSs—without confusing the reader. The DBMS market is becoming increasingly fragmented, and students need (1) a relatively agnostic main textbook to describe the common features, and (2) a workbook that provides the hands-on steps to actually build a database application. The online e-book method provides the additional benefit of showing the students how to accomplish the same tasks with multiple DBMSs. Even if students begin by learning one DBMS, they can download a second or third workbook and transfer their knowledge to a new DBMS.

E-books provide additional benefits, including advanced search capabilities. Students can also set bookmarks and highlight sections. More importantly, they get to keep the books, instead of being forced to sell them back at the end of the term. Database application development is an important topic, and the examples, comments, and tips in the books will be valuable to students throughout their careers.

Major Changes with the Sixth Edition

The overall goals and structure remain the same with the sixth edition. Several sections were rewritten to improve clarity and incorporate some newer concepts. But the biggest change in each chapter was the addition and rewriting of most of the exercises.

Every chapter has a new Two-Minute Chapter section. This section summarizes the most important topics and overall goal of the chapter. It is useful for review to ensure students understand the key topics.

The new text also has a stronger bias towards Web-based applications. Almost any new application built today is either Web-based or possibly phone-based. Web (and mobile) applications heavily use centralized databases. Database design and queries remain similar, but interface, usability, and management issues are different from traditional in-house applications. As part of this emphasis, a new chapter (13) has been added to discuss features and limitations of the new crop of non-relational (sometimes called noSQL) database systems. These tools are designed for specific tasks: writing and retrieving specific data for millions of users. The open-source project Cassandra is used as an example to demonstrate the design and query tradeoffs in a highly-parallel environment.

Brief Contents

Contents

Chapter 1

Introduction

Chapter Outline

What You Will Learn in This Chapter

- What is a database?
- What do database applications look like?
- How are databases used to build applications?
- What are the major components of a database management system?
- What are the advantages of using a database management system?
- What are the main database management systems?
- How have database management systems changed over time?
- What potential problems exist with a DBMS approach?
- What is an application?
- What databases are used with this book?
- What are the first steps to start a project?

A Developer's View

Miranda: My uncle just called me and said his company was desperate. It needs someone to build an application for the sales team. The company wants a laptop system for each salesperson to enter orders. The system needs to track the order status over time and generate notices and weekly reports. My uncle said that because I know a lot about computers, I should call and get the job. His company is willing to pay $6,000, and I can work part-time.

Ariel: Wow! Sounds like a great job. What's the problem?

Miranda: Well, I know how to use basic computer tools, and I can program a little, but I'm not sure I can build a complete application. It could take a long time.

Ariel: Why not use a database management system? It should be easier than writing code from scratch.

Miranda: Do you really think so? What can a database system do? How does it work?

Getting Started

You need to choose a DBMS to use for exercises and projects. The Workbooks support Microsoft Access, Microsoft SQL Server, and Oracle. You can also use any SQL-based DBMS for most of the chapters. If necessary, install the DBMS on your computer—this step can take a few hours if you need to download the software. This chapter describes the basic features of a DBMS, but you will learn the details in the later chapters. You should read the descriptions of the sample databases, install them, and check out some of the data.

Introduction

What is a database? Do you want to build computerized business applications? Do you want to create business applications that operate in multiple locations? Do you want to conduct business on the Internet? Do you want to enable customers to place orders using the Web? If you are going to build a modern business application, you need a database management system.

Think about applications that you use: Almost any Web application, student course registration or billing systems, calendars, even games. All of them need to store data. If you are creating an application you have to identify the data to be stored and the best location to store it. You could use file read/write tools to save the data in a proprietary format that could only be read by your custom application. Or, you could store the data using a database management system. **database management system (DBMS)** is a software tool created to solve the common problems of sharing data among multiple users and applications. It has many features that make it easy to store and retrieve data efficiently.

The alternative to using a DBMS is to write file storage routines for every application that you create. For each application, you could store data in separate files, but only your application would know how to retrieve that data. Imagine how difficult it would be for you to write a new application that uses data created by a program that someone else wrote ten years ago. Also, for every file-based application, you would have to rewrite the data-handling routines to deal with multiple users accessing the data at the same time (concurrent access). You would also have to provide security and data management routines. It is far easier and more reliable to use a database management system. It already has these features and more, so you can concentrate on building an application that meets the needs of the users.

Business applications often utilize common types of data—information about customers, employees, products, sales, purchases, and so on. A database system is one of the most powerful tools you can use to build business applications because it easily handles common business data, supports security controls, and sharing. Most systems also have query systems, powerful report writers, and application systems that make it easy to quickly build applications and retrieve data to support common business needs.

The most important features of a DBMS are the ability to define a database, store the data efficiently, and retrieve data with a query language. A **database** is a collection of data stored in a standardized format designed to be shared by multiple users. These concepts and the tools are discussed in detail in this book. The key point to remember now is that a database is independent of any specific application. Once you create a database within a DBMS, the data can be accessed with a variety of tools. The DBMS provides many tools to manage the data including security controls, data storage options, and backup facilities.

This chapter describes the basic role of a DBMS in application development. It also describes the major features of a DBMS and how you will use them in building business applications. It also summarizes the evolution of DBMS technology so you understand some of the background and can think about possible changes for future systems. This book focuses on building applications that use databases. It avoids detailed discussions of how database management systems are written.

Two-Minute Chapter

Chapter 1 is an introduction to databases and what they provide to developers of business applications. The main purpose is to explain their importance and outline how this book can be used to learn how to build and create applications using database systems. Database management systems are powerful tools that are used in almost all business applications. They solve many common problems for storing and accessing data and maintaining the integrity of the data with multiple users making changes. But, databases must be carefully designed to obtain these benefits. Relational databases are the most common tools in business and a set of normalization rules are used to identify exactly which columns belong in a table and when data need to be split into multiple tables with additional key columns. Chapters 2 and 3 focus on the rules for designing tables. One of the main strengths of a DBMS is the separation of the data from the application. By concentrating on the data storage, the data remains protected and accessible to almost any application. Tools and applications can change, and the data remains useful and accessible.

Newer key-value pair highly-distributed databases, such as Cassandra, are designed for massive Web sites with millions of users, where performance of a few specific queries takes priority over everything else. Database design is also critical for these tools, but it is not as rigid—which means that experience and experimentation are needed to determine how to optimize the storage and retrieval for each specific problem. Chapter 13 explores the specific details of key-value pair databases, but it relies on basic understanding of relational design and queries—at least chapters 2 and 4.

Most DBMSs include the data storage engine and a query processor. SQL is a standard language used by most relational systems. Basic queries are straightforward, but SQL has powerful features to help answer complex business questions. Chapters 4 and 5 focus on constructing SQL queries to answer common business questions.

Ultimately, databases are not built in isolation are part of an application. The database is likely to be invisible to most users. Instead, users interact with the application through forms and reports. These tools can be created with desktop tools or as Web-based forms. The capabilities and tools for building forms and reports present the greatest differences between database tools. Eventually, developers need to learn to use at least one set of tools in depth. The basic skills are transferrable to other tools, but a lack of standards requires learning picky details for each system. Chapters 6, 7, and 8 explain the process in general; but individual details for specific tools are covered in the accompanying workbooks.

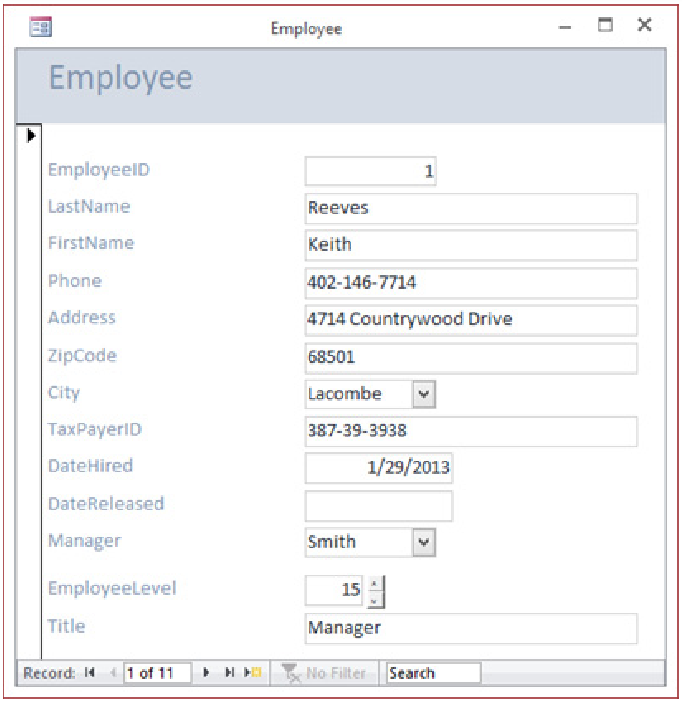
Increasingly, managers are looking to expand their use and understanding of a huge amount of data being collected. Much of this analysis requires statistical knowledge and tools that require additional background. However, the data storage needs are slightly different, so data warehouses and some basic analytical tools are covered in Chapter 9.

Chapter 10 examines some of the issues involved in setting up and maintaining a database. Security is important in all business applications today and DBMSs provide tools for assigning and monitoring various security conditions. Chapter 11 looks at the basic issues involved in distributed systems—where data is stored across multiple servers to improve performance and reliability. Chapter 12 is a bridge to computer science. DBMSs use specific methods to store and retrieve data. These topics are critical in computer science classes, and the chapter briefly shows how they are used to build the DBMS software; which highlights some of the strengths and weaknesses of the software.

A Small Sample Database Application

What do database applications look like? You have probably worked with several database-oriented applications and were not aware of the role of the database or the DBMS. In a business application, users do not care about the underlying data storage mechanism—they are only interested in the final application features and usability. But a DBMS can provide amazing functionality with less effort than programming for many applications. Before trying to explain the functions and benefits of a DBMS it is useful to look at some of the basic application features. This book uses several sample databases to illustrate the various capabilities and features. You should download them from the Web site and look through the applications. Note that the versions in Microsoft Access have more elements (forms and reports) than those in the server-based systems. These versions are also easier to examine—but you need a copy of Microsoft Access software.

A couple of examples from the Pet Store database are useful to understand how a database can be used to create business applications. The Pet Store database is a partially-completed application. It has all of the tables and a small amount of sample data. But only a couple of forms and reports were created. The Bicycle application is much larger and contains several useful and more complex forms.



Employee

EmployeeID	1
LastName	Reeves
FirstName	Keith
Phone	402-146-7714
Address	4714 Countrywood Drive
ZipCode	68501
City	Lacombe
TaxPayerID	387-39-3938
DateHired	1/29/2013
DateReleased	
Manager	Smith
EmployeeLevel	15
Title	Manager

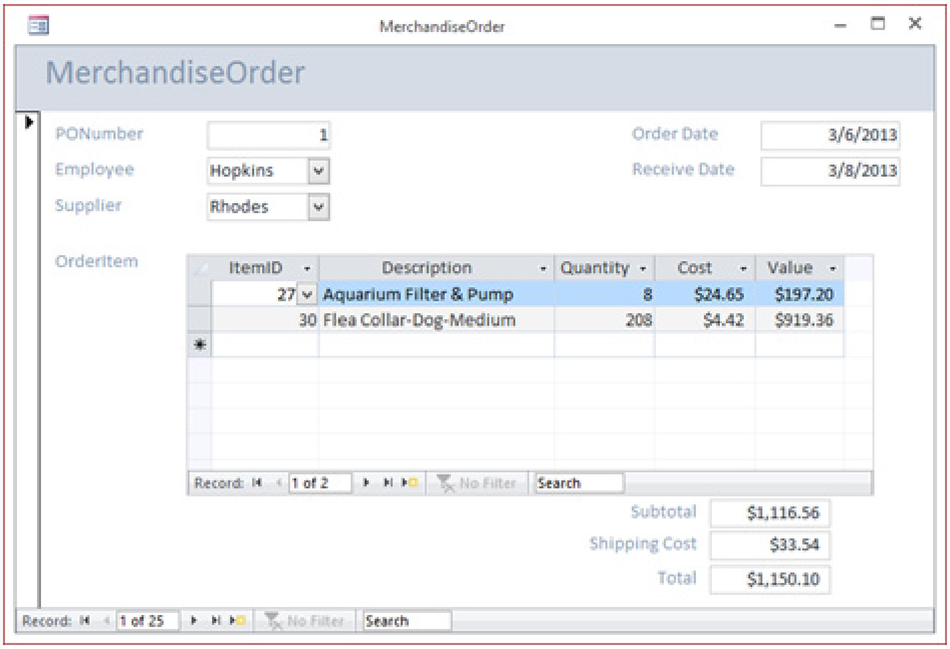
Figure 1.1

Sample Employee form. Users see a form with controls to help them enter and edit data. The data items are stored in the database but the form could be located on a single computer, a Web site, or even a mobile application.

The Bicycle case is good for examining detailed features of a mostly finished application. But it is easier to see the process of creating an application by looking at the partially finished Pet Store or CornerMed examples. Figure 1.1 shows a simple Employee form which is used to add data for new employees or edit values for existing workers. The form focuses attention on a single Employee at a time. It contains controls to make some tasks easier including selecting cities (or managers) from a list.

The Employee form is relatively simple because it shows only one concept: information about a person. Business applications are often more complex. Figure 1.2 shows a basic purchase order form, where the Pet Store is buying bulk items from a supplier. This form needs to display information about the order itself: Dates, the supplier, and total cost. It also needs to collect data on the individual items being purchased such as the cost and the quantity. The form is also capable of computing totals automatically. In this example, the repeating Value column, Subtotal, and Total values are computed based on arithmetic formulas that are programmed as properties in the form. Many additional features can be added such as filters to show all orders from a single supplier, orders within a range of dates, or orders that have not yet been received.

Applications usually have reports that can include tables, subtotals, and charts. Increasingly, these tools are interactive, where users can click buttons to compare totals across various categories or quickly create charts to see how values



MerchandiseOrder

PONumber	1	Order Date	3/6/2013
Employee	Hopkins	Receive Date	3/8/2013
Supplier	Rhodes		

OrderItem

ItemID	Description	Quantity	Cost	Value
27	Aquarium Filter & Pump	8	$24.65	$197.20
30	Flea Collar-Dog-Medium	208	$4.42	$919.36

Record: 1 of 2

Subtotal	$1,116.56
Shipping Cost	$33.54
Total	$1,150.10

Record: 1 of 25

Figure 1.2

Sample Purchase Order form. The order form is more complex and handles data entry for the order itself as well as the individual items being purchased in the detail/ repeating section.

change over time. These capabilities are often built into the DBMS tools and can sometimes be created in a few minutes by a skilled developer. In other cases, a programmer needs to write custom applications that provide tools to the users and interact with the database to store and retrieve the desired data.

The key thing to remember is that all of the data is handled by the DBMS. Even if the forms and reports are created with traditional programming tools, the DBMS stores and controls the data centrally. And the DBMS handles security controls, simultaneous access by multiple users, data backup, and data integrity issues—such as preventing negative values for prices.

Databases and Application Development

How are databases used to build applications? It is rare that someone would ask you to build a database and just use it to store and retrieve data. In almost all situations, someone asks you to build an application. The difference is that an application is used to perform specific tasks. In the process, you will use the DBMS to store and retrieve the data, but ultimately, you are building the database as part of the solution to the user's problem.

As shown in Figure 1.3, the database itself is just one element of the application. Developers define tables to hold the data in the DBMS. Application forms are screens displayed to the user to collect or display data. The data is stored in the underlying tables. Reports are structured displays of data from the tables—typically containing subtotals and charts. In new applications, these forms and reports are accessible using a Web browser. Applications built with older systems might require database components installed on each computer.

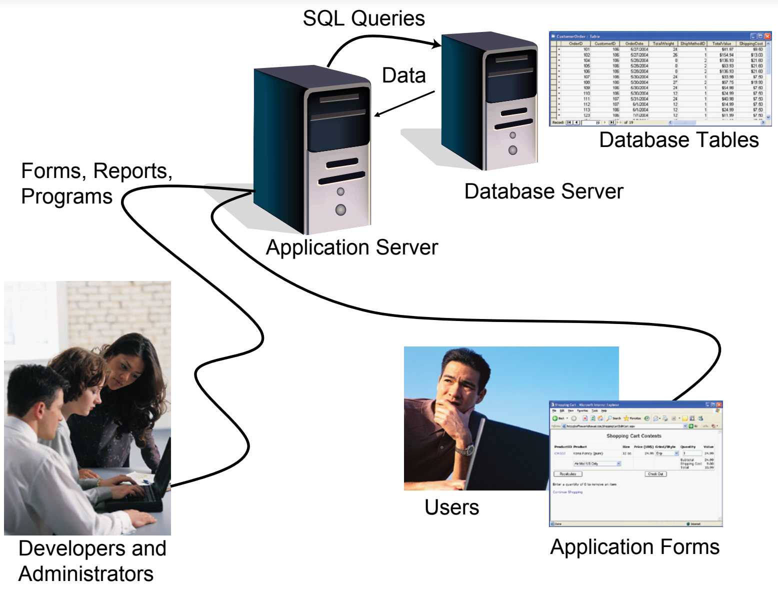


Figure 1.3

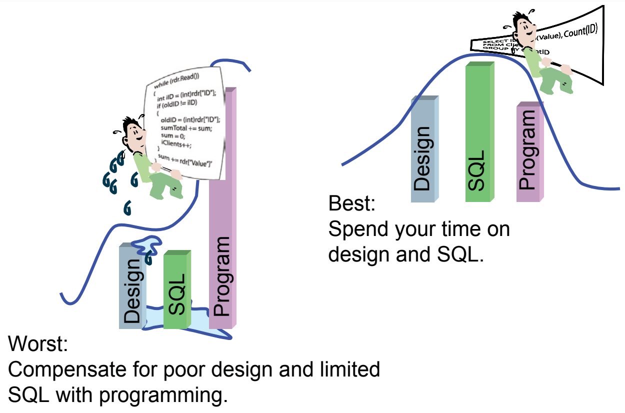
Application development with a DBMS. Developers and administrators define the database in the form of tables. They then create forms and reports on the application server. Users run the application and enter data or make choices.

Several different DBMS tools exist today and one of the first decisions you must make is to select the DBMS for the application being built. In many situations, this choice has already been made for you. For example, your instructor has probably already selected a DBMS, or within a large company, one tool is used as the standard platform. The Workbooks that accompany this textbook provide details about several specific DBMS platforms, including Microsoft Access, Microsoft SQL Server, and Oracle. Each of them uses different tools to create forms and reports, but the Workbooks provide examples and steps for using those tools.

Storing and retrieving data is relatively standard today—most of the DBMSs use the SQL query language. Designing the database tables and retrieving the data are critical tasks that are essentially the same regardless of the DBMS. Chapters 2 and 3 of this book focus on database design. Chapters 4 and 5 explore the power of SQL to retrieve and manipulate data. These chapters and tasks apply to almost any DBMS.

Building applications sometimes requires writing programming code. Some tools require more programming than others. Some tools have their own internal programming language, while others rely on standard languages (such as C++ or Java), and embed the database elements as extensions to the language. In any case, database applications will be easier to understand if you have already had at least one programming course.

The development process for a DBMS is somewhat different from traditional programming. One of the key changes is that your primary focus is on the data and how it is organized. Later, you can build forms and reports. To gain the advantages, data must be carefully organized. The query language is also a powerful component of a DBMS. It makes it easy to retrieve data—usually with a few


Design
SQL
Program
Worst:
Compensate for poor design and limited SQL with programming.
Design
SQL
Program
Best:
Spend your time on design and SQL.

```
while (rdr.Read())
{
  int iID = (int)rdr["ID"];
  if (oldID != iID)
  {
    oldID = (int)rdr["ID"];
    sumTotal += sum;
    sum = 0;
    iClients++;
  }
  sum += rdr["Value"]'
}
```

```
SELECT ID (Value), Count(ID)
FROM Clie
GROUP BY  tID
```

Figure 1.4

Creating business applications. A DBMS can save you hundreds of hours of work in building applications. However, you must design your database correctly and use SQL to do the heavy work.

lines of simple commands. Once you understand the concepts of database design, queries, and application building, you will be able to create complex applications in a fraction of the time it would take with traditional programming techniques. Figure 1.4 illustrates the tradeoffs that you face in building applications. It is critical that you spend time and design your database correctly. You also need to use the query language (SQL) to do the heavy work in retrieving data. With these two tools, your application programming becomes easy and you can spend most of your time building forms and reports with automated tools. It still takes time and effort, but it is considerably easier than relying on detailed coding.

In the last few years, database systems have become the foundation of almost all application development projects. From large enterprise resource planning systems, to e-business Web sites, to standalone business applications, database systems store and retrieve data efficiently, provide security, and make it easier to build the applications. Today, when you build or modify an application, you will first create the database. To understand the capabilities of a DBMS and how you will use them to create applications, it is best to examine the process used to develop applications.

Organizations typically follow the basic steps outlined in Figure 1.5 when creating technology applications. Larger projects may require several people in each phase, whereas smaller projects might be created entirely by one or two developers. Organizations can rearrange the tasks that fall within each step, but all of the tasks must be completed for a project to be successful. The feasibility step defines the project and provides estimates of the costs. During the analysis phase, systems analysts collect data definitions, forms, and reports from users. These are used to design the database and all of the new forms, reports, and user interactions. During the development step, the forms, reports, and application features such as help files are created. Implementation generally consists of the transfer of data, installation, training, and review.

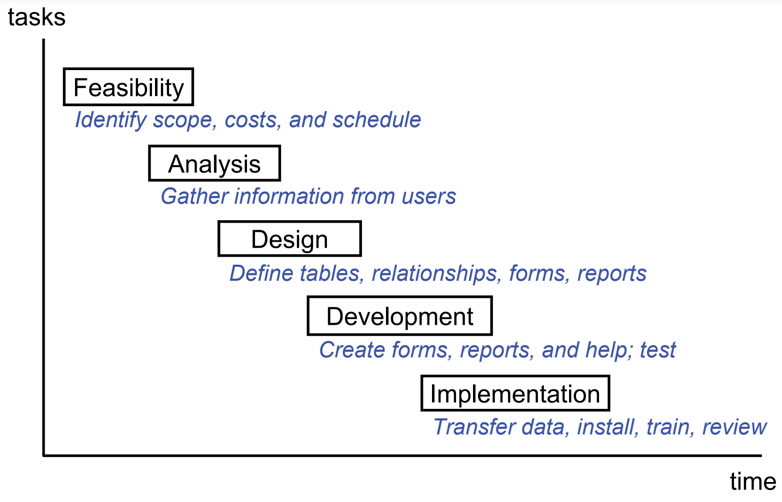

tasks
Feasibility
Identify scope, costs, and schedule
Analysis
Gather information from users
Design
Define tables, relationships, forms, reports
Development
Create forms, reports, and help; test
Implementation
Transfer data, install, train, review
time

Figure 1.5

Systems development. Particularly for large projects, it is useful to divide application development into separate steps. They can be used to track the progress of the development team and highlight the steps remaining. For some projects, it is possible to overlap or even iterate the tasks, but steps should not be skipped.

For database-driven applications, the design stage is critical. Database systems and the associated development tools are incredibly powerful, but databases must be carefully designed to take advantage of this power. Figure 1.6 shows that the business rules and processes are converted into database tables and relationship definitions. Forms are defined that transfer data into the database, and reports use queries to retrieve and display data needed by users. These forms and reports, along with features such as menus and help screens, constitute applications. Users generally see only the application and not the underlying database or tables.

Designing the database tables and relationships is a key step in creating a database application. The process and rules for defining tables are detailed in Chapters 2 and 3. Using the database requires the ability to retrieve and manipulate the data. These tasks are handled by the query system, which is described in Chapters 4 and 5. With these foundations, it is relatively easy to use the tools to create forms and reports and build them into applications as discussed in Chapters 6, 7, and 8.

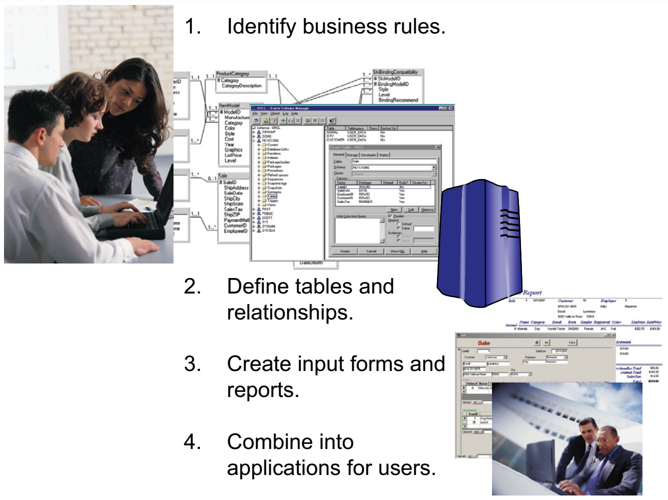

1. Identify business rules.
2. Define tables and relationships.
3. Create input forms and reports.
4. Combine into applications for users.

Figure 1.6

Steps in database design. The business rules and data are used to define database tables. Forms are used to enter new data. The database system retrieves data to answer queries and produce reports. Users see only the application in terms of forms and reports.

Components of a Database Management System

What are the major components of a database management system? To understand the value of a DBMS, it helps to see the components that are commonly provided. This basic feature list is also useful when you evaluate various products to determine which DBMS your company should use. Each DBMS has unique strengths and weaknesses. You can evaluate the various products according to how well they perform in each of these categories. The primary categories are the database engine, query processor, report service, forms development, management tools, and security.

Database Engine

The **database engine** is the heart of the DBMS. It is responsible for storing, retrieving, and updating the data. This component is the one that most affects the performance (speed) and the ability to handle large problems (scalability). The other components rely on the engine to store not only the application data but also the internal system data that defines how the application will operate. Figure 1.7 illustrates the primary relationship between the database engine and the data tables.

With some systems the database engine is a stand-alone component that can be purchased and used as an independent software module. For example, the Microsoft "jet engine" forms the foundation of Access. Similarly, the database engines for Oracle and Microsoft SQL Server can be purchased separately.

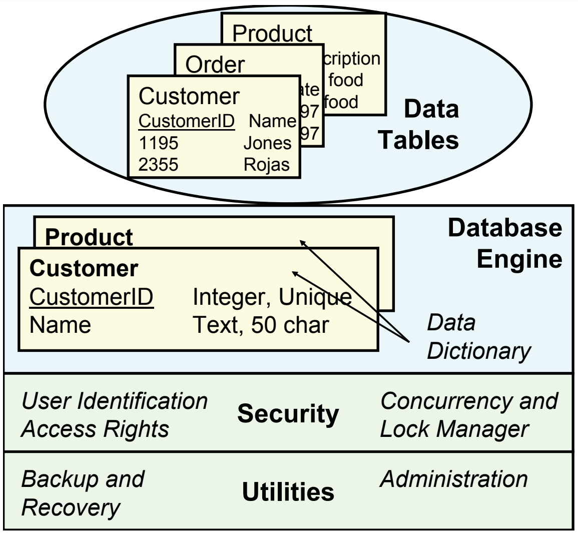


Figure 1.7

Database engine. The engine is responsible for defining, storing, and retrieving the data. The security subsystem of the engine identifies users and controls access to data.

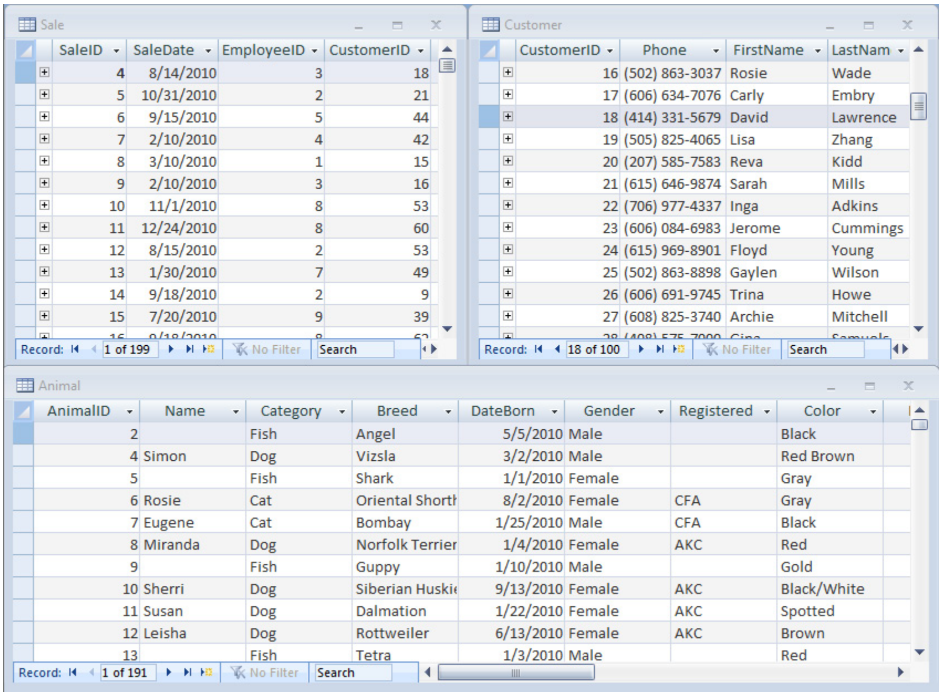
The database engine is also responsible for enforcing business rules regarding the data. For example, most businesses would not allow negative prices to be used in the database. Once the designer creates that rule, the database engine will warn the users and prevent them from entering a negative value.

As shown in Figure 1.8, the database engine stores data in carefully designed tables. Tables are given names that reflect the data they hold. Columns represent simple attributes that describe the object, such as an employee's name, phone, and address. Each row represents one object in the table.

Database performance is an important issue. The speed of your application depends on the hardware, the DBMS software, the design of your database, and on how you choose to store your data. Chapter 12 discusses some popular methods, such as indexing, that improve the performance of a database application. Performance is also affected by how the software is written. Tools such as Microsoft Access have limitations on the size of the database and on how the data is processed. Similarly, free tools, including versions for Microsoft SQL Server, Oracle, and IBM, have limits on size and processing (such as support for only one processor). More expensive versions and other software tools remove these limitations.

Data Dictionary

The **data dictionary** holds the definitions of all of the data tables. It describes the type of data that is being stored, allows the DBMS to keep track of the data, and helps developers and users find the data they need. Most modern database systems hold the data dictionary as a set of system tables. For example, Microsoft Access keeps a list of all the tables in a hidden system table called MsysObjects. The larger systems like SQL Server and Oracle also have proprietary tables such as sys.dba_tables in Oracle. However, most of the vendors (except Oracle) have



Sale

SaleID	SaleDate	EmployeeID	CustomerID
4	8/14/2010	3	18
5	10/31/2010	2	21
6	9/15/2010	5	44
7	2/10/2010	4	42
8	3/10/2010	1	15
9	2/10/2010	3	16
10	11/1/2010	8	53
11	12/24/2010	8	60
12	8/15/2010	2	53
13	1/30/2010	7	49
14	9/18/2010	2	9
15	7/20/2010	9	39

Customer

CustomerID	Phone	FirstName	LastNam
16	(502) 863-3037	Rosie	Wade
17	(606) 634-7076	Carly	Embry
18	(414) 331-5679	David	Lawrence
19	(505) 825-4065	Lisa	Zhang
20	(207) 585-7583	Reva	Kidd
21	(615) 646-9874	Sarah	Mills
22	(706) 977-4337	Inga	Adkins
23	(606) 084-6983	Jerome	Cummings
24	(615) 969-8901	Floyd	Young
25	(502) 863-8898	Gaylen	Wilson
26	(606) 691-9745	Trina	Howe
27	(608) 825-3740	Archie	Mitchell

Animal

AnimalID	Name	Category	Breed	DateBorn	Gender	Registered	Color
2		Fish	Angel	5/5/2010	Male		Black
4	Simon	Dog	Vizsla	3/2/2010	Male		Red Brown
5		Fish	Shark	1/1/2010	Female		Gray
6	Rosie	Cat	Oriental Shortl	8/2/2010	Female	CFA	Gray
7	Eugene	Cat	Bombay	1/25/2010	Male	CFA	Black
8	Miranda	Dog	Norfolk Terrier	1/4/2010	Female	AKC	Red
9		Fish	Guppy	1/10/2010	Male		Gold
10	Sherri	Dog	Siberian Huskie	9/13/2010	Female	AKC	Black/White
11	Susan	Dog	Dalmation	1/22/2010	Female	AKC	Spotted
12	Leisha	Dog	Rottweiler	6/13/2010	Female	AKC	Brown
13		Fish	Tetra	1/3/2010	Male		Red

Figure 1.8

Database tables in Access. Tables hold data about one business entity. For example, each row in the Animal table holds data about a specific animal.

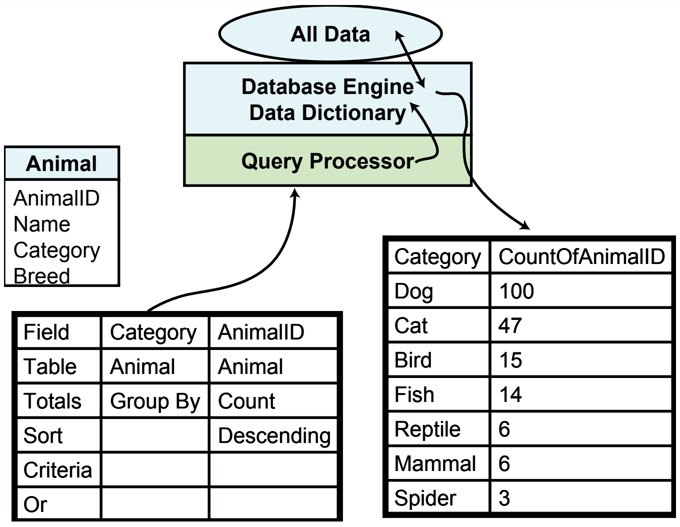
standardized on the Information_Schema queries, such as the Information_Schema.Tables view. These tools and related administrative issues are described in Chapter 10. If you need to install your own copy of the DBMS software now, you should read the basic steps in Chapter 1 of the associated Workbook. You can also skim through Chapter 10 if you want more detailed explanations.

These meta-data tables are used by the system, but most database systems also provide visually-oriented administration tools so you do not have to memorize commands. For example, it is relatively easy to obtain a list of tables using the basic administration tools for Access, SQL Server, Oracle, and DB2. For independent tools like MySQL, you will probably have to track down and install a separate management utility.

Query Processor

The query processor is a fundamental component of the DBMS. It enables developers and users to store and retrieve data. In some cases the query processor is the only connection you will have with the database. That is, all database operations can be run through the query language. Chapters 4 and 5 describe the features and power of query languages—particularly standard SQL.

Queries are derived from business questions. The query language is necessary because natural languages like English are too vague to trust with a query. To minimize communication problems and to make sure that the DBMS understands



Field	Category	AnimalID
Table	Animal	Animal
Totals	Group By	Count
Sort		Descending
Criteria		
Or		

Category	CountOfAnimalID
Dog	100
Cat	47
Bird	15
Fish	14
Reptile	6
Mammal	6
Spider	3

Figure 1.9

Database query processor. The data dictionary determines which tables and columns should be used. When the query is run, the query processor communicates with the database engine to retrieve the requested data.

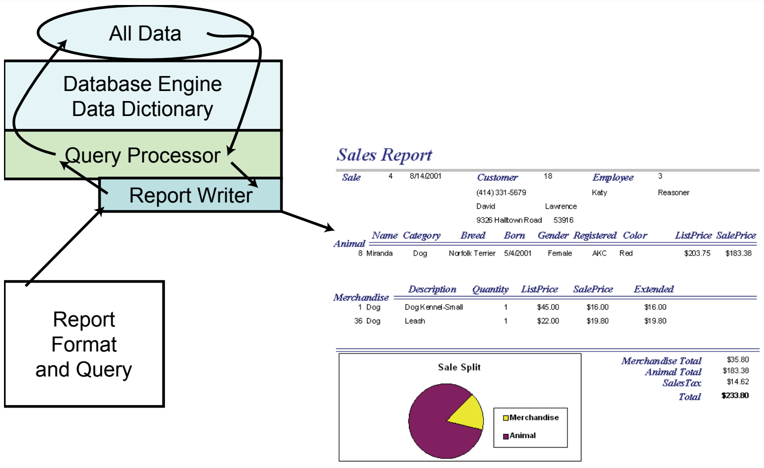
your question, you should use a query language that is more precise than English. As shown in Figure 1.9, the DBMS refers to the data dictionary to create a query. When the query runs, the DBMS query processor works with the database engine to find the appropriate data. The results are then formatted and displayed on the screen.

Report Service

Most business users want to see summaries of the data in some type of report. Many of the reports follow common formats. A **report writer** enables you to set up the report on the screen to specify how items will be displayed or calculated. Most of these tasks are performed by dragging data onto the screen. Professional-level writers enable you to produce complex reports in a short time without writing any program code. Chapter 8 describes several of the common business reports and how they can be created with a database report writer. Increasingly, vendors are shipping **report services** that run as Web applications to make it easy to deliver your reports to users without relying on paper. Users can choose to explore the data interactively, or print the report on their own printer.

The process of exploring data interactively is increasingly important. The system demands for this type of application are quite different from traditional transactions and reporting systems. Consequently, most companies rely on separate report services and **online analytical processing (OLAP)**, and statistical **data mining** tools. Database designs and application tools are relatively new and substantially different from traditional database applications, so they are examined separately in Chapter 9.

The report writer can be integrated into the DBMS, or it can be a stand-alone application that the developer uses to generate code to create the needed report. As



Sales Report

Sale	4	8/14/2001	*Customer*	18	*Employee*	3
			(414) 331-5679		Katy	Reasoner
			David	Lawrence		
			9326 Halltown Road	53916		

Animal	*Name*	*Category*	*Breed*	*Born*	*Gender*	*Registered*	*Color*	*ListPrice*	*SalePrice*
8	Miranda	Dog	Norfolk Terrier	5/4/2001	Female	AKC	Red	$203.75	$183.38

Merchandise		*Description*	*Quantity*	*ListPrice*	*SalePrice*	*Extended*
1	Dog	Dog Kennel-Small	1	$45.00	$16.00	$16.00
36	Dog	Leash	1	$22.00	$19.80	$19.80

Merchandise Total	$35.80
Animal Total	$183.38
SalesTax	$14.62
Total	**$233.80**

Figure 1.10

Database report writer. The design template sets the content and layout of the report. The report writer uses the query processor to obtain the desired data. Then it formats and prints the report.

shown in Figure 1.10, the developer creates a basic report design. This design is generally based on a query. When the report is executed, the report writer passes the query to the query processor, which communicates with the database engine to retrieve the desired rows of data. The report writer then formats the data according to the report template and creates the report complete with page numbers, headings and footers.

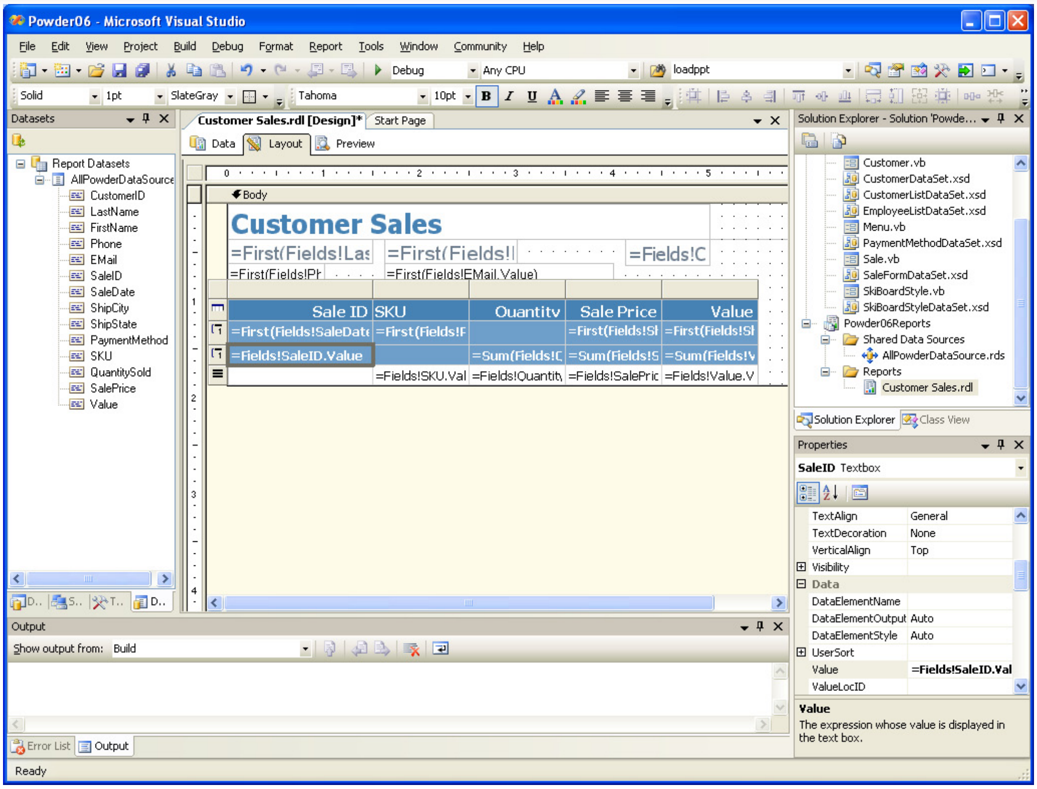
Figure 1.11 shows the report writer that Microsoft SQL Server provides with its Business Intelligence Reporting Services tool. The report writer generates reports that are posted to a Web site to be run by other users. You set up sections on the report and display data from the database. The report writer includes features to perform computations and format the columns. You also have control over colors, you can place images on the report (e.g., logos), and you can draw lines and other shapes to make the report more attractive or to call attention to specific sections.

Forms Development

A **forms builder** or input screen editor helps the developer create input forms. As described in Chapter 7, the goal is to create forms that represent common user tasks, making it easy for users to enter data. The forms can include graphs and images. The forms builder enables developers to create forms by dragging and dropping items on the screen. Figure 1.12 shows that forms make heavy use of the query processor to display data on the form.

Many database systems also provide support for traditional, third-generation languages (3GL) to access the database. The issues in writing programs and accessing data through these programs are directly related to the topics discussed in Chapter 7.

One of the most important questions you need to address for new projects is whether the application needs to be built as a Web site. Today, most new develop-



Customer Sales

=First(Fields!Las =First(Fields!! =Fields!C
=First(Fields!Pr =First(Fields!EMail.Value)

Sale ID	SKU	Quantity	Sale Price	Value
=First(Fields!SaleDate	=First(Fields!F		=First(Fields!SI	=First(Fields!SI
=Fields!SaleID.Value		=Sum(Fields!C	=Sum(Fields!S	=Sum(Fields!V
	=Fields!SKU.Val	=Fields!Quantity	=Fields!SalePric	=Fields!Value.V

Show output from: Build

Properties

SaleID Textbox

TextAlign	General
TextDecoration	None
VerticalAlign	Top
Visibility	
Data	
DataElementName	
DataElementOutput	Auto
DataElementStyle	Auto
UserSort	
Value	=Fields!SaleID.Val
ValueLocID	

Value
The expression whose value is displayed in the text box.

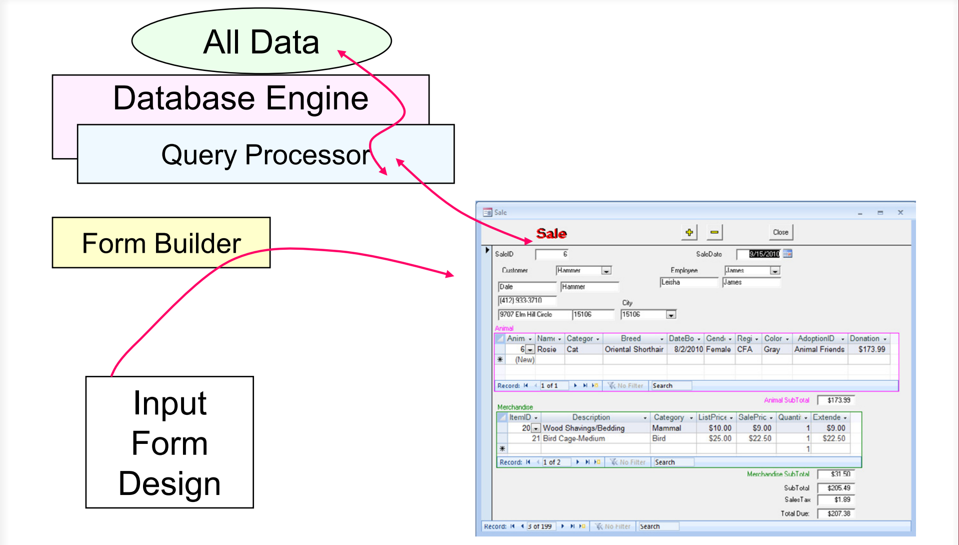
Figure 1.11

Oracle Reports report writer. The Data Model is used to create a query and select the data to be displayed. Then Reports creates the basic report layout. You can modify the layout and add features to improve the design or highlight certain sections.

ments are based on Web pages. However, the tools for building Web sites varies greatly depending on the underlying platform. For example, Microsoft uses its ASP.Net server to create and deliver pages, Java platforms (including Oracle) use the Java language and Java application servers, other tools use the open-source Apache server and often the PHP or Python programming languages to develop Web forms. The overall approach to building forms and reports is similar in these tools, but the methods and details are quite different.

Management Utilities and Security

Because data is so important to organizations, the DBMS includes several mechanisms and tools to protect the data and assign security permissions. As a result, someone needs to be in charge of assigning security, monitoring the database, and performing other management chores. A DBMS typically provides command-line tools as well as visual tools to help you perform these jobs. Chapter 10 describes the various tasks and introduces some of the commonly available tools. Typical features include backup and recovery, user management, data storage evaluation, and performance-monitoring tools.



Sale

SaleID	6	SaleDate	9/15/2010
Customer	Hammer	Employee	James
Dale	Hammer	Leisha	James
(412) 933-3710		City	
9707 Elm Hill Circle	15106	15106	

Animal

Anim	Name	Categor	Breed	DateBo	Gend	Regi	Color	AdoptionID	Donation
6	Rosie	Cat	Oriental Shorthair	8/2/2010	Female	CFA	Gray	Animal Friends	$173.99
(New)									

Record: 1 of 1

Animal SubTotal $173.99

Merchandise

ItemID	Description	Category	ListPrice	SalePric	Quanti	Extende
20	Wood Shavings/Bedding	Mammal	$10.00	$9.00	1	$9.00
21	Bird Cage-Medium	Bird	$25.00	$22.50	1	$22.50
					1	

Record: 1 of 2

Merchandise SubTotal	$31.50
SubTotal	$205.49
SalesTax	$1.89
Total Due:	$207.38

Record: 3 of 199

Figure 1.12

Database form. A form is used to collect data. It is designed to match the tasks of the user, making it easy to enter data and look up information. The query processor is used to obtain related data and fill in look-up data in combo boxes.

For security to work, it has to be embedded into the database engine. Consequently, you will encounter some security questions before you reach Chapter 10. In most cases, you will have a separate security account that has the permissions needed to complete most of the exercises in the book. However, if you need to share a database with dozens of other students, you might be denied the ability to perform some tasks, such as deleting data or tables. The challenge is even greater in Chapter 10. If you are serious about learning database administration tasks, you will need to install your own copy of the DBMS so that you have complete access and the ability to alter detailed elements.

Advantages of the Database Management System Approach

What are the advantages of using a database management system? Many business applications need the same features (efficient storage and retrieval of data, sharing data with multiple users, security, and so on). Rather than re-create these features within every application program, it makes more sense to purchase a database management system that includes these basic facilities. Then developers can focus on creating applications to solve business problems. The primary benefits provided by a DBMS are shown in Figure 1.13.

First, the DBMS stores data efficiently. As described in Chapters 2 and 3, if you set up your database according to a few basic rules, the data will be stored with minimal wasted space. Additionally, the data can be retrieved rapidly to answer any query. Although these two goals seem obvious, they can be challenging to handle if you have to write programs from scratch every time.

The DBMS also has systems to maintain data consistency with minimal effort. Most systems enable you to create basic business rules when you define the data. For example, price should always be greater than zero. These rules are enforced

- Minimal data redundancy
- Data consistency
- Integration of data
- Sharing of data
- Enforcement of standards
- Ease of application development
- Uniform security, privacy, and integrity
- Data independence

Figure 1.13

Advantages of a DBMS. The DBMS provides a solution to basic data storage and retrieval problems. By using a DBMS to handle data storage problems, programmers can concentrate on building applications—saving time and money in developing new systems and simplifying maintenance of existing applications.

for every form, user, or program that accesses the data. With traditional programs, you would have to force everyone to follow the same rules. Additionally, these rules would be stored in hundreds or thousands of separate programs—making them hard to find and hard to modify if the business changes.

The DBMS, particularly the query language, makes it easy to integrate data. For example, one application might collect data on customer sales. Another application might collect data on customer returns. If programmers created separate programs and independent files to store this data, combining the data would be difficult. In contrast, with a DBMS any data in the database can be easily retrieved, combined, and compared using the query system.

Focus on Data

With the old programming-file method, developers focused on the process and the program. Developers started projects by asking these kinds of questions: How should the program be organized? What computations need to be made? The database approach instead focuses on the data. Developers now begin projects by asking: What data will be collected? This change is more than just a technicality. It alters the entire development process.

Think about the development process for a minute. Which component changes the most: programs (forms and reports) or the data? Yes, companies collect new data all the time, but the structure of the data is relatively constant. And when it does change, the reason is usually that you are adding new elements—such as cellular phone numbers. In particular, business data is intentionally kept similar to enable comparisons over time. Sales, Costs, Inventory, and so on are stable numbers that are always collected. On the other hand, users constantly need modifications to forms and reports.

As shown in Figure 1.14, the database approach concentrates on the data. The DBMS is responsible for defining, storing, and retrieving the data. All requests for data must go through the database engine. Hence the DBMS is responsible for efficient data storage and retrieval, concurrency, data security, and so on. Once the data structure is carefully defined, additional tools like the report writer, forms generator, and query language make it faster and easier to develop business applications.

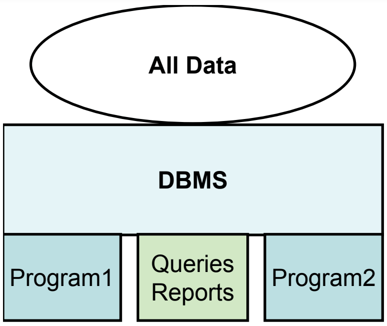


Figure 1.14

DBMS focus on data. First, define the data. Then all queries, reports, and programs access the data through the DBMS. The DBMS always handles common problems such as concurrency and security.

Data Independence

The other important feature of focusing on the data is the separation of the data definition from the program—known as **data independence**. Data independence enables you to change the data definition without altering the program. Similarly, data can be moved to new hardware or a completely different machine. Once the DBMS knows how to access the data, you do not have to alter the forms, reports, or programs that use that data. Similarly, you can alter individual programs without having to change the data definitions.

There are exceptions to this idealistic portrayal. Obviously, if you delete entire chunks of the database structure, some of your applications are not going to work properly. Similarly, if you make radical changes to the data definitions—such as changing phone number data storage from a numeric to a text data type—you will probably have to alter your reports and forms. However, a properly designed database rarely needs these radical changes.

Consider the problem of adding cell phone numbers to an Employee table. Figure 1.15 shows part of the data definition for employees. Regardless of how many forms, reports, or programs exist, the procedure is the same. Simply go to the

Figure 1.15

Adding cellular phone numbers to the Employee table. Adding a new element to a table does not affect the existing queries, reports, forms or programs.

Field Name	Data Type	Description
EmployeeID	Number	Generated
TaxpayerID	Text	Federal ID
LastName	Text	
FirstName	Text	
…		
Phone	Text	
…		
CellPhone	Text	Cellular number

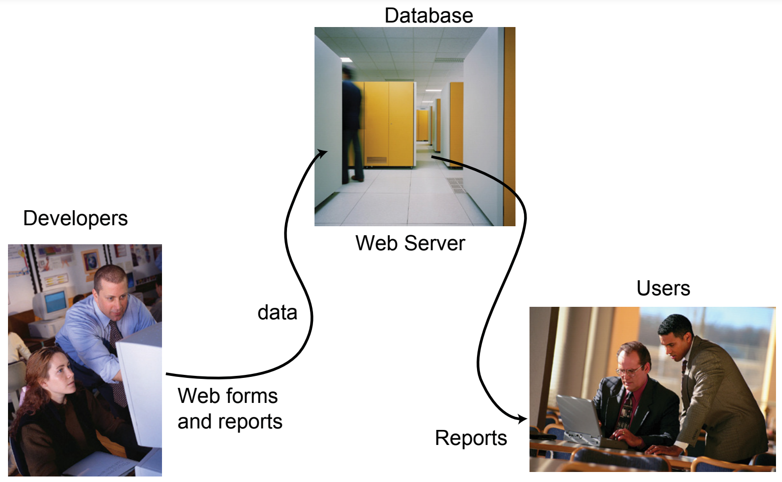


Figure 1.16

Web databases. Developers build forms and reports that are stored on a central server. Users access the data and display reports using a standard Web browser.

table definition and insert the entry for CellPhone. The existing queries, forms, reports, and programs will function exactly as they did before. Of course, they will ignore the new phone number entry. If you want to see the new values on a report, you will have to insert the new field onto the report. With a modern report writer, this change can be as simple as dragging the CellPhone item to the appropriate location on the form or report.

The focus on data and careful design enable database systems to avoid the problems experienced with traditional programming-file methods. The consolidation of common database functions within one application enables experts to create powerful database management systems and frees application programmers to focus on building applications that solve business problems.

Data Independence and Web Applications

For several years, business applications were built on a client-server model, where personal computers ran heavy applications such as DBMS and spreadsheet software. Data was shared with other users by placing it in a central database server and the individual applications connected across a network to retrieve and update the data. With the expanding use of the Web, the approach to business applications is changing. It is increasingly popular to build applications to run completely on centralized servers. With the Web approach, users only need access to a Web browser that can connect to the application server, which stores data in a database server. In many cases, the Web browser could run on simple inexpensive hardware, including cell phones. Although it raises new security issues, the Web approach also means that users can get access to the business data from almost any location.

More importantly, the Web approach makes it easier to modify the application. As shown in Figure 1.16, the data and the application forms and reports are all stored in a central location. It is easy to create new versions and change the software. In many cases, it is even possible to change the entire DBMS and the un-

Vendor	Product
Oracle	Oracle
Microsoft	SQL Server Access
IBM	DB2 Informix
Open source	PostgreSQL
MySQL (Oracle)	MySQL

Figure 1.17

Commercial DBMS vendors. These are the leading DBMS products that you are likely to encounter. Many older systems exist, and dozens of smaller vendors provide complete systems and other tools.

derlying hardware. Instead of sending patches and new versions to hundreds or thousands of users, the developers simply update the single copy sitting on the application server.

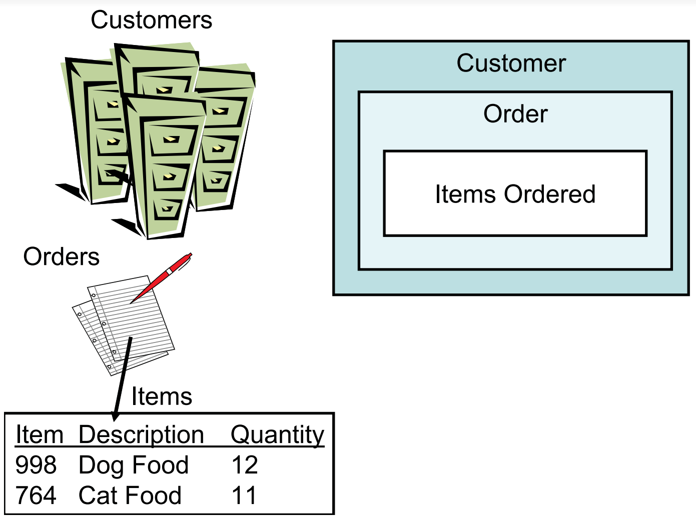
Developers can create new applications without altering the database. Similarly, they can expand the database or even move it to multiple servers, and the applications remain the same. Users continue to work with their familiar personal computer applications. Developers retain control over the data. The DBMS can monitor and enforce security and integrity conditions to protect the data, yet still give access to authorized users. Chapter 11 discusses the use of distributed database systems in more detail, including building client/server systems on the World Wide Web.

Leading Commercial Database Systems

What are the main database management systems? Figure 1.17 lists some of the leading database systems, including Oracle, DB2, and SQL Server. Many of the systems are available for multiple platforms. The PostgreSQL and MySQL tools are generally available free or at a low cost. Many other systems and tools exist, but these are the most common ones you will encounter. All of them have different strengths and weaknesses. Although the big three (Oracle, DB2, and SQL Server) can carry high price tags, the vendors can provide detailed support. All three of the vendors also provide inexpensive (or free) versions that are particularly useful for learning to use the systems. The free versions have various performance restrictions, but generally support fairly active smaller databases. You can download the free copies from the company Web sites.

Choosing a database system can be a major challenge. Many larger organizations standardize on a major vendor, negotiate reduced license costs, and make it available throughout the organization for all projects. However, if you need to choose a DBMS for a specific project, you want to carefully investigate the vendor options.

The premiere database systems are useful for large projects, offer extensive options and control over thousands of detailed features. However, these options make it difficult for beginners to understand the major concepts. It is generally best if you begin your studies with a simpler database system or stick with a smaller subset of options.



Item	Description	Quantity
998	Dog Food	12
764	Cat Food	11

Figure 1.18

Hierarchical database. To retrieve data, the DBMS starts at the top (customer). When it retrieves a customer, it retrieves all nested data (order, then items ordered).

The Evolution of Database Management Systems

How have database management systems changed over time? Developers quickly realized that many business applications needed a common set of features for sharing data, and they began developing database management systems. Developers gradually refined their goals and improved their programming techniques. Many of the earlier database approaches still survive, partly because it is difficult to throw away applications that work. It is worth understanding some of the basic differences between these older methods. The following discussion simplifies the concepts and skips the details. The purpose is to highlight the differences between these various database systems—not to teach you how to design or use them.

The earliest database management systems were based on a hierarchical method of storing data. The early systems were an extension of the COBOL file structure. To provide flexible access, these systems were extended with network databases. However, the relational database approach originated by E. F. Codd eventually became the dominant method of storing and retrieving data. As programming methodologies changed to object-oriented techniques, developers started looking for ways to save internal object data in databases and some OO databases were created.With the high popularity of some Web sites such as social networks, new systems have been created to handle the huge amounts of specialized data. In particular, data is often stored in simple key-value pairs. A user uploads content, the application generates an ID value and the content is stored in a specialized DBMS to be quickly retrieved using the generated key value.

Hierarchical Databases

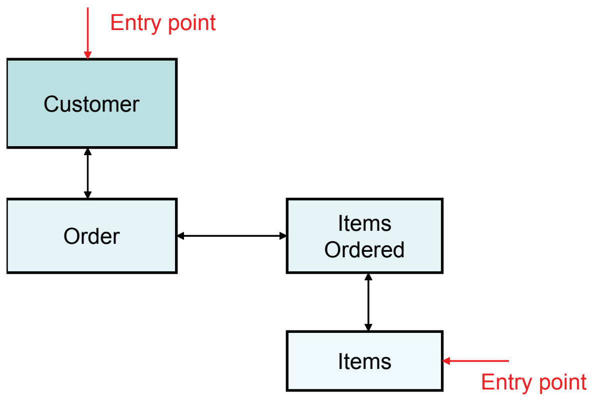
The **hierarchical database** approach begins by claiming that business data often exhibits a hierarchical relationship. For example, a small office without computers might store data in filing cabinets. The cabinets would be organized by customer. Each customer section would contain folders for individual orders, and the orders would list each item being purchased: Customer -> Orders -> Items. To store or retrieve data, the database system must start at the top—with a customer in this example. As shown in Figure 1.18, when the database stores the customer data, it stores the rest of the hierarchical data with it.

The hierarchical database approach is relatively fast—as long as you only want to access the data from the top. The most serious problem related to data storage is the difficulty of searching for items in the bottom or middle of the hierarchy. For example, to find all of the customers who ordered a specific item, the database would have to inspect each customer, every order, and each item.

The hierarchical model is an old concept in management. Many people are familiar with hierarchical objects and have a tendency to classify items using hierarchies. Consequently, hierarchical methods come into fashion every few years. However, the relational approach is substantially better at storing and retrieving data; so you have to be careful when you encounter new data formats. **Extensible markup language (XML)** is a good example. XML is a standard defined to support the transfer of data between diverse machines and companies. All data is marked with tags using angle brackets. The person or group transferring the data is free to create any labels or structure for the tags. For example, you might define an <Order> tag to transfer purchase order data. The structure of an XML file tends to be hierarchical instead of relational. It is designed to be parsed or searched from the top down. Most DBMSs have implemented the XML data type that enables you to store the raw XML file as a single unit within the database. This approach keeps the hierarchical structure of the XML file. If you use this approach, you need a way to search within the XML file. Many DBMSs support the standard **XQuery** tool for searching XML data. With this approach, you first use the relational database to locate a particular XML file, and then call XQuery to find individual items within that file. In effect, it squeezes a hierarchical dataset into a single cell in a relational database table. This approach has limitations but

Figure 1.19

Network database. All data sets must be connected with indexes as indicated by the arrows. Likewise, all entry points (starting point for a query) must be defined and created before the question can be answered.



works if you really need to keep the XML file in one piece. On the other hand, most DBMS vendors recommend that if you want to search the data received in an XML file, you should parse the data out and store the individual elements into relational database tables. Essentially, you should generally use XML as a transfer mechanism and not a storage method.

Network Databases

The **network database** has nothing to do with physical networks (e.g., local area networks). Instead, the network model is named from the network of connections between the data elements. The primary goal of the network model was to solve the hierarchical problem of searching for data from different perspectives.

Figure 1.19 illustrates the Customer, Order, and Item data components in a network model. First, notice that the items are now physically separated—typically stored in separate files. Second, note that they are connected by arrows. Finally, notice the entry points, which are indicated with arrows. The entry points are predefined items that can be searched. In all cases the purpose of the arrows is to show that once you enter the database, the DBMS can follow the arrows to find and display matching data. As long as there is an arrow, the database can make an efficient connection.

Although this approach seems to solve the search problem, the cost is high. All arrows must be physically implemented as indexes or embedded pointers. Essentially, an index duplicates every key data item in the associated data set and associates the item with a pointer to the storage location of the rest of the data. The problem with the network approach is that the indexes must be built before the user can ask a question. Consequently, the developer must anticipate every possible question that users might ask about the data. Worse, building and maintaining the indexes can require huge amounts of processor time and storage space.

Relational Databases

E. F. Codd originated the **relational database** approach in the 1970s, and within several years three elements came together to make the relational database the predominant method for storing data. First, theoreticians defined the basic concepts and illustrated the advantages. Second, programmers who built database management system software created efficient components. Third, hardware performance improved to handle the increased demands of the system.

Figure 1.20 illustrates how the four basic tables in the example are represented in a relational database. The key is that the tables (called "relations" by Codd) are sets of data. Each table stores attributes in columns that describe specific entities.

Figure 1.20

Relational database. Data is stored in separate sets of data. The tables are not physically connected; instead, data is linked between columns. For example, when retrieving an order, the database can match and retrieve the corresponding customer data based on CustomerID.

Customer(CustomerID, Name, …)
Order(OrderID, CustomerID, OrderDate, …)
ItemsOrdered(OrderID, ItemID, Quantity, …)
Items(ItemID, Description, Price, …)

These data tables are not physically connected to each other. The connections exist through the matching data stored in each table. For example, the Order table contains a column for CustomerID. If you find an order that has a CustomerID of 15, the database can automatically find the matching CustomerID and retrieve the related customer data.

The strength of the relational approach is that the designer does not need to know which questions might be asked of the data. If the data is carefully defined (see Chapters 2 and 3), the database can answer virtually any question efficiently (see Chapters 4 and 5). This flexibility and efficiency is the primary reason for the dominance of the relational model. Most of this book focuses on building applications for relational databases.

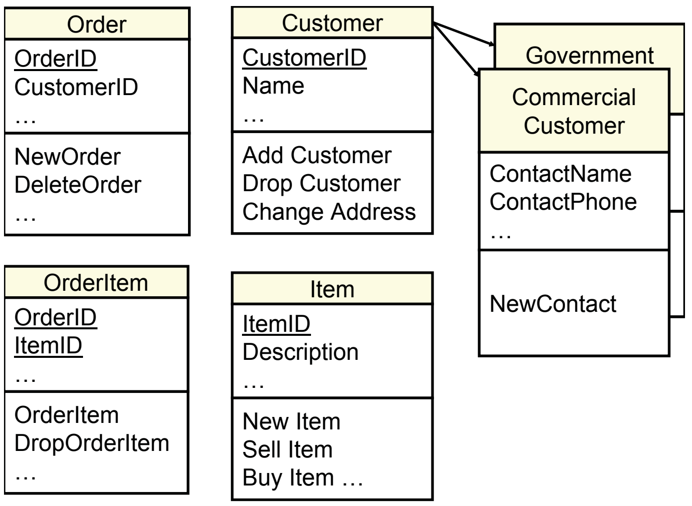
Object-Oriented Databases

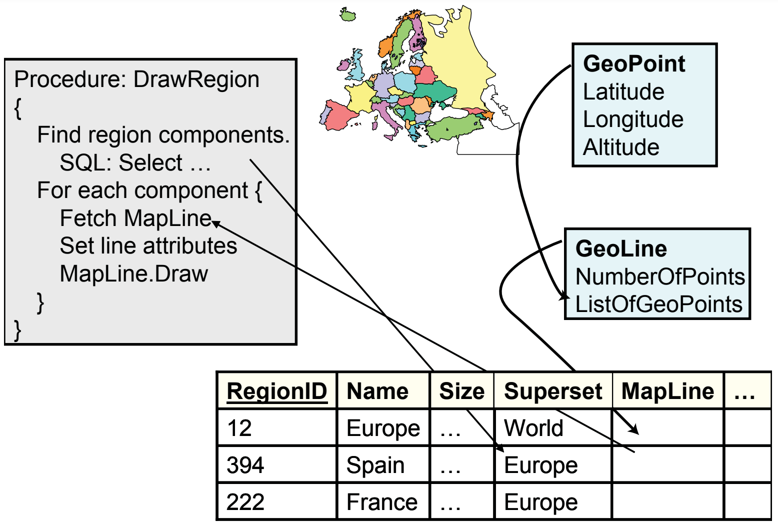
An **object-oriented (OO) database** is a different method of organizing data. The OO approach began as a new method to create programs. The goal is to define objects that can be reused in many programs—thus saving time and reducing errors. As illustrated in Figure 1.21, a class or object has three major components: a name, a set of properties or attributes, and a set of methods or functions. The properties describe the object—just as attributes describe an entity in the relational database. *Methods* are short programs that define the actions that each object can take. For example, the code to add a new

Note: This section contains a relatively detailed description of object-oriented databases and could be skipped for an introductory class. Or read it now and return to it later. Object features add a level of complexity to problems that can be confusing to beginners.

Figure 1.21

Object-oriented database. Objects have properties—just as relational entities have attributes— that hold data to describe the object. Objects have methods that are functions the objects can perform. Objects can be derived from other objects.





RegionID	Name	Size	Superset	MapLine	…
12	Europe	…	World		
394	Spain	…	Europe		
222	France	…	Europe		

Figure 1.22

Abstract data types or objects. A geographic information system needs to store and share complex data types. For example, regions are defined by geographic line segments. Each segment is a collection of points, which are defined by latitude, longitude, and altitude. Using a database makes it easier to find and share data.

customer would be stored with the Customer object. The innovation is that these methods are stored with the object definition.

Figure 1.21 also hints at the power of the OO approach. Note that the base objects (Order, Customer, OrderItem, and Item) are the same as those for the relational approach. However, with the OO approach, new objects can be defined in terms of existing objects. For example, the company might create separate classes of customers for commercial and government accounts. These new objects would contain (inherit) all of the original Customer properties and methods, and also add variations that apply only to the new types of customers.

Two basic approaches are used to handle true object-oriented data: (1) extend the relational model to include typical OO features or (2) create a new object-oriented DBMS. Today, most commercially successful database systems follow the first approach by adding object features to the relational model.

The approach that adds OO features to the relational model is best exemplified by the American National Standards Institute (ANSI). Object-oriented features were a major component to the SQL 99 version. The SQL 2003 standard clarified some of the OO issues as well. In 1997 the SQL3 development group merged with the Object Database Management Group (ODMG). Three features are suggested to add OO capabilities: (1) abstract data types, (2) subtables, and (3) persistent stored modules. DBMS vendors have implemented most of these features.

Object Properties

The first issue involves defining and storing properties. In particular, OO programmers need the ability to create new composite properties that are built from other data types. SQL supports **abstract data types** to enable developers to create

```
CREATE SET TABLE Customer
(
   CustomerID         INTEGER,
   Address            VARCHAR,
   Phone              CHAR(15)
)

CREATE SET TABLE CommercialCustomer
(
   Contact            VARCHAR,
   VolumeDiscount     NUMERIC(5,2)
)
   UNDER Customer;
```

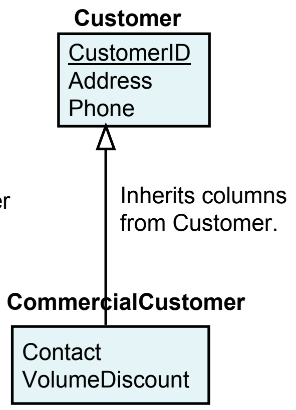


Figure 1.23

SQL subtables. A subtable inherits the columns from the selected supertable. Queries to the CommercialCustomer table will also retrieve data for the CustomerID, Address, and Phone columns inherited from the Customer table.

new types of data derived from existing types. This technique supports inheritance of properties. The type of data stored in a column can be a composite of several existing abstract types. Consider the example shown in Figure 1.22, which shows part of a database for a geographic information system (GIS). The GIS defines an abstract data type for location (GeoPoint) in terms of latitude, longitude, and altitude. Similarly, a line segment (e.g., national boundary), would be a collection of these location points (GeoLine). By storing the data in tables, the application can search and retrieve information based on user requirements. The database also makes it easier to share and to update the data. In the GIS example, the database handles the selection criteria (Region = Europe). The database can also match and retrieve demographic data stored in other tables. The advantage to this approach is that the DBMS handles the data storage and retrieval, freeing the developer to concentrate on the application details.

The abstract data type enables developers to create and store any data needed by the application. The abstract data type can also provide greater control over the application development. First, by storing the data in a DBMS, it simplifies and standardizes the way that all developers access the data. Second, the elements within the data type can be encapsulated. By defining the elements as private, application developers (and users) can only access the internal elements through the predefined routines. For example, developers could be prevented from directly modifying the latitude and longitude coordinates of any location by defining the elements as private.

SQL provides a second method to handle inheritance by defining subtables. A **subtable** inherits all the columns from a base table and provides inheritance similar to that of the abstract data types; however, all the data is stored in separate columns. The technique is similar to the method shown in Figure 1.21, which stores subclasses in separate tables. The difference is that the OO subtables will not need to include the primary key in the subtables. As indicated in Figure 1.23, inheritance is specified with an UNDER statement. You begin by defining the highest level tables (e.g., Customer) in the hierarchy. Then when you create a new table (e.g., CommercialCustomer), you can specify that it is a subtable by adding the UNDER statement. If you use the unified modeling language (UML) triangle-pointer notation, or an IS-A icon for inheritance, it will be easy to create the tables in SQL. Just define the properties of the table and add an UNDER statement if there is a "pointer" to another table.

Do not worry about the details of the CREATE TABLE command. Instead, it is important to understand the difference between abstract data types and subtables. An abstract data type is used to set the type of data that will be stored in one column. With a complex data type, many pieces of data (latitude, longitude, etc.) will be stored within a single column. With a subtable the higher level items remain in separate columns. For example, a subtable for CommercialCustomer could be derived from a base Customer table. All the attributes defined by the Customer table would be available to the CommercialCustomer as separate columns.

Object Methods

Each abstract data type can also have methods or functions. In SQL, the routines are called **persistent stored modules**. They can be written as SQL statements. The SQL language is also being extended with programming commands—much like Oracle's PL/SQL extensions. Routines are used for several purposes. They can be used as code to support triggers, which have been added to SQL. Persistent routines can also be used as methods for the abstract data types. Designers can define functions that apply to individual data types. For example, a GIS location data type could use a subtraction operator that computes the distance between two points.

To utilize the power of the database, each abstract data type should define two special functions: (1) to test for equality of two elements and (2) to compare elements for sorting. These functions enable the DBMS to perform searches and to sort the data. The functions may not apply to some data types (e.g., sound clips), but they should be defined whenever possible.

Object-Oriented Languages and Sharing Persistent Objects

The development of true OODBMS models was initiated largely in response to OO programmers who routinely create their own objects within memory. They needed a way to store and share those objects. Although the goals may appear similar to the modified-relational approach, the resulting database systems are unique.

Most OO development has evolved from programming languages. Several languages were specifically designed to utilize OO features. Common examples include C++, Smalltalk, and Java. Data variables within these languages are defined as objects. Each class has defined properties and methods. Currently, developers building applications in these languages must either create their own storage mechanisms or translate the internal data to a relational database.

Complex objects can be difficult to store within relational databases. Most languages have some facility for storing and retrieving data to files, but not to databases. For example, C++ libraries have a serialize function that transfers objects directly to a disk file. There are two basic problems with this approach: (1) it is difficult to search files or match data from different objects, and (2) the developer is responsible for creating all sharing, concurrency, and security operations. However, this approach causes several problems because data is now intrinsically tied to the programs and is no longer independent.

- InterSystems Caché
- Progress Software ObjectStore
- Objectivity
- McObject Perst
- Versant
- JADE

Figure 1.24

OODBMS vendors and products. Each tool has different features and goals. Contact the vendors for details or search the Web for user comments.

Essentially, OO programmers want the ability to create **persistent objects**, that is, objects that can be saved and retrieved at any time. Ideally, the database would standardize the definitions, control sharing of the data, and provide routines to search and combine data. The basic difficulty is that no standard theory explains how to accomplish all these tasks. Nonetheless, as shown in Figure 1.24, several OODBMS exist, and users have reportedly created many successful applications with these tools. But, most of these tools have minimal market share and may no longer exist.

The key to an OODBMS is that to the programmer it simply looks like extended storage. An object and its association links are treated the same whether the object is stored in RAM or shared through the DBMS. Clearly, these systems make development easier for OO programmers. The catch is that you have to be an OO programmer to use the system at all. In other words, if your initial focus is on OO programming, then a true OODBMS may be useful. If you started with a traditional relational database, you will probably be better off with a relational DBMS that has added OO features.

In theory, the 1997 agreements between ANSI and ODMG were designed to bring the SQL and OODBMS models closer to a combined standard. In practice, it could take a few years and considerable experimentation in the marketplace. For now, if you are serious about storing and sharing objects, you will have to make a choice based on your primary focus: OO programming or the relational database. As of 2010, the relational database approach with some OO extensions appears to have won out over pure OO database systems. Otherwise, programmers simply write data objects to individual files. If speed is an overriding issue, simple files are often the best answer.

Key-Value Pairs: Cassandra

The expanding use of highly-popular Web sites has created a need for high-performance, specialized data storage. With hundreds of millions of users uploading megabytes of data every hour (or minute), banks of servers are needed to handle the heavy demands. In many cases, the data to be stored is also non-traditional—it consists of complex objects including photos, raw files, blog entries, or various text items. Time is also a common feature, where people want to store multiple versions or organize data by the time an object was created. Initially, most big sites created proprietary storage methods to handle their unique situation. More recently, some people have started sharing portions of their ideas and works. One tool has gained some popularity and illustrates some of the features useful in these situations. The database system is called Cassandra and is available as an open source project for several different hardware systems—although Linux-based systems are the most popular.

Storing data in Cassandra is unlike any of the earlier methods. Cassandra does borrow ideas from several advanced features of relational systems but trying to compare it with relational systems leads to confusion for beginners, so the details are not covered in this chapter. The two most important points are that data values are stored and accessed via a key item, and that the data is deliberately designed to be spread across multiple servers. By eliminating "relationships" data can be split into mostly independent pieces. Spreading these pieces across multiple servers enables the system to harness the power of massively parallel systems to perform storage, retrieval, and searches simultaneously on thousands of machines working independently. Chapter 13 explores the issues of design and querying for non-relational systems, including examples for Cassandra.

Drawbacks to Database Management Systems

What potential problems exist with a DBMS approach? The discussion of OO systems brings up the most common criticism of a DBMS: performance. The DBMS is a layer of complex software between the application and the data storage. Although this layer provides many useful features, it can slow down the storage and retrieval of huge amounts of data. And huge amounts of data are where the problems arise. Automated systems can easily generate gigabytes of data per hour or even faster. Writing massive amounts of data to a disk drive taxes the transfer capabilities even for fast servers. Pushing the data through a DBMS adds overhead for backup-and-recovery, concurrency controls, and indexes. It is possible for a DBMS to add two-to-three times the amount of data written for each byte of original data.

The high-end DBMSs provide tools to analyze data storage and to improve read/write times for large data transfers. But, you must always consider the possibility that it might be necessary to bypass the DBMS and store some data directly to the file system.

For example, it is possible to store images and even video data directly into a database table. However, in many applications—particularly Web-based ones—it is better to store the raw data in separate files and then store just the filename in the database. One reason this approach works is because this data rarely needs to be altered—once a file is stored, it is rarely edited, or the editing is not relevant to the DBMS. Consequently, there is little need to control for multiple users or to create repeated backups.

The problem of overhead created by the DBMS arises every few years in the industry. For example, around 2010 several writers began pushing the value of "non-SQL" database systems. The term is inaccurate because the query language SQL is not really the problem. Instead, some examples of extremely large databases create performance issues in terms of storage. Creating storage mechanisms for extremely large databases is difficult. But, before discarding the DBMS approach, you must seriously consider what features you are willing to give up. DBMS performance is slower than directly writing data to a disk largely because the DBMS provides safety through backup logs and concurrency controls. Alternative storage methods that bypass these features can provide faster performance, but you often give up the safety and security controls. When you evaluate alternatives, be sure that you understand exactly what features are being removed to provide performance increases. And then determine whether you need those features or how they might be provided through other methods.

The examples used in this book, and the Workbooks, are small enough that performance is not an issue. All of the data can be stored in individual tables. However, you should learn to recognize potential problems so that when you work on large-scale applications, you can choose the appropriate time to bypass the DBMS and store files directly to a server.

The other major drawback to a DBMS is the cost of the software. For relatively small projects, this cost can be small or even zero. Microsoft Access works for small projects. You can also obtain free copies of software for Oracle, SQL Server and IBM's DB2. These copies have size and performance limitations but work for many smaller projects. You can also obtain open-source software such as MySQL (now controlled by Oracle) and PostgreSQL. These two have been used for relatively large-scale commercial Web sites. However, keep in mind that "open-source" ultimately means that you pay for maintenance and support—either by paying a third party or by hiring more people. So, open-source is not actually a zero-cost option.

Application Development

What is an application? If you carefully examine Figures 1.16, 1.17, and 1.18, you will notice that they all have essentially the same data sets. This similarity is not an accident. Database design methods described in Chapters 2 and 3 should be followed regardless of the method used to implement the database. In other words, any database project begins by identifying the data that will be needed and analyzing that data to store it as efficiently as possible.

The second step in building applications is to identify forms and reports that the users will need. These forms and reports are based on queries, so you must create any queries or views that will be needed to produce the reports and forms as described in Chapters 4 and 5. Then you use the report writer and forms generator to create each report and form as described in Chapters 6 and 7. As described in Chapter 8, the next step is to combine the forms and reports into an application that handles all of the operations needed by the user. The goal is to create an application that matches the jobs of the users and helps them to do their work.

Chapter 7 describes how to deal with common problems in a multiuser environment to protect the integrity of the data and support transactions. Chapter 9 shows one more design method of storing data: the data warehouse. To deal with large databases, transaction processing systems use indexes and other features to optimize storage tasks. Today, managers want to retrieve and analyze the data. Data warehouses provide special designs and tools to support online analytical processing (OLAP).

When the application is designed and while it is being used, several database administration tasks have to be performed. Setting security parameters and controlling access to the data is one of the more important tasks. Chapter 10 discusses various administration and security issues.

As an organization grows, computer systems and applications become more complex. An important feature in modern organizations is the need for users to access and use data from many different computers throughout the organization. At some point you will need to increase the scope of your application so that it can be used by more people in different locations. Distributed databases discussed in Chapter 11 are a powerful way to create applications that remove the restrictions of location. The Internet is rapidly becoming a powerful tool for building and implementing database applications that can be used by anyone around the world. The same technologies can be used for applications that are accessed only by in-house personnel. Systems that use Internet technology but limit access to insiders are called intranets.

Chapter 12 introduces the considerations of how the DBMS physically stores data. This chapter is particularly helpful for students who have a background in programming, but the topics are presented carefully so that non-programmers can understand the issues as well. The basic point is that high-end DBMSs allow the administrator to control how the data is stored in operating system files. This level of control is sometimes needed to improve the performance of large databases.

Introduction to this Book's Databases

What databases are used with this book? Several databases are used as examples in this book. The workbook has an additional database as well but it is described in the workbook. These databases are important because they provide concrete examples of various issues in database design, queries, and application development. You can study the databases to help you understand the topics discussed in this book. Bear in mind that the databases are not completed. In fact, each database is at a different level of completion so that you can see how an application is built in stages.

The main database in this book is Sally's Pet Store. The design is complete, and some forms and reports have been created, but many application features need to be added. The Corner Med database is newer and it is designed to be a smaller application and to provide more examples and exercises that illustrate some common issues in the healthcare industry. Rolling Thunder Bicycles is a relatively large database, in terms of design, application, and data. It was originally developed in Microsoft Access and contains many detailed forms. Scripts exist to build the data tables in other DBMSs, but the forms and reports have not been converted. The All Powder Board and Ski Shop in the Workbooks has similarities to Rolling Thunder, and many of the forms and reports have been developed for multiple DBMSs. Keep in mind that the purpose of the Workbook is to show you all of the steps in building an application.

Sally's Pet Store

A young lady with a love for animals is starting a new type of pet store. Sally wants to match pets with owners who will take good care of the animals. The Pet Store database was changed for this edition of the book. Specifically, the store no longer "sells" animals. Instead, animals are brought in for adoption by various local adoption organizations. Customers donate money to the organization to adopt an animal. The donations are handled by the store clerks at checkout time and the accumulated donations are paid to the adoption organizations. This approach cuts down on "backyard breeders," and enables Sally and her customers to support local charitable organizations that help find homes for animals.

At the moment Sally has only one store, but she dreams of expanding into additional cities. She wants to hire and train workers to be "animal friends," not salespeople. These friends will help customers choose the proper animal. They will answer questions about health, nutrition, and pet behavior. They will even be taught that some potential customers should be convinced not to buy an animal.

Because the workers will spend most of their time with the customers and animals, they will need technology to help them with their tasks. The new system will also have to be easy to use, since little time will be available for computer training.

Even based on a few short discussions with Sally, it is clear that the system she wants will take time to build and test. Fortunately, Sally admits that she does not need the complete system immediately. She has decided that she first needs a basic system to handle the store operations: sales, orders, customer tracking, and basic animal data. However, she emphasizes that she wants the system to be flexible enough to handle additional features and applications.

Details of Sally's Pet Store will be examined in other chapters. For now, you might want to visit a local pet store or talk to friends to get a basic understanding of the problems they face and how a database might help them.

Corner Med

Corner Med is currently a small medical office with big plans. Eventually, the owners want to franchise the concept and create a chain of walk-in medical offices that are affordable and accessible to customers in cities around the country. Currently, the company is run by a handful of physicians, supported by nurses and a few clerical staff. For the most part, the physicians focus on family practice and handle routine medical exams and common problems. More complex cases are referred to specialists, but the doctors at Corner Med are often responsible for the initial diagnosis and generally participate in the long-term care of the patients.

The company has a small medical lab and can perform simple tests, such as basic blood workups and routine x-rays. More complex tests and procedures such as CT scans and MRIs are handled by specialty firms available in every major city.

In terms of a business application, Corner Med wants to keep basic patient information in a database instead of thousands of paper folders. Of course, security and privacy issues become critical—particularly if the company eventually decides to centralize the data for multiple offices. One simplifying aspect of medical management and billing is that insurance companies along with various governments, and ultimately, the World Health Organization (WHO), have defined a common set of numbers used to define diagnoses (conditions) and procedures (treatments). The main reference is the International Disease Classification system. It is commonly referenced by its initials and the version number. The most commonly-used version is ICD-9. However, ICD-10 has been introduced and the U.S. government is currently stating that all medical organizations are supposed to switch to ICD-10 by 2013.

The version of the database for this edition has been modified to use the ICD-10 diagnosis and procedure codes (two sets). These codes are available for download from the U.S. government Web sites. The database also uses a DrugListing file which contains registered drugs directly from the FDA Web site. However, the original ICD-9 codes have been kept in the database tables to provide examples in converting data from the older codes to the newer ones.

The sample data for patient visits is derived from a U.S. physician survey. The names are fictional, but gender and diagnoses and procedures were created by randomly drawing data from that survey. The fees and payments are weak estimates and are likely to be inaccurate, but they illustrate the concepts.

Physicians are familiar with the diagnostic and procedure terms, but they rarely memorize the entire list of ICD codes. Many hospitals and large practices hire medical encoders to translate the physician's descriptions into the proper codes. In some ways, the codes simplify the medical database. But, as you will see, they complicate the user interface because you have to find a way to make the list easy to use.

Rolling Thunder Bicycles

The Rolling Thunder Bicycle Company builds custom bicycles. Its database application is much more complete than the Pet Store application, and it provides an example of how the pieces of a database system fit together. This application also contains many detailed forms that illustrate the key concepts of creating a user interface. Additionally, most of the forms contain programming code that handles common business tasks. You can study this code to help you build your own applications. The Rolling Thunder application has a comprehensive help system that describes the company and the individual forms. The database contains realistic data for hundreds of customers and bicycles.

One of the most important tasks at the Rolling Thunder Bicycle Company is to take orders for new bicycles. Several features have been included to help non-experts select a good bicycle. As the bicycles are built, the employees record the construction on the Assembly form. When the bicycle is shipped, the customers are billed. Customer payments are recorded in the financial forms. As components are installed on bicycles, the inventory quantity is automatically decreased. Merchandise is ordered from suppliers, and payments are made when the shipments arrive.

The tasks performed at Rolling Thunder Bicycles are similar to those in any business. By studying the application and the techniques, you will be able to create solid applications for any business.

Starting a Project: The Feasibility Study

What are the first steps to start a project? Ideas for information systems can come from many sources: users, upper management, information system analysts, competitors, or firms in other industries. Ideas that receive initial support from several people might be proposed as new projects. If the project is small enough and easy to create, it might be built in a few days. Larger projects require more careful study. If the project is going to involve critical areas within the organization, require expensive hardware, or require substantial development time, then a more formal feasibility study is undertaken.
Feasibility studies are covered in detail within systems analysis texts. However, because of their unique nature, it is helpful to examine the typical costs and benefits that arise with the database approach.

The goal of a feasibility study is to determine whether a proposed project is worth pursuing. The study examines two fundamental categories: costs and potential benefits. As noted in Figure 1.25, costs are often divided into two categories: up-front or one-time costs and ongoing costs once the project is operational. Benefits can often be found in one of three categories: reduced operating costs, increased value, or strategic advantages that lock out competitors.

Costs

Almost all projects will entail similar up-front costs. The organization will often have to purchase additional hardware, software, and communication equipment (such as a Web server or expand a local area network). The cost of developing the

- Costs
  - Up-front/one-time
    - Software
    - Hardware
    - Communications
    - Data conversion
    - Studies and design
    - Training
  - Ongoing costs
    - Personnel
    - Software upgrades
    - Supplies
    - Support
    - Hardware maintenance

- Benefits
  - Cost savings
    - Software maintenance
    - Fewer errors
    - Less data maintenance
    - Less user training
  - Increased value
    - Better access to data
    - Better decisions
    - Better communication
    - More timely reports
    - Faster reaction to change
    - New products and services
  - Strategic advantages
    - Lock out competitors

Figure 1.25

Common costs and benefits from introducing a database management system. Note that benefits can be hard to measure, especially for tactical and strategic decisions. But it is still important to list potential benefits. Even if you cannot assign a specific value, managers need to see the complete list.

system is listed here, including the cost for all additional studies. Other one-time costs include converting data to the new system and initial training of users. Database management systems are expensive software items. For example, for larger projects, the cost for software such as Oracle can easily run to several million dollars. You will also have to purchase "maintenance" upgrades of the software at least on an annual basis.

Hardware and software costs can be estimated with the help of vendors. As long as you know the approximate size of the final system (e.g., number of users), vendors can provide reasonably accurate estimates of the costs. Data conversion costs can be estimated from the amount of data involved. The biggest challenge often lies in estimating the costs of developing the new system. If an organization has experience with similar projects, historical data can be used to estimate the time and costs based on the size of the project. Otherwise, the costs can be estimated based on the projected number of people and hours involved.

Once the project is completed and the system installed, costs will arise from several areas. For example, the new system might require additional personnel and supplies. Software and hardware will have to be modified and replaced—entailing maintenance costs. Additional training and support might be required to deal with employee turnover and system modifications. Again, most of these costs are straightforward to estimate—as long as you know the size of the project.

Unfortunately, information system (IS) designers have not been very successful at estimating the costs. For example, in January 1995 PC Week reported that 31 percent of new IS projects are canceled before they are completed. Additionally, 53 percent of those that are completed are 189 percent over budget. This pattern is not unique. A study published in MIS Quarterly in 2000 also estimated that 30-40 percent of projects "escalated" into late or over budget status. The greatest difficulty is in estimating the time it takes to design and develop new software. Every developer is different with large variations in programmer productivity. In large projects, where the staff members are constantly changing, accurately predicting the amount of time needed to design and develop a new system is often impossible. Nonetheless, managers need to provide some estimate of the costs. On a related note, building anything new can be difficult to estimate in terms of time and cost. The Boeing 787 "Dreamliner" took several years longer and millions of extra dollars to design and build than originally anticipated. The problem in estimating information systems and physical systems is that you need to predict the future, and it is probably impossible to anticipate every possible problem that might arise.

Benefits

In many cases benefits are even more difficult to estimate. Some benefits are tangible and can be measured with a degree of accuracy. For instance, transaction processing systems are slightly easier to evaluate than a decision support system, since benefits generally arise from their ability to decrease operations costs. A system might enable workers to process more items, thus allowing the firm to expand without increasing labor costs. A database approach might reduce IS labor costs by making it easier for workers to create and modify reports. Finally, a new information system might reduce errors in the data, leading to improved decisions.

Many benefits are intangible and cannot be assigned specific monetary values. For instance, benefits can arise because managers have better access to data. Communication improves, better decisions are made, and managers can react faster to a changing environment. Similarly, the new system might enable the company to produce new products and services or to increase the sales of ancillary products to existing customers. Similarly, firms might implement systems that provide a competitive advantage. For example, an automated order system between a firm and its customers often encourages the customers to place more orders with the firm. Hence the firm gains an advantage over its competitors.

When information systems are built to automate operations-level tasks and the benefits are tangible, evaluating the economic benefits of the system is relatively straightforward. The effects of improving access to data are easy to observe and measure in decreased costs and increased revenue. However, when information systems are implemented to improve tactical and strategic decisions, identifying and evaluating benefits is more difficult. For instance, how much is it worth to a marketing manager to have the previous week's sales data available on Monday instead of waiting until Wednesday?

In a database project benefits can arise from improving operations—which leads to cost savings. Additional benefits occur because it is now easier and faster to create new reports for users, so less programmer time will be needed to modify the system. Users can also gain better access to data through creating their own queries—instead of waiting for a programmer to write a new program.

Database projects can provide many benefits, but the organization will receive those benefits only if the project is completed correctly, on time, and within the specified budget. To accomplish this task, you will have to design the system carefully. More than that, your team will have to communicate with users, share work with each other, and track the progress of the development. You need to follow a design methodology.

Summary

One of the most important features of business applications is the ability to share data with many users at the same time. Without a DBMS sharing data causes several problems. For example, if data definitions are stored within each separate program, making changes to the data file becomes very difficult. Changes in one program and its data files can cause other programs to crash. Every application would need special code to provide data security, concurrency, and integrity features. By focusing on the data first, the database approach separates the data from the programs. This independence makes it possible to expand the database without crashing the programs.

A DBMS has many components. Required features include the database engine to store and retrieve the data and the data dictionary to help the DBMS and the user locate data. Other common features include a query language, which is used to retrieve data from the DBMS to answer business questions. Application development tools include a report writer, a forms generator, and an application generator to create features like menus and help files. Advanced database systems provide utilities to control secure access to the data, cooperate with other software packages, and communicate with other database systems.

Database systems have evolved through several stages. Early hierarchical databases were fast for specific purposes but provided limited access to the data. Network databases enabled users to build complex queries but only if the links were built with indexes in advance. The relational database is currently the leading approach to building business applications. Once the data is defined carefully, it can be stored and retrieved efficiently to answer any business question. The OO approach is a new technique for creating software. Object-oriented systems enable you to create your own new abstract data types. They also support subtables, making it easier to extend a class of objects without redefining everything from scratch. Recently, key-value databases are being developed to handle massive loads of complex data types but they are still evolving.

Regardless of the type of database implemented, application development follows similar steps. First, identify the user requirements, determine the data that needs to be collected, and define the structure of the database. Then, develop the forms and reports that will be used, and build the queries to support them. Next, combine the various elements into a polished application that ties everything together to meet the user needs. If necessary, distribute the database across the organization or through an Internet or intranet. Additional features can be provided by integrating the database with powerful analytical and presentation tools, such as spreadsheets, statistical packages, and word processors.

A Developer's View

For Miranda to start on her database project, she must first know the strengths of the tools she will use. At the starting point of a database project, you should collect information about the specific tools that you will use. Get the latest reference manuals. Install the latest software patches. Set up work directories and project space. For a class project, you should log on, get access to the DBMS, make sure you can create tables, and learn the basics of the help system.

Key Terms

abstract data types
data dictionary
data independence
data mining
database
database engine
database management system (DBMS)
extensible markup language (XML)
feasibility study
forms development
hierarchical database
intranets
network database
object-oriented (OO) database
online analytical processing (OLAP)
persistent objects
persistent stored modules
relational database
report services
report writer
subtable
XQuery

Review Questions

1. What features does a DBMS provide that make application development easier?
2. What are the basic components of a DBMS?
3. Why is data independence important and how is it achieved with a DBMS?
4. Why is the relational database approach better than earlier methods?
5. How do relational databases implement object-oriented features?
6. What potential drawbacks exist to a DBMS?
7. What are the main steps in application development with a database system?
8. What is the purpose of a feasibility study?
9. Why do many of the biggest Web sites use non-relational databases?

Exercises

1. Create a new database with the two tables shown in the figure. Feel free to add more data. Be sure to set a primary key for the underlined columns. Be sure to create a relationship that links the two tables. Use a report wizard to create the report shown. You should be able to use a visual tool to create the tables. Otherwise, check Chapter 3 for the syntax of the CREATE TABLE command.

Employee

EmployeeID	LastName	FirstName	Address	DateHired
332	Ant	Adam	354 Elm	5/5/1964
442	Bono	Sonny	765 Pine	8/8/1972
553	Cass	Mama	886 Oak	2/2/1985
673	Donovan	Michael	421 Willow	3/3/1971
773	Moon	Keith	554 Cherry	4/4/1972
847	Morrison	Jim	676 Sandalwood	5/5/1968

Client

ClientID	LastName	FirstName	Balance	EmployeeID
1101	Jones	Joe	113.42	442
2203	Smith	Mary	993.55	673
2256	Brown	Laura	225.44	332
4456	Cieter	Jackie	664.90	442
5543	Wodkoski	John	984.00	847
6673	Sanchez	Paula	194.87	773
7353	Chen	Charles	487.34	332
7775	Hagen	Fritz	595.55	673
8890	Hauer	Marianne	627.39	773
9662	Nguyen	Suzie	433.88	553
9983	Martin	Mark	983.31	847

```
Report

Ant, Adam          5/5/1964
           Brown, Laura        225.24
           Chen, Charles        47.34
                               712.58
Bono, Sonny
           Dieter, Jackie      664.90
           Jones, Joe          114.32
                               779.22
```

2. Read the documentation to your DBMS and write a brief outline that explains how to:
   a) Create a table.
   b) Create a simple query.
   c) Create a report.
3. Describe two business or Web applications that could use a DBMS. Identify some of the main data elements that would be collected.

4. Find a reference or check Web sites so you can compare the specifications on three free DBMS products. At least one of the products should be from a major commercial vendor and one from an open-source or free source.

5. Describe how a university club or student organization could use a database to improve its service operations.

6. A company wants you to build a custom Web site to support sales of computer cables over the Internet. The company anticipates receiving an average of 100 orders per business day, with an average of $57.19 per order. Gross profit margins (including credit card costs) are about 15 percent. Shipping costs are priced directly so do not affect profits. No new workers will be needed to package and ship the orders, but the company expects to hire a designer with a salary of about $25,000, to keep the product list up to date with photos and descriptions. The initial costs include cost of the computer hardware and software ($10,000) and the development cost ($35,000). The ISP costs will be about $400 per month. Annual maintenance costs are expected to be $1000 per year. Assuming a project life of five years and an interest rate of 8 percent, compute the economic feasibility of the project. Compare the costs and benefits to the alternative of selling the items through Amazon instead.

7. You have just been hired by a company and have wandered around talking to people. A few accountants have developed a database-driven application to handle fixed-assets and track depreciation. They claim the system has enabled them to function with one less accounting clerk ($30,000/year). A guy in finance has created a custom database in Microsoft Access to generate reports on a set of financial investments. Much of the data comes from brokerage firms and he notes that he is able to save 10 hours a week in clerical time (minimum wage). However, he complained about the difficulty of loading data from the main company database and says he spends 2 hours a week typing in data from printed reports. Two people in marketing have created separate databases to track survey and sales results for their projects. They claim the projects have not saved any labor costs, but enhance sales by at least $1 million a year. Yesterday, your manager said that all of these people complained about their existing systems and the inability to get data from the corporate database. You need to define projects for each group, identify the cost of developing a new system and the potential benefits. Rank the projects by economic return and make a recommendation to management.

8. Find two companies that provide Web-based database hosting and compare the basic costs for running an online relational database with 500 GB of data, a medium-sized server, and 2 TB of monthly data transfer.

9. Find a company, government agency, or a Web site (perhaps through an article or blog) and briefly explain the data collected and how the database is used.

10. Use articles or blogs to identify a Web site that relies on a non-relational (NoSQL) database and briefly explain the benefits of using that approach over a relational DBMS in this situation.

Sally's Pet Store

11. Install the Pet Store database or find it on your local area network if it has already been installed. Print out (or write down) the list of the tables used in the database. Use the Help command to find the version number of Microsoft Access that you are using.

12. Visit a local pet store and make a list of 10 merchandise items and five animals for sale. Enter this data into the appropriate Pet Store database tables.

13. Identify the hardware and software that would be required to install this system in a typical Pet Store. Estimate the costs and the time required to build and install the system.

14. Outline the basic tasks that take place in running a pet store. Identify some of the basic data items that will be needed.

15. Find an online pet store and estimate the number of different products for sale. Assume the company has 500,000 customers and handles 1,000 sales a day with about 3 items per sale. Assume it takes 300 bytes to store data for one product, 80 bytes for a customer, and 500 bytes for a sale. Estimate the amount of data needed to be saved in one year.

Rolling Thunder Bicycles

16. Install the Rolling Thunder database or find it on your local area network if it has already been installed. Using the BicycleOrder form, create an entry for a new bicycle.

17. Use the Rolling Thunder Help system, or the Web site description, to briefly describe the firm and its major processes. Identify the primary business entities in the company.

18. Use Web sites or visit a local bike shop to find prices for at least two bicycles. Try to find the most expensive bicycle you can.

19. Refer to the relationship/class diagram to explain what a Component is and how it is connected to a Bicycle. Give an example from the data.

20. Use the Employee table and Excel to compute the company's total salary costs. What problems might you encounter if you tried a similar approach for a table with 5 million customers?

Corner Med

21. Install the Corner Med database if necessary. Use the Patient form to enter data for a new patient.

22. Use the Patient Visit form to enter data for a patient with at least one diagnostic code and one procedure code. Describe any usability or performance issues that might arise.

23. Make a list with a brief description of other items that the company might want to store in the database.

24. Check the U.S. government Web sites (e.g., Health and Human Services) to see when the conversion to ICD-10 is going to be required. Also, check the Web site to see what information is available for vendors and developers. What problems are likely to arise with the conversion from ICD9 to ICD10 codes?

25. Look at the design for the Corner Med database. Briefly explain the purpose of the three tables: VisitDiagnoses, VisitProcedures, and VisitMedications. Why is it necessary to have three separate tables instead of combining them into a single table?

Web Site References

http://office.microsoft.com/en-us/access/	Microsoft Access
http://www.microsoft.com/en-us/sqlserver/default.aspx	Microsoft SQL Server
http://www.oracle.com	Oracle
http://www.oracle.com/technetwork	Oracle technology network with software downloads
hhttp://www-01.ibm.com/software/data/db2/	IBM DB2
http://www.mysql.com	Free DBMS now controlled by Oracle
http://www.postgresql.org	A better free DBMS
http://www.acm.org	Association for Computing Machinery
http://groups.google.com/groups/dir?sel=usenet%3Dcomp.databases http://dbforums.com http://dbasupport.com http://www.devx.com/outgoing/databasefeed.xml http://www.sqlservercentral.com	Newsgroups for database questions.
http://www.cms.gov/Medicare/Coding/ICD10/index.html	U.S. government Web site (HHS) with ICD-10 codes.

Additional Reading

Keil, M., J. Mann, and A. Rai, “Why Software Projects Escalate,” *MIS Quarterly*,24(4), 2000. [Estimates 30-40 percent of IT development projects escalate above budget.]

Perry, J. and Post, G. *Introduction to Oracle 10g*, Englewood Cliffs: Prentice-Hall, 2007. [A step-by-step introduction to Oracle 10g with several databases.]

Perry, J. and Post, G. *Introduction to SQL Server 2005*, Englewood Cliffs: Prentice-Hall, *2008*. [A step-by-step introduction to SQL Server 2005 and Visual Studio 2005.]

Zikopoulos, P.C. and R.B. Melnyk, *DB2: The Complete Reference*, McGraw-Hill, 2001. [One of few reference books on DB2 and written by IBM employees.]

Part 1

Systems Design

To create a useful application, you must first understand the business and determine how to help the users. In a database context, the most important issue is to identify the data that must be stored. This process requires two basic steps. In Chapter 2 you will design a logical (or conceptual) data model that examines business entities and their relationships. This logical data model is displayed on a class diagram and specifies the various business rules of the company.

The second design step is to create an implementation model of how the data will be stored in the database management system. This step usually consists of creating a list of nicely behaved tables that will make up the relational database. Chapter 3 describes how to create this list of tables and explains why it is important to define them carefully.

Chapter 2

Database Design

Chapter Outline

What You Will Learn in This Chapter

- What is database design and why is it important?
- Why are models important in designing systems?
- How do you begin a database project?
- How do you know what data to put in the database?
- What is a class diagram (or entity-relationship diagram)?
- Is there an easier way to get started with database design?
- How are some common business associations handled in class diagrams?
- Are more complex diagrams different?
- What are the different data types?
- What are events, and how are they described in a database design?
- How are teams organized on large projects?
- How does UML split a big project into packages?
- What is an application?
- What process is followed when starting a project?

A Developer's View

Miranda: Well, Ariel, you were right as usual. A database seems like the right tool for this job.

Ariel: So you decided to take the job for your uncle's company?

Miranda: Yes, it's good money, and the company seems willing to let me learn as I go. But, it's only paying me a small amount until I finish the project.

Ariel: Great. So when do you start?

Miranda: That's the next problem. I'm not really sure where to begin.

Ariel: That could be a problem. Do you know what the application is supposed to do?

Miranda: Well, I talked to the manager and some workers, but there are a lot of points I'm not clear about. This project is bigger than I thought. I'm having trouble keeping track of all the details. There are so many reports and terms I don't know. And one salesperson started talking about all these rules about the data—things like customer numbers are five digits for corporate customers but four digits and two letters for government accounts.

Ariel: Maybe you need a system to take notes and diagram everything they tell you.

Getting Started
Begin by identifying the data that needs to be stored. Group the data into entities or classes that are defined by their attributes. It is often easiest to start with common entities such as Customers, Employees, and Sales, such as Customer(CustomerID, LastName, FirstName, Phone, …). Identify or create primary key columns. Look for one-to-many or many-to-many relationships and use key columns to specify the "many" side. Use the online DBDesign to create a diagram of the entities and relationships. Add a table and decide which attributes (columns) belong in that table. A database design is a model of the business and the tables, relationships and rules must reflect the way the business is operated.

Introduction

What is database design and why is it important? Database management systems are powerful tools but you cannot just push a button and start using them. Designing the database—specifying exactly what data will be stored—is the most important step in building the database. The table is the fundamental concept in a relational database. A table represents entities or classes of objects in the business world (Customer, Employee, Sale, Merchandise, and so on). Its columns define the properties. So the developer's main goal is to identify all of the business entities and their properties which can be turned into tables. As you will see, the actual process of creating a table in a DBMS is relatively easy. The hard part is identifying exactly what columns are needed in each table, determining the primary keys, and determining relationships among tables.

Even the process of defining business entities or classes is straightforward. If you use the DBDesign tool (highly recommended), it is easy to define a business class and add columns to it. The real challenge is that your database design has to match the business rules and assumptions. Every business has slightly different needs, goals, and assumptions. Your design should reflect these rules. Consequently, you first have to learn the individual business rules. Then you have to figure out how those rules affect the database design. In a real-world project, you will need to talk with users and managers to learn the rules. A database represents a model of the organization. The more closely your model matches the original, the easier it will be to build and use the application. This chapter shows how to build a visual model that diagrams the business entities and relationships. Chapter 3 discusses these concepts in more detail and defines specific rules that tables need to follow.

To be successful, any information system has to add value for the users. You need to identify the users and then decide exactly how an information system can help them. Along the way, you identify the data needed and the various business rules. This process requires research, interviews, and cross-checking.

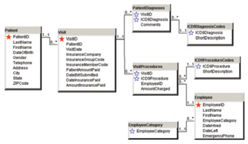
Initially, the main things to look for are: 1. Business entities (Customer, Merchandise, Employee, and so on), 2. Primary keys that identify the entities (CustomerID, SKU, EmployeeID), and 3. Associations or relationships among the entities. In particular, focus on the degree of the association: Can customers place one order or many orders? Is an order assigned to one employee or can many employees be involved? The answers to these questions can depend on the specific business and they will change the overall database design.

Two-Minute Chapter

Relational databases are powerful tools to build business applications. One of the strengths is the way data is separated into tables. A table is a set of data that has certain properties, but essentially, data in a table represents a single business object (or entity). The columns of a table represent the attributes of the object, such as Name, and Phone number for a Customer. Each row is a single instance of the object. Defining the tables needed for a business application is the key goal of this chapter (and the next). A project often begins with a collection of forms and reports that the users need. You need to identify the primary objects on those forms and reports. Business objects typically including things such as Customers, Employees, Items, and Vendors. Other objects arise because of events, such as Sales and Purchases. More complex objects arise from repeating sections on forms (subforms) and to link tables together.

A critical feature of a table is that it must have a primary key—which is a column or set of columns that uniquely identify each row. In base cases, an ID value can be generated by the DBMS. For example, a CustomerID column often contains generated values for Customers to ensure the ID is unique. But, avoid using generated keys for every table. Some tables need multiple columns as part of the key. These columns represent many-to-many relationships. The common business example is SaleItems—a table that lists items that were purchased on a given sale. It contains two columns as part of the key: SaleID + ItemID, because each Sale can contain many Items being sold, and each ItemID can be sold many times. When trying to decide which columns should be part of the primary key, write them down and ask: For each of the other columns (Sale), can there be one or many of these items? If the answer is "many," then set the new column as key.

User views of data.



Conceptual data model.

Class diagram that shows business entities, relationships, and rules.

Patient(PatientID, LastName, FirstName, DateOfBirth, ...)
Visit(VisitID, PatientID, VisitDate, InsuranceCompany, ...)
PatientDiagnoses(VisitID, ICD9Diagnosis, Comments)
VisitProcedures(VisitID, ICD9Procedure, EmployeeID, AmountCharged)
ICD9DiagnosisCodes(ICD9Diagnosis, ShortDescription)
ICD9ProcedureCodes(ICD9Procedure, ShortDescription)
Employee(EmployeeID, LastName, FirstName, EmployeeCategory, ...)
EmployeeCategory(EmployeeCategory)

Implementation (relational) data model.

List of nicely-behaved tables. Use data normalization to derive the list.

Physical data storage.

Indexes and storage methods to improve performance.

Figure 2.1

Design models. The conceptual model records and describes the user views of the system. The implementation model describes the way the data will be stored. The final physical database may utilize storage techniques like indexing to improve performance.

Then turn the question around and ask it again: Can each Item appear on one or many Sales? If the answer is many, add that column as part of the key. Getting the correct primary key is a critical step in designing a relational database.

This chapter focuses on a visual approach to design where each table is defined to represent a single object. Primary keys are defined for each table and highlight one-to-many and many-to-many relationships.

Models

Why are models important in designing systems? Small projects that involve a few users and one or two developers are generally straightforward. However, you still must carefully design the databases so they are flexible enough to handle future needs. Likewise, you have to keep notes so that future developers can easily understand the system, its goals, and your decisions. Large projects bring additional complications. With many users and several developers, you need to split the project into smaller problems, communicate ideas between users and designers, and track the team's progress. Models and diagrams are often used to communicate information among user and developers.

An important step in all development methodologies is to build models of the system. A model is a simplified abstraction of a real-world system. In many cases the model consists of a drawing that provides a visual picture of the system. Just as contractors need blueprints to construct a building, information system developers need designs to help them create useful systems. As shown in Figure 2.1, conceptual models are based on user views of the system. Implementation models are based on the conceptual models and describe how the data will be stored. The implementation model is used by the DBMS to store the data.

Three common types of models are used to design systems: process models, class or object models, and event models. Process models are displayed with a **collaboration diagram** or a data flow diagram (DFD). They are typically explained in detail in systems analysis courses and are used to redesign the flow of

1. Identify the exact goals of the system.
2. Talk with the users to identify the basic forms and reports.
3. Identify the data items to be stored.
4. Design the classes (tables) and relationships.
5. Identify any business constraints.
6. Verify the design matches the business rules.

Figure 2.2

Initial steps in database design. A database design represents the business rules of the organization. You must carefully interview the users to make sure you correctly identify all of the business rules. The process is usually iterative, as you design classes you have to return to the users to obtain more details.

information within an organization. Class diagrams or the older entity-relationship diagrams are used to show the primary entities or objects in the system. Event models such as a sequence or statechart diagram are newer and illustrate the timing of various events and show how messages are passed between various objects. Each of these models is used to illustrate a different aspect of the system being designed. A good designer should be able to create and use all three types of models. However, the class diagrams are the most important tools used in designing and building database applications.

The tools available and the models you choose will depend on the size of the project and the preferences of the organization. This book concentrates on the class diagrams needed for designing tables. You can find descriptions of the other techniques in any good systems analysis book. You can also use the online DBDesign system to help create and analyze the class diagrams used in this book.

Getting Started

How do you begin a database project? Today's DBMS tools are flashy and alluring. It is always tempting to jump right in and start building the forms and reports that users are anxious to see. However, before you can build forms and reports, you must design the database correctly. If you make a mistake in the database design it will be hard to create the forms and reports, and it will take considerable time to change everything later.

Before you try to build anything, determine exactly what data will be needed by talking with the users. Occasionally, the users know exactly what they want. Most times, users have only a rough idea of what they want and a vague perception of what the computer is capable of producing. Communicating with users is a critical step in any development project. The most important aspect is to identify (1) exactly what data to collect, (2) how the various pieces of data are related, and (3) how long each item needs to be stored in the database. Figure 2.2 outlines the initial steps in the design process.

Once you have identified the data elements, you need to organize them properly. The goal is to define classes and their attributes. For example, a Customer is defined in terms of a CustomerID, LastName, FirstName, Phone number and so on. Classes are related to other classes. For example, a Customer participates in a Sale. These relationships also define business rules. For instance, in most cases, a Sale can have only one Customer but a Customer can be involved with many Sales. These business rules ultimately affect the database design. In the example, if more than one customer can participate in a sale, the database design will be different. Hence, the entire point of database design is to identify and formalize the business rules.

To build business applications, you must understand the business details. The task is difficult, but not impossible and almost always interesting. Although every business is different, many common problems exist in the business world. Several of these problems are presented throughout this book. The patterns you develop in these exercises can be applied and extended to many common business problems.

Designing Databases

How do you know what data to put in the database? A database system has to reflect the rules and practices of the organization. You need to talk with users and examine the business practices to identify the rules. And, you need a way to record these rules so you can verify them and share them with other developers. System designs are models that are used to facilitate this communication and teamwork. Designs are a simplification or picture of the underlying business operations.

Identifying User Requirements

One challenging aspect of designing a system is to determine the requirements. You must thoroughly understand the business needs before you can create a useful system. A key step is to interview users and observe the operations of the firm. Although this step sounds easy, it can be difficult—especially when users disagree with each other. Even in the best circumstances, communication can be difficult. Excellent communication skills and experience are important to becoming a good designer.

As long as you collect the data and organize it carefully, the DBMS makes it easy to create and modify reports. As you talk with users, you will collect user documents, such as reports and forms. These documents provide information about the basic data and operations of the firm. You need to gather four basic pieces of information for the initial design: (1) the data that needs to be collected, (2) the data type (domain), (3) the amount of data involved, and (4) rules about the object relationships.

Business Objects

Database design focuses on identifying the data that needs to be stored. Later, queries can be created to search the data, input forms to enter new data, and reports to retrieve and display the data to match the user needs. For now, the most important step is to organize the data correctly so that the database system can handle it efficiently.

All businesses deal with entities or objects, such as customers, products, employees, and sales. From a systems perspective, an **entity** is some item in the real world that you wish to track. That entity is described by its **attributes** or **properties**. For example, a customer entity has a name, address, and phone number. In modeling terms, an entity listed with its properties is called a **class**. In a programming environment, a class can also have **methods** or functions that it can perform, and these can be listed with the class. For example, the customer class might have a method to add a new customer. Database designs seldom need to describe methods, so they are generally not listed.

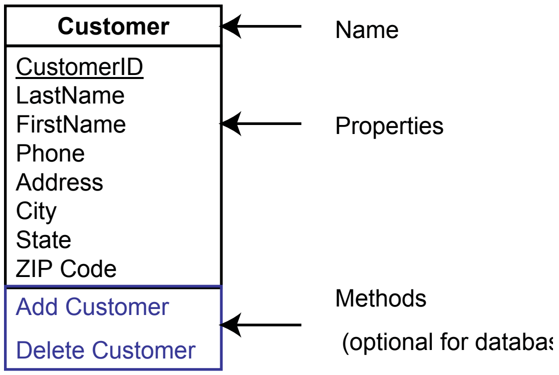


Figure 2.3

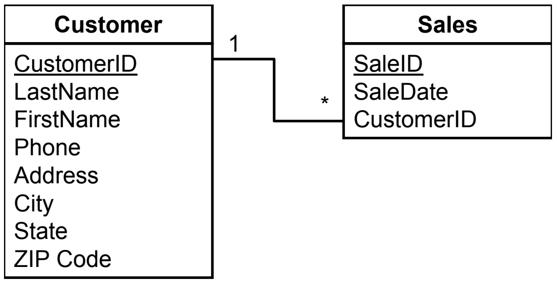
Class. A class has a name, properties, and methods. The properties describe the class and represent data to be collected. The methods are actions the class can perform, and are seldom used in a database design.

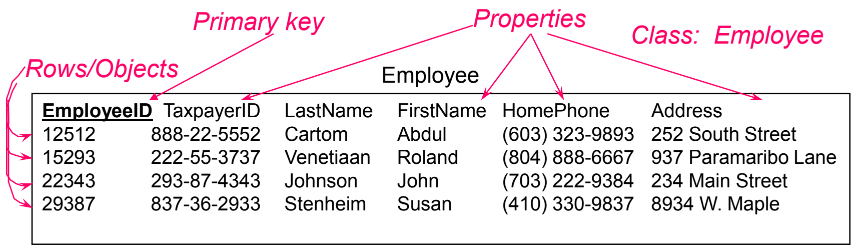
Database designers need some way to keep notes and show the list of classes to users and other designers. Several graphical techniques have been developed, but the more modern approach (and easiest to read) is the class diagram. A **class diagram** displays each class as a box containing the list of properties for the class. Class diagrams also show how the classes are related to each other by connecting them with lines. Figure 2.3 shows how a single class is displayed.

When drawing a class diagram, you often begin by identifying the major classes or entities. As you create a class, you enter the attributes that define this object. These attributes represent the data that the organization needs to store. In the Customer example, you will always need the customer name, and probably an address and phone number. Some organizations also might specify a type of customer (government, business, individual, or something else).

Figure 2.4

Relationships. The Sales tables needs CustomerID to reveal which customer participated in the sale. Putting the rest of the customer data into the Sales table would waste space and cause other problems. The relationship link to the Customer table enables the database system to find all of the related data based on just the CustomerID value.





EmployeeID	TaxpayerID	LastName	FirstName	HomePhone	Address
12512	888-22-5552	Cartom	Abdul	(603) 323-9893	252 South Street
15293	222-55-3737	Venetiaan	Roland	(804) 888-6667	937 Paramaribo Lane
22343	293-87-4343	Johnson	John	(703) 222-9384	234 Main Street
29387	837-36-2933	Stenheim	Susan	(410) 330-9837	8934 W. Maple

Figure 2.5

Basic database definitions. Codd has more formal terms and mathematical definitions, but these are easier to understand. One row in the data table represents a single object, a specific employee in this situation. Each column (attribute) contains one piece of data about that employee.

Tables and Relationships

Classes will eventually be stored as tables in the database system. You have to be careful what columns you include in each table. Chapter 3 describes specific rules in detail, but they also apply when you create the class diagram. One of the most important aspects is to avoid unnecessary duplication of data. Figure 2.4 shows a simple example. The Sales table needs to identify which customer participated in a sale. It accomplishes this task by storing just the primary key CustomerID in the Sales table. An alternative would be to store all of the Customer attributes in the Sales table. However, it would be a waste of space to repeat all of the customer data every time you sell something to a customer. Instead, you create a CustomerID primary key in the Customer table and place only this key value into the Sales table. The database system can then retrieve all of the related customer data from the Customer table based on the value of the CustomerID.

Notice the 1 and the * annotations in the diagram. These represent the business rules. Most companies have a policy that only one (1) customer can be listed on a sale, but a customer can participate in many (*) different sales.

Definitions

To learn how to create databases that are useful and efficient, you need to understand some basic definitions. The main ones are shown in Figure 2.5. Codd created formal mathematical definitions of these terms when he defined relational databases and these formal definitions are presented in the Appendix to Chapter 3. However, for designing and building business applications, the definitions presented here are easier to understand.

A **relational database** is a collection of carefully defined tables organized for a common purpose. A **table** is a collection of columns (attributes or properties) that describe an entity. Individual objects are stored as rows (tuples in Codd's terms) within the table. For example, EmployeeID 12512 represents one instance of an employee and is stored as one row in the Employee table. An attribute (property) is a characteristic or descriptor of an entity. Two important aspects to a relational database are that (1) all data must be stored in tables and (2) all tables must be carefully defined to provide flexibility and minimize problems. **Data normalization** is the process of defining tables properly to provide flexibility, minimize redundancy, and ensure data integrity. The goal of database design and data normalization is to produce a list of nicely behaved tables. Each table describes a single type of object in the organization.

Primary Key

Every table must have a primary key. The **primary key** is a column or set of columns that identifies a particular row. For example, in the customer table you might use customer name to find a particular entry. But that column does not make a good key. What if eight customers are named John Smith? In many cases you will create new key columns to ensure they are unique. For example, a customer identification number is often created to ensure that all customers are correctly separated. The relationship between the primary key and the rest of the data is one-to-one. That is, each entry for a key points to exactly one customer row. To highlight the primary key, the names of the columns that make up the key will be underlined. The DBDesign system uses a star in front of primary key column names because it is easier to see. You can use either approach (or both) if you draw class diagrams by hand.

In some cases there will be several choices to use as a primary key. In the customer example you could choose name or phone number, or create a unique CustomerID. If you have a choice, the primary key should be the smallest set of columns needed to form a unique identifier.

Some U.S. organizations might be tempted to use Social Security numbers (SSN) as the primary key. Even if you have a need to collect the SSN, you will be better off using a separate number as a key. One reason is that a primary key must always be unique, and with the SSN you run a risk that someone might present a forged document. More important, primary keys are used and displayed in many places within a database. If you use the SSN, too many employees will have access to your customers' private information. Because SSNs are used for many financial, governmental, and health records, you should protect customer privacy by limiting employee access to these numbers. In fact, you should encrypt them to prevent unauthorized or accidental release of the data.

The most important issue with a primary key is that it can never point to more than one row or object in the database. For example, assume you are building a database for the human resource management department. The manager tells you that the company uses names of employees to identify them. You ask whether or not two employees have the same name, so the manager examines the list of employees and reports that no duplicates exist among the 30 employees. The manager also suggests that if you include the employee's middle initial, you should never have a problem identifying the employees. So far, it sounds like name might be a potential key. But wait! You really need to ask what the possible key values might be in the future. If you build a database with employee name as a primary key, you are explicitly stating that no two employees will *ever* have the same name. That assumption is almost guaranteed to cause problems in the future. It is far safer to use the database to generate a key number—your application can always provide the ability to search by name, but internally, the DBMS will not mix up two people with the same name.

Class Diagrams: Introduction

What is a class diagram (or entity-relationship diagram)? The DBMS approach focuses on the data. In many organizations data remains relatively stable. For example, companies collect the same basic data on customers today that they

Term	Definition	Pet Store Examples
Entity	Something in the real world that you wish to describe or track.	Customer, Merchandise, Sales
Class	Description of an entity that includes its attributes (properties) and behavior (methods).	Customer, Merchandise, Sale
Object	One instance of a class with specific data.	Joe Jones, Premium Cat Food, Sale #32
Property	A characteristic or descriptor of a class or entity.	LastName, Description, SaleDate
Method	A function that is performed by the class.	AddCustomer, UpdateInventory, ComputeTotal
Association	A relationship between two or more classes.	Each sale can have only one customer

Figure 2.6

Basic definitions. These terms describe the main concepts needed to create a class diagram. The first step is to identify the business entities and their properties. Methods are less important than properties in a database context, but you should identify important functions or calculations.

collected 20 or 30 years ago. Basic items such as name, address, and phone number are always needed. Although you might choose to collect additional data today (cell phone number and e-mail address for example), you still utilize the same base data. On the other hand, the way companies accept and process sales orders has changed over time, so forms and reports are constantly being modified. The database approach takes advantage of this difference by focusing on defining the data correctly. Then the DBMS makes it easy to change reports and forms. The first step in any design is to identify the things or entities that you wish to observe and track.

Classes and Entities

Figure 2.6 shows some examples of the entities and relationships that will exist in the Pet Store database. Note that these definitions are informal. Each entry has a more formal definition in terms of Codd's relational model and precise semantic definitions in the **Unified Modeling Language (UML)**. However, you can develop a database without learning the mathematical foundations.

A tricky problem with database design is that your specific solution depends on the underlying assumptions and business rules. The design process becomes easier as you learn the common business rules. But, any business can have different rules, so you always have to verify the assumptions. For example, consider an employee. The employee is clearly a separate entity because you always need to keep detailed data about the employee (date hired, name, address, and so on). But what about the employee's spouse? Is the spouse an attribute of the Employee entity, or should he or she be treated as a separate entity? If the organization only cares about the spouse's name, it can be stored as an attribute of the Employee

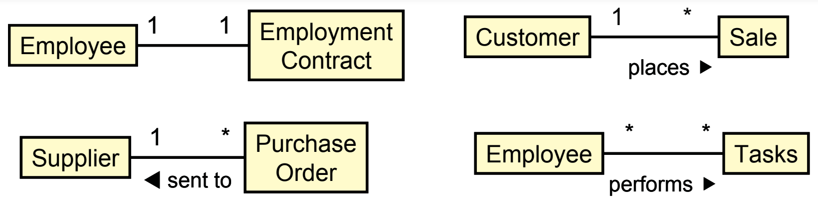


Figure 2.7

Associations. Three types of relationships (one-to-one, one-to-many, and many-to-many) occur among entities. They can be drawn in different ways, but they represent business or organizational rules. Avoid vague definitions where almost any relationship could be classified as many-to-many. They make the database design more complex.

entity. On the other hand, if the organization wants to keep additional information about the spouse (e.g., birthday, occupation, or health records), it might be better to create a separate Spouse entity with its own attributes. Your first step in designing a database is to identify the entities and their defining attributes. The second step is to specify the relationships among these entities.

Associations and Relationships

An important step in designing databases is identifying associations or relationships among entities. Details about these relationships represent the business rules. **Associations** or **relationships** represent business rules. For example, it is clear that a customer can place many orders. But the relationship is not as clear from the other direction. How many customers can be involved with one particular order? Many businesses would say that each order could come from only one customer. Hence there would be a one-to-many relationship between customers and orders. On the other hand, some organizations (such as home sales) might have multiple customers on one order, which creates a many-to-many relationship.

Associations can be named: UML refers to the **association role**. Each end of a binary association may be labeled. It is often useful to include a direction arrow to indicate how the label should be read. Figure 2.7 shows how to indicate that one customer places many sales orders.

UML uses numbers and asterisks to indicate the **multiplicity** in an association. As shown in Figure 2.7, the asterisk (*) represents many. So each supplier can receive many purchase orders, but each purchase order goes to only one supplier. Some older entity-relationship design methods used multiple arrowheads or the letters M and N to represent the “many” sides of a relationship. Correctly identifying relationships is important in properly designing a database application.

Class Diagram Details

A class diagram is a visual model of the classes and associations in an organization. These diagrams have many options, but the basic features that must be included are the class names (entities) in boxes and the associations (relationships) connecting them. Typically, you will want to include more information about the classes and associations. For example, you will eventually include the properties of the classes within the box.

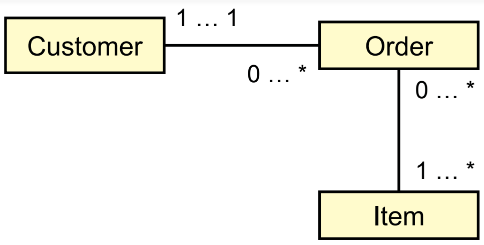


Figure 2.8

Class diagram or entity-relationship diagram. Each customer can place zero or many orders. Each sale must come from at least one and no more than one customer. The zero (0) represents an optional item, so a customer might not have placed any orders yet.

Associations also have several options. One of the most important database design issues is the multiplicity of the relationship, which has two aspects: (1) the maximum number of objects that can be related, and (2) the minimum number of objects, if any, that must be included. As indicated in Figure 2.8, multiplicity is shown as a number for the minimum value, ellipses (…), and the maximum value. An asterisk (*) represents an unknown quantity of "many." In the example in Figure 2.8, exactly one customer (1…1) can be involved with any sale.

Most of the time, a relationship requires that the referenced entity must be guaranteed to exist. For example, what happens if you have a sale form that lists a customer (CustomerID = 1123), but there is no data in the Customer table for that customer? There is a referential relationship between the sales order and the customer entity. Business rules require that customer data must already exist before that customer can make a purchase. This relationship can be denoted by specifying the minimum value of the relationship (0 if it is optional, 1 if it is required). In the Customer-Sales example, the annotation on the Customer would be 1…1 to indicate that a CustomerID value in the Sales table points to exactly one customer (no less than one and no more than one).

Be sure to read relationships in both directions. For example, in Figure 2.8, the second part of the customer/sales association states that a customer can place from zero to many sales orders. That is, a customer is not required to place an order. Some might argue that if a person has not yet placed a sale, that person should not be considered a customer. But that interpretation is getting too picky, and it would cause chicken-and-the-egg problems if you tried to enforce such a rule. Consider the Customer table to include potential customers who have signed up but not purchased anything yet.

Moving down the diagram, note the many-to-many relationship between Sale and Item (asterisks on the right side for both classes). A sale must contain at least one item (empty sales orders are not useful in business), but the firm might have an item that has not been sold yet.

Quick Start

Is there an easier way to get started with database design? In the end, details are important in designs. But often it helps to focus on the bigger picture first and fill in the details later. The sections after this one examine some common patterns

1. Identify the primary classes and data elements.
2. Create the easy classes.
3. Create generated keys if necessary.
4. Add tables to split many-to-many relationships.
5. Check primary keys.
6. Verify relationships.
7. Verify data types.

Figure 2.9

Steps to create a class diagram. Primary keys often cause problems. Look for many-to-many relationships and split them into new tables.

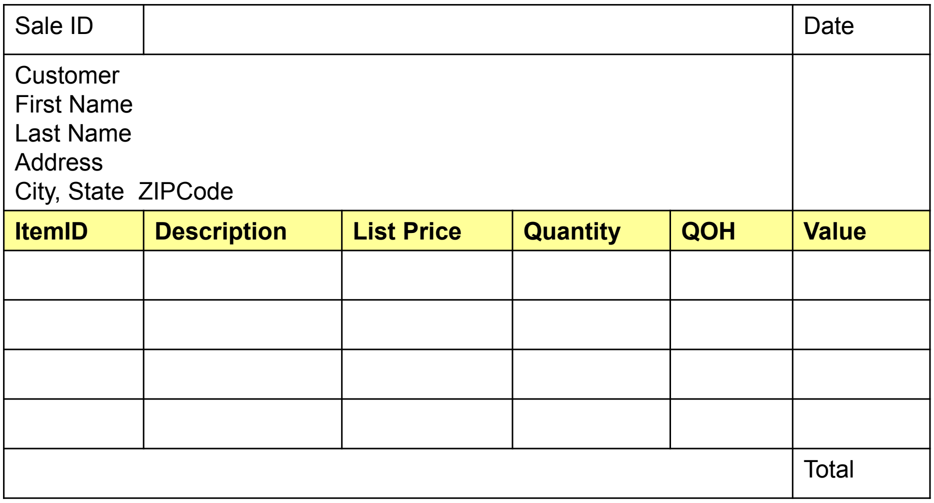
that arise in designing business databases, but it is easy to get lost in the details. Eventually, you need to understand those details, but first you need to start thinking in terms of a few basic concepts. The purpose of this section is to show you some ways to begin learning database design.

Creating a Class Diagram

This section summarizes how you begin a class diagram and highlights the issue of primary keys. Figure 2.9 outlines the major steps. The real trick is to start with the easy classes. Look for the base entities that do not depend on other classes. For instance, most business applications have relatively simple classes for Customers, Employees, and Items. These tables often use a generated key as the primary key column, and the data elements are usually obvious. A generated key is one that is created by the DBMS and guaranteed to be unique within that table.

Figure 2.10

Basic Sales form. Look through the form and see if you can identify the basic business objects. You should be able to easily find three objects.



Sale ID					Date
Customer First Name Last Name Address City, State ZIPCode					
ItemID	**Description**	**List Price**	**Quantity**	**QOH**	**Value**
					Total

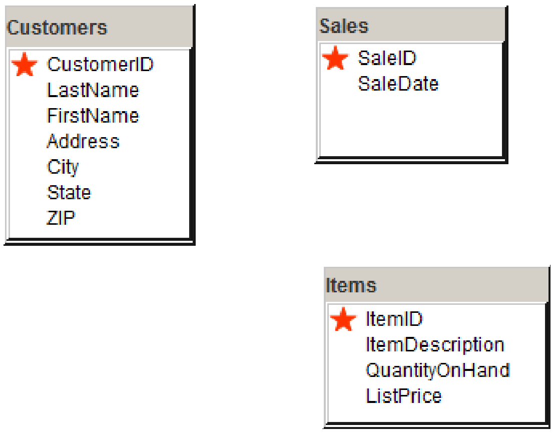


Figure 2.11

Initial objects for the Sales form: Customers, Items, and Orders. Each has a single generated column as the primary key because rows are created independently. Notice that they are not yet related to each other.

Consider the basic Sales form. Figure 2.10 shows a simplified version. Look over the form and see if you can identify the main objects it contains. Think about standard business sales for a second and you should be able to identify at least three common objects contained on the form: Customers, Items, and Sales. Data for the Customer object seems to be displayed in its own section of the form so that is a useful indicator, but people are often represented as a separate object so you should become comfortable with creating that table. The Items are a little trickier to see, but think about the business transaction and you can see the importance of treating merchandise items as a separate object. The Sales object is slightly trickier because it records an event. But in most cases, the overall form itself will become a table—because it ties together all of the other objects. In this case, Sales represent the integration of Customers and Items.

Each of these three tables (Customers, Items, and Sales) can stand alone as a base table. Customers are clearly defined in terms of standard properties such as name and address. Items have properties such as a description and list price. Sales take place on a specified date. Every table needs a primary key, but each of these three tables can benefit from using a generated key. Some companies might rely on the marketing department to create unique CustomerID and ItemID values, but database systems are good at creating unique numbers, so it is far easier to let the DBMS generate ID values as new customers, items, and sales are entered. Figure 2.11 shows these three tables in DBDesign, but you could also draw them by hand or write them with the main columns.

Notice that these three tables are not yet connected to each other. Eventually, to be able to recreate the Sales form, all of the tables must be related or connected somehow. But it is worth examining exactly how these tables might be connected. Begin by focusing on the Customers and Sales tables. Three possibilities exist: 1. Put the SaleID into the Customers table, 2. Put the CustomerID into the Sales table, or 3. Put both ID values into a third table. Before trying to find the answer, understand that each of those three possibilities could be correct—the answer depends on the specific business rules. That is, each possibility represents a different set of business rules.

CustomerID SaleID

Each customer can place many Sales (key SaleID).
Each order comes from one customer (do not key CustomerID).

*SaleID
CustomerID

Figure 2.12

Identifying primary keys. Write down the potential key columns. Ask if each of the first entity (Customer) can have one or many of the second entity (Sale). If the answer is many, key the second item. Reverse the process to see if one of the second items can be associated with one or many of the first items.

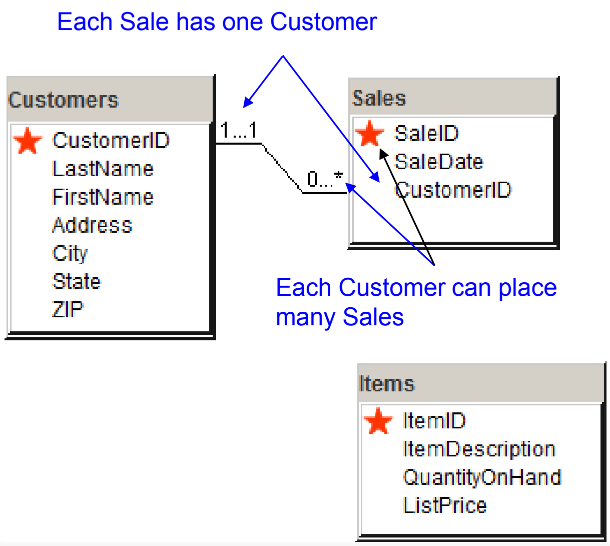
Primary Keys and Relationships

Primary keys and many-to-many relationships are often difficult for students. The trick is to remember that any column that is part of the primary key represents a "many" relationship. Consider the classic Customers-Sales relationship.

When you are not certain how to identify the keys in a table, Figure 2.12 shows a process for identifying the class relationship. Write down the columns you want to study with no key indicators. Ask yourself if each customer (first column) can place one or many Sales (second column). If the answer is many Sales, add a key indicator (underline) to the SaleID. Reverse the process and ask if a specific Sale can come from one Customer or many. The standard business rule says only one

Figure 2.13

Relationship between Customers and Sales. CustomerID belongs in the Sales table but CustomerID is not part of the key. Each Sale has one Customer—so CustomerID is not keyed in the Sales table. Each Customer can have multiple Sales. Read keyed columns as "many" and non-keyed columns as "one."



SaleID ItemID

Each Sale can have many Items (key ItemID).
Each Item can be sold many times (key SaleID).

Need a table with both SaleID and ItemID as keys

*SaleID
*ItemID

Figure 2.14

Identifying primary keys. Write down SaleID and ItemID and identify the associations. The repeating section in the original sales form shows that a Sale can list many Items. When Items are represented as SKUs (such as cans of dog food), the item can be sold many times. So a table is needed with both SaleID and ItemID as keys. Because this table does not yet exist, it must be created: SaleItems.

customer is responsible for a sale, so do not key CustomerID. The result says that SaleID is keyed but CustomerID is not. So you need to put CustomerID into a table with only SaleID as the key column. That would be the Sales table.

If you had put the SaleID into the Customers table the relationships would be reversed. With CustomerID keyed but not SaleID the design would be saying that each customer can never participate in more than one sale, and every sale could involve many customers. Read the keyed column as "many" and non-keyed columns as "one." Figure 2.13 shows the resulting design by placing CustomerID into the Sales table where it is not part of the key. From a more mechanical perspective, CustomerID could never be keyed in the Sales table. The SaleID is already a generated key so it is guaranteed to be unique. No other column would ever need to be keyed in that table. Generated key columns always stand alone.

The design is closer, but notice that the Items table is still not connected to any of the others. Thinking about the business associations, it seems that Items and Sales should be related somehow—because the Items are shown on the original Sales Form. So try the same key process. Figure 2.14 shows the process. Write down SaleID and ItemID and ask yourself if a Sale can contain one or many Items. If you are uncertain, look back at the original sales form and notice the repeating section that can list many items, so the answer is Many. Mark ItemID as a key. Working the other direction: Can an Item be sold more than once? If the items are standardized, such as cans of dog food, the answer is also "many" times. Yes, a given, specific can is sold only once, but that particular ItemID representing a brand and flavor of dog food can be sold many times. So also key SaleID.

Hence, you need a table that contains both SaleID and ItemID as a key. Look at your work so far and you will see that no such table exists. So create a new table that contains those two keys. Figure 2.15 shows the resulting table and how it ties the Sales and Items tables together. Many-to-many relationships are always handled with this third table. In DBDesign, note that a blue star is used to represent the keys in the SaleItems table. Remember that a filled red star is only used for generated keys—in the table where the values are created. So a new SaleID value will be created when a Sale is added to the Sale table, and that value will be inserted into the SaleItems table. Similarly, an existing ItemID from the Items table will be inserted into the SaleItems table to indicate which item is being sold.

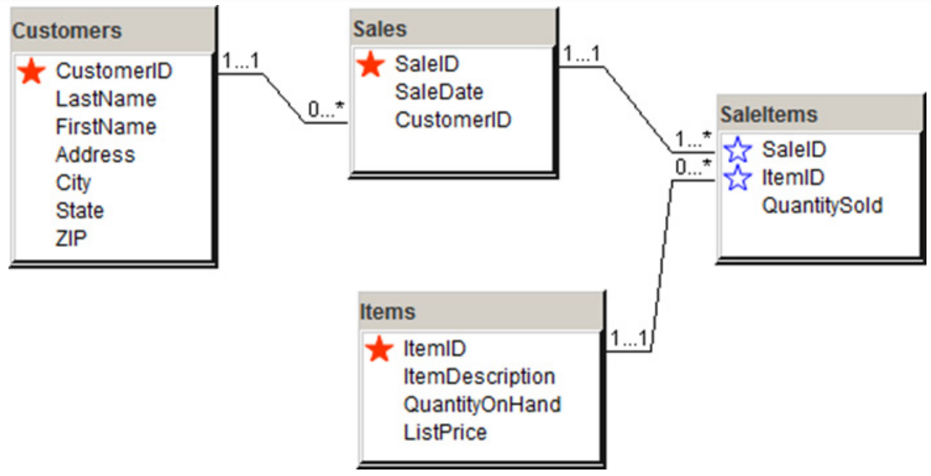


Figure 2.15

Two columns as part of the primary key. Sales and Items have a many-to-many relationship which is handled as a new table (SaleItems) with both of the columns keyed.

Think of it as a bar code scanner that reads the existing ItemID and inserts it into a new row of the SaleItems table.

DBDesign uses a special symbol to show where keys are generated to remind you that: (1) In a generating table, the generated key can be the only key column, (2) A generated key can be generated only once, and (3) You can never have a relationship that ties two generated keys together (because it would never make sense to link two randomly generated numbers).

Getting the primary keys right is critical at this stage of the design. In many ways, the keys identify the objects and tables. In the example, the generated keys CustomerID, ItemID, and SaleID uniquely identify each related object. The composite key: SaleID+ItemID identifies the many-to-many relationship between Sales and Items. From this point, you then assign the other columns as properties of the correct objects. For instance, Last Name, First Name, and Address are attributes of Customers. ListPrice is an attribute of Items because each item has one list price, if we do not worry about changes over longer periods of time. And QuantitySold is an attribute of the SaleItems because it represents the amount of a specific Item on a given Sale.

Class Diagrams: Details

How are some common business associations handled in class diagrams? Class diagrams are useful to visualize the business entities and the underlying relationships. Many business entities can be represented by simple classes (Customers, Employees, Merchandise, Sales, and so on). However, some common business problems can lead to relatively complex relationships. Many of these situations are tied to events, such as sales or as-

Note: It is possible to temporarily skip this section and return to it once the student is more familiar with the basic design issues.

sembly. A couple of tricky concepts evolved from object-oriented design require special handling in class diagrams, and are trickier to handle within relational databases. Two classic situations are composition (objects built from other objects), and inheritance (objects defined as extensions of parent objects). This section examines how to diagram these relatively complex topics.

Association Details: N-ary Associations

Many-to-many associations between classes cause problems in the database design. They are acceptable in an initial diagram such as Figure 2.16, but they will eventually have to be split into one-to-many relationships. This process is explained in detailed in Chapter 3.

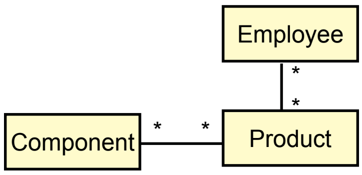
In a related situation, as shown in Figure 2.16, entities are not always obvious. Consider a basic manufacturing situation in which employees assemble components into final products. At first glance, it is tempting to say that there are three entities: employees, components, and products. This design specifies that the database should keep track of which employees worked on each product and which components go into each product. Notice that two many-to-many relationships exist.

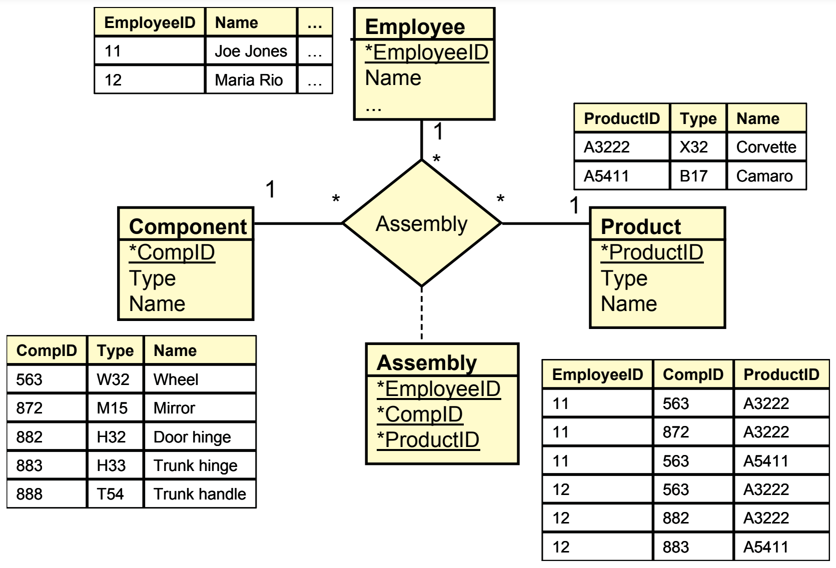
To understand the problem caused by the many-to-many relationships, consider what happens if the company wants to know which employees assembled each component into a product. To handle this situation, Figure 2.17 shows that the three main entities (Employee, Product, and Component) are actually related to each other through an Assembly association. When more than two classes are related, the relationship is called an **n-ary association** and is drawn as a diamond. This association (actually any association) can be described by its own class data. In this example an entry in the assembly list would contain an EmployeeID, a ComponentID, and a ProductID. In total, many employees can work on many products, and many components can be installed in many products. Each individual event is captured by the Assembly association class. The Assembly association solves the many-to-many problem, because a given row in the Assembly class holds data for one employee, one component, and one product. Ultimately, you would also include a Date/Time column to record when each event occurred.

According to the UML standard, multiplicity has little meaning in the n-ary context. The multiplicity number placed on a class represents the potential number of objects in the association when the other n-1 values are fixed. For example, if ComponentID and EmployeeID are fixed, how many products could there be? In other words, can an employee install the same component in more than one

Figure 2.16

Many-to-many relationships cause problems for databases. In this example, many employees can install many components on many products, but we do not know which components the employee actually installed.





EmployeeID	Name	…
11	Joe Jones	…
12	Maria Rio	…

ProductID	Type	Name
A3222	X32	Corvette
A5411	B17	Camaro

CompID	Type	Name
563	W32	Wheel
872	M15	Mirror
882	H32	Door hinge
883	H33	Trunk hinge
888	T54	Trunk handle

EmployeeID	CompID	ProductID
11	563	A3222
11	872	A3222
11	563	A5411
12	563	A3222
12	882	A3222
12	883	A5411

Figure 2.17

Many-to-many associations are converted to a set of one-to-many relationships with an n-ary association, which includes a new class. In this example each row in the Assembly class holds data for one employee, one component, and one product. Notice that the Assembly class (box) is connected to the Assembly association (diamond) by a dashed line.

product? In most situations the answer will be yes, so the multiplicity will generally be a "many" asterisk.

Eventually to create a database, all many-to-many relationships must be converted to a set of one-to-many relationships by adding a new entity. Like the Assembly entity, this new entity usually represents an activity and often includes a date/time stamp.

As a designer you will use class diagrams for different purposes. Sometimes you need to see the detail; other times you only care about the big picture. For large projects, it sometimes helps to create an overview diagram that displays the primary relationships between the main classes. On this diagram it is acceptable to use many-to-many relationships to hide some detail entities.

Association Details: Aggregation

Some special types of associations arise often enough that UML has defined special techniques for handling them. One category is known as an **aggregation** or a collection. For example, a Sale consists of a collection of Items being purchased. As shown in Figure 2.18, aggregation is indicated by a small diamond on the association line next to the class that is the aggregate. In the example, the diamond is next to the Sale class. Associations with a many side can be ordered or unordered. In this example, the sequence in which the Items are stored does not matter. If order did matter, you would simply put the notation {ordered} underneath the association. Be sure to include the braces around the word. Aggregations are rarely marked separately in a database design.

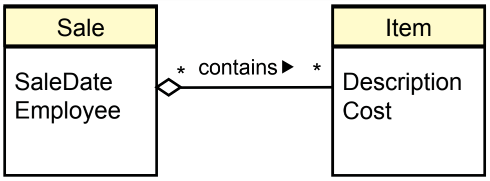


Figure 2.18

Association aggregation. A Sale contains a list of items being purchased. A small diamond is placed on the association to remind us of this special relationship.

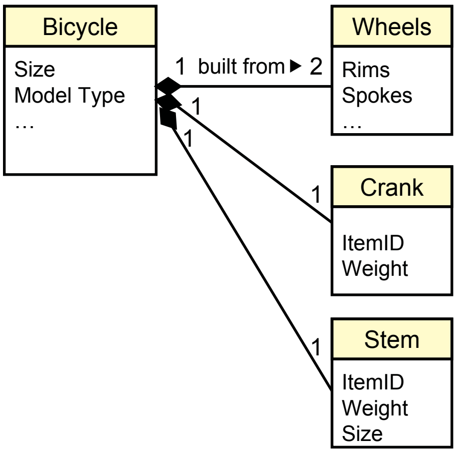
Association Details: Composition

The simple aggregation indicator is not used much in business settings. However, **composition** is a stronger aggregate association that does arise more often. In a composition, the individual items become the new object. Consider a bicycle, which is built from a set of components (wheels, crank, stem, and so on). UML provides two methods to display composition. In Figure 2.19 the individual classes are separated and marked with a filled diamond. An alternative technique shown in Figure 2.20 is to indicate the composition by drawing the component classes inside the main Bicycle class. It is easier to recognize the relationship in the embedded diagram, but it could get messy trying to show 20 different objects required to define a bicycle. Figure 2.20 also highlights the fact that the component items could be described as properties of the main Bicycle class.

The differences between aggregation and composition are subtle. The UML standard states that a composition can exist only for a one-to-many relationship. Any many-to-many association would have to use the simple aggregation indicator. Composition relationships are generally easier to recognize than aggregation relationships, and they are particularly common in manufacturing environments.

Figure 2.19

Association composition. A bicycle is built from several individual components. These components no longer exist separately; they become the bicycle.



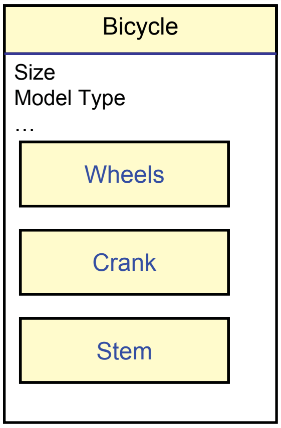


Figure 2.20

Association composition. It is easier to see the composition by embedding the component items within the main class.

Just remember that a composition exists only when the individual items become the new class. After the bicycle is built, you no longer refer to the individual components.

Association Details: Generalization

Another common association that arises in business settings is **generalization**. This situation generates a class hierarchy. The most general description is given at the top, and more specific classes are derived from it. Figure 2.21 presents a sample from Sally's Pet Store. Each animal has certain generic properties (e.g., DateBorn, Name, Gender, ListPrice), contained in the generic Animal class. But specific types of animals require slightly different information. For example, for a mammal (perhaps a cat), buyers want to know the size of the litter and whether or not the animal has claws. On the other hand, fish do not have claws, and customers want different information, such as whether they are fresh- or saltwater fish and the condition of their scales. Similar animal-specific data can be collected for each species. There can be multiple levels of generalization. In the pet store example, the Mammal category could be further split into Cat, Dog, and Other.

A small, unfilled triangle is used to indicate a generalization relationship. You can connect all of the subclasses into one triangle as in Figure 2.21, or you can draw each line separately. For the situation in this example, the collected approach is the best choice because the association represents a disjoint (mutually exclusive) set. An animal can fall into only one of the subclasses.

An important characteristic of generalization is that lower-level classes inherit the properties and methods of the classes above them. Classes often begin with fairly general descriptions. More detailed classes are **derived** from these base classes. Each lower-level class inherits the properties and functions from the higher classes. **Inheritance** means that objects in the derived classes include all of the properties from the higher classes, as well as those defined in their own class. Similarly, functions defined in the related classes are available to the new class.

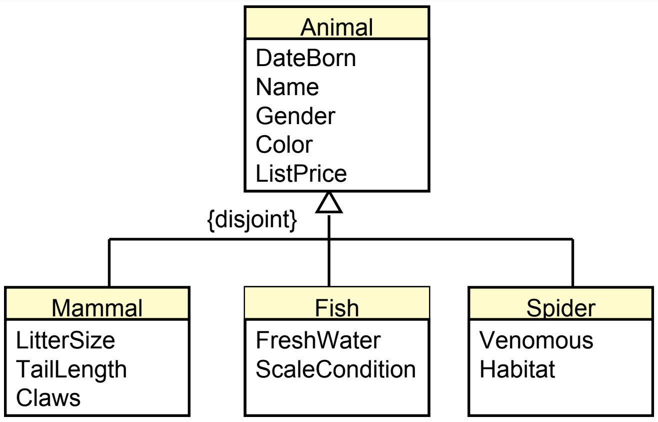


Figure 2.21

Association generalization. The generic Animal class holds data that applies to all animals. The derived subclasses contain data that is specific to each species.

Consider the example of a bank accounting system displayed in Figure 2.22. A designer would start with the basic description of a customer account. The bank is always going to need basic information about its accounts, such as AccountID, CustomerID, DateOpened, and CurrentBalance. Similarly, there will be common functions including opening and closing the account. All of these basic properties and actions will be defined in the base class for Accounts.

New accounts can be derived from these accounts, and designers would only have to add the new features—saving time and reducing errors. For example, Checking Accounts have a MinimumBalance to avoid fees, and the bank must track the number of Overdrafts each month. The Checking Accounts class is derived from the base Accounts class, and the developer adds the new properties and functions. This new class automatically inherits all of the properties and functions from the Accounts class, so you do not have to redefine them. Similarly, the bank pays interest on savings accounts, so a Savings Accounts class is created that records the current InterestRate and includes a function to compute and credit the interest due each month.

Additional classes can be derived from the Savings Accounts and Checking Accounts classes. For instance, the bank probably has special checking accounts for seniors and for students. These new accounts might offer lower fees, different minimum balance requirements, or different interest rates. To accommodate these changes, the design diagram is simply expanded by adding new classes below these initial definitions. These diagrams display the **class hierarchy** which shows how classes are derived from each other, and highlights which properties and functions are inherited. The UML uses open diamond arrowheads to indicate that the higher-level class is the more general class. In the example, the Savings Accounts and Checking Accounts classes are derived from the generic Accounts class, so the association lines point to it.

Each class in Figure 2.22 can also perform individual functions. Defining properties and methods within a class is known as **encapsulation**. It has the advantage of placing all relevant definitions in one location. Encapsulation also provides some security and control features because properties and functions can be protected from other areas of the application.

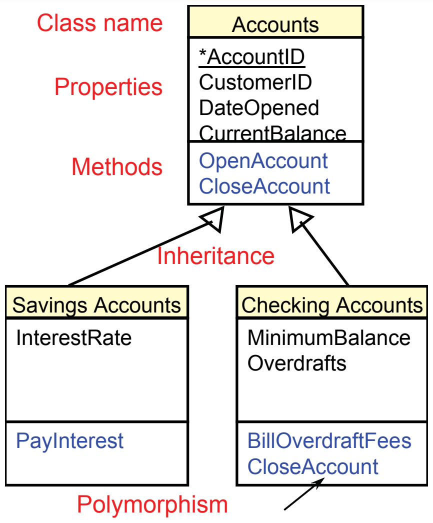


Figure 2.22

Class inheritance. Object classes begin with a base class (e.g., Accounts). Other classes are derived from the base class. They inherit the properties and methods, and add new features. In a bank, all accounts need to track basic customer data. Only checking accounts need to track overdraft fees.

Another interesting feature of encapsulation can be found by noting that the Accounts class has a function to close accounts. Look carefully, and you will see that the Checking Accounts class also has a function to close accounts (CloseAccount). When a derived class defines the same function as a parent class, it is known as **polymorphism**. When the system activates the function, it automatically identifies the object's class and executes the matching function. Designers can also specify that the derived function (CloseAccount in the Checking Accounts class) can call the related function in the base class. In the banking example, the Checking Account's CloseAccount function would cancel outstanding checks, compute current charges, and update the main balance. Then it would call the Accounts CloseAccount function, which would automatically archive the data and remove the object from the current records.

Polymorphism is a useful tool for application builders. It means that you can call one function regardless of the type of data. In the bank example you would simply call the CloseAccount function. Each different account could perform different actions in response to that call, but the application does not care. The complexity of the application has been moved to the design stage (where all of the classes are defined). The application builder does not have to worry about the details.

Note that in complex situations, a subclass can inherit properties and methods from more than one parent class. In Figure 2.23, a car is motorized, and it is designed for on-road use, so it inherits properties from both classes (and from the generic Vehicle class). The bicycle situation is slightly more complex because it could inherit features from the On-Road class or from the Off-Road class, depending on the type of bicycle. If you need to record data about hybrid bicycles, the Bicycle class might have to inherit data from both the On-Road and Off-Road classes.

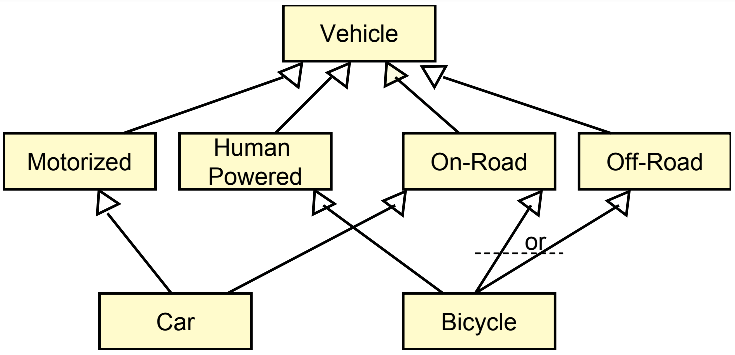


Figure 2.23

Multiple parent classes. Classes can inherit properties from several parent classes. The key is to draw the structure so that users can understand it and make sure that it matches the business rules.

Association Details: Reflexive Association

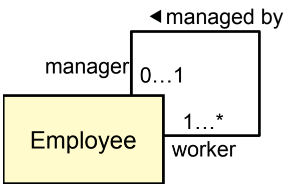
A reflexive relationship is another situation that arises in business that requires special handling. A **reflexive association** is a relationship from one class back to itself. The most common business situation is shown in Figure 2.24. most employees (worker) have a manager. Hence there is an association from Employee (the worker) back to Employee (the manager). Notice how UML enables you to label both ends of the relationship (manager and worker). Also, the "◀managed by" label indicates how the association should be read. The labels and the text clarify the purpose of the association. Associations may not need to be labeled, but reflexive relationships should generally be explained so other developers understand the purpose.

Sally's Pet Store Class Diagram

Are more complex diagrams different? It takes time to learn how to design databases. It is helpful to study other examples. Remember that Sally, the owner of the pet store, wants to create the application in sections. The first section will track the basic transaction data of the store. Hence you need to identify the primary entities involved in operating a pet store.

Figure 2.24

Reflexive relationship. A manager is an employee who manages other workers. Notice how the labels explain the purpose of the relationship.



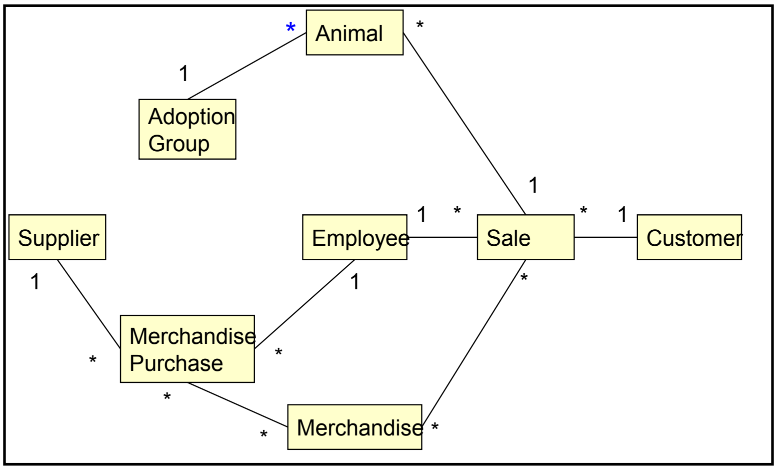


Figure 2.25

Initial class diagram for the PetStore. Animal purchases and sales are tracked separately from merchandise because the store needs to monitor different data for the two entities.

The first step in designing the pet store database application is to talk with the owner (Sally), examine other stores, and identify the primary components that will be needed. After talking with Sally, it becomes clear that the Pet Store has some features that make it different from other retail stores. The most important difference is that the store must track two separate types of sales: animals are handled differently from products. For example, the store tracks more detailed information on each animal. Also, products can be sold in multiple units (e.g., six cans of dog food), but animals must be tracked individually. Figure 2.25 shows an initial class diagram for Sally's Pet Store that is based on these primary entities. The diagram highlights the two separate tracks for animals and merchandise. Note that animals are also adopted instead of sold. Because each animal is unique and is adopted only once, the transfer of the animal is handled differently than the sale of merchandise.

While talking with Sally, a good designer will write down some of the basic items that will be involved in the database. This list consists of entities for which you need to collect data. For example, for the Pet Store database you will clearly need to collect data on customers, suppliers, animals, and products. Likewise, you will need to record each purchase and each sale. Right from the beginning, you will want to identify various attributes or characteristics of these entities. For instance, customers have names, addresses, and phone numbers. For each animal, you will want to know the type of animal (cat, dog, etc.), the breed, the date of birth, and so on.

The detailed class diagram will include the attributes for each of the entities. Notice that the initial diagram in Figure 2.25 includes several many-to-many relationships. All of these require the addition of an intermediate class. Consider the MerchandiseOrder class. Several items can be ordered at one time, so you will create a new entity (OrderItem) that contains a list of items placed on each MerchandiseOrder. The AnimalOrder and Sale entities will gain similar classes.

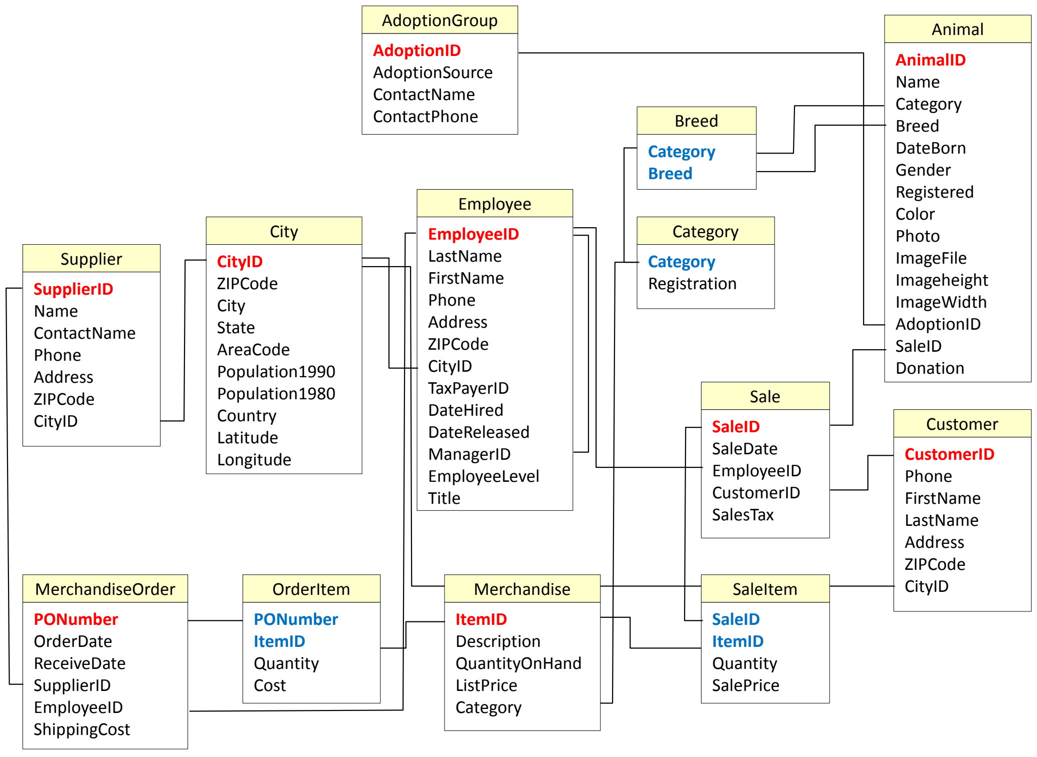


Figure 2.26

Detailed class diagram for the pet store. Notice the tables added to solve many-to-many problems: OrderItem, AnimalOrderItem, SaleItem, and SaleAnimal. The City table was added to reduce data entry. The Breed and Category tables were added to ensure data consistency. Users select the category and breed from these tables, instead of entering text or abbreviations that might be different every time.

Figure 2.26 shows the more detailed class diagram for the Pet Store with these new intermediate classes. It also contains new classes for City, Breed, and Category. Postal codes and cities raise issues in almost every business database. There is a relationship between cities and postal codes, but it is not one-to-one. One simple solution is to store the city, state, and postal code for every single customer and supplier. However, for local customers, it is highly repetitive to enter the name of the city and state for every sale. Clerks end up abbreviating the city entry and every abbreviation is different, making it impossible to analyze sales by city. A solution is to store city and postal code data in a separate class as a lookup table. Commonly used values can be entered initially. An employee can select the desired city from the existing list without having to reenter the data.

The Breed and Category classes are used to ensure consistency in the data. One of the annoying problems of text data is that people rarely enter data consistently. For example, some clerks might abbreviate the Dalmatian dog breed as Dal, others might use Dalma, and a few might enter the entire name. To solve this problem, you want to store all category and breed names one time in separate classes. Then employees simply choose the category and breed from the list in these classes. Hence data is pulled from these lookup-tables and entered exactly the same way every time.

Both the overview and the detail class diagrams for the Pet Store can be used to communicate with users. Through the entities and relationships, the diagram displays the business rules of the firm. For instance, the separate treatment of animals and merchandise is important to the owner. Similarly, capturing only one customer per each sale is an important business rule. This rule should be confirmed by Sally. If a family adopts an animal, does she want to keep track of each member of the family? If so, you would need to add a Family class that lists the family members for each customer. The main point is that you can use the diagrams to display the new system, verify assumptions, and get new ideas.

Data Types (Domains)

What are the different data types? As you list the properties within each class, you should think about the type of data they will hold. Each attribute holds a specific **data type** or data domain. For example, what is an EmployeeID? Is it numeric? At what value does it start? How should it be incremented? Does it contain letters or other alphanumeric characters? You must identify the domain of each attribute or column. Figure 2.27 identifies several common domains. The most common is text, which holds any characters.

Note that any of the domains can also hold missing data. Users do not always know the value of some item, so it may not be entered. Missing data is defined as a **null** value.

Text

Text columns generally have a limited number of characters. SQL Server and Oracle both cut the limit in half for Unicode (2-byte) characters. Microsoft Access is the most limited at 255 characters. Some database management systems ask you to distinguish between fixed-length and variable-length text. Fixed-length strings always take up the amount of space you allocate and are most useful to improve speed in handling short strings like identification numbers or two-letter state abbreviations. Variable-length strings are stored so they take only as much space as needed for each row of data.

Memo or long-text columns are also used to hold large variable-length text data. The difference from variable-length text is that the database can allocate more space for as it is needed. The exact limit depends on the DBMS and the computer used, but long text can often include tens of thousands or millions of characters in one database column. Long text columns are often used for long comments or even short reports. However, some systems limit the operations that you can perform with these columns, such as not allowing you to sort the column data or apply pattern-matching searches.

Numbers

Numeric data is also common, and computers recognize several variations of numeric data. The most important decision you have to make about numeric data columns is choosing between integer and floating-point numbers. Integers cannot hold fractions (values to the right of a decimal point). Integers are often used for counting and include values such as 1; 2; 100; and 5,000. Floating-point numbers can include fractional values and include numbers like 3.14159 and 2.718.

Generic	Access	SQL Server	Oracle
Text			
fixed	NA	char	CHAR
variable	Short Text	varchar	VARCHAR2
Unicode	Short Text	nchar, nvarchar	NVARCHAR2
Long text	Long Text	nvarchar(max)	LONG
XML	NA	XML	XMLType
Number			
Byte (8 bits)	Byte	tinyint	INTEGER
Integer (16 bits)	Integer	smallint	INTEGER
Long (32 bits)	Long	int	INTEGER
(64 bits)	NA	bigint	NUMBER(127,0)
Fixed precision	Decimal	decimal(p,s)	NUMBER(p,s)
Float	Float	real	NUMBER, FLOAT
Double	Double	float	NUMBER
Currency	Currency	money	NUMBER(38,4)
Yes/No	Yes/No	bit	INTEGER
Date/Time	Date/Time	datetime	DATE
		smalldatetime	
Interval	NA	interval year...	INTERVAL YEAR...
Image	OLE Object	varbinary(max)	LONG RAW, BLOB
AutoNumber	AutoNumber	Identity	SEQUENCES
		rowguidcol	ROWID

Figure 2.27

Data types (domains). Common data types and their variations in three database systems. The text types in SQL Server and Oracle beginning with an "N" hold Unicode character sets, particularly useful for non-Latin based languages.

The first question raised with integers and floating-point numbers is, Why should you care? Why not store all numbers as floating-point values? The answer lies in the way that computers store the two types of numbers. In particular, most machines store integers in 2 (or 4) bytes of storage for every value; but they store each floating point number in 4 (or 8) bytes. Although a difference of 2 bytes might seem trivial, it can make a huge difference when multiplied by several billion rows of data. Additionally, arithmetic performed on integers is substantially faster than computations with floating-point data. Something as simple as adding two numbers together can be 10 to 100 times faster with integers than with floating-point numbers. Although machines have become faster and storage costs keep declining, performance is still an important issue when you deal with huge databases and a large customer base. If you can store a number as an integer, do it—you will get a measurable gain in performance.

Most systems also support long integers and double-precision floating-point values. In both cases the storage space is doubled compared to single-precision data. The main issue for designers involves the size of the numbers and precision that users need. For example, if you expect to have 100,000 customers, you cannot use an integer to identify and track customers (a key value). Note that only 65,536 values can be stored as 16-bit integers. To count or measure larger values, you need to use a long integer, which can range between +/- 2,000,000,000. Similarly,

Data Types	**Size**		
	Access	**SQL Server**	**Oracle**
Text (characters)			
fixed		8K, 4K	2 K
variable	255	8K, 4K	4 K
long text	64 KB	2 G, 1G	2 G
XML		2 G	
Numeric			
Byte (8 bits)	255	255	38 digits
Integer (16 bits)	+/- 32767	+/- 32767	38 digits
Long (32 bits)	+/- 2 B	+/- 2 B	38 digits
(64 bits)	NA	18 digits	p: 38 digits
Fixed precision	p: 1-28	p: 1-38	s: -84-127, p: 1-38
Float	+/- 1 E 38	+/- 1 E 38	38 digits
Double	+/- 1 E 308	+/- 1 E 308	38 digits
Currency	+/- 900.0000 tril.	+/- 900.0000 tril.	38 digits
Yes/No	0/1	0/1	
Date/Time	1/1/100 - 12/31/9999 (1 sec)	1/1/1753 - 12/31/9999 (3 ms) 1/1/1900 - 6/6/2079 (1 min)	1/1/-4712 - 1/31/9999 (sec)
Image	1 GB	2 GB	2 GB, 4 GB
AutoNumber	Long (4 B)	4 B or 18 digits with bigint	38 digits max.

Figure 2.28

Data sizes. Make sure that you choose a data type that can hold the largest value you will encounter. Choosing a size too large can waste space and cause slow calculations, but if in doubt, choose a larger size.

floating point numbers can support about six significant digits. Although the magnitude (exponent) can be larger, no more than six or seven digits are maintained. If users need greater precision, use double-precision values, which maintain 14 or more significant digits. Figure 2.28 lists the maximum sizes of the common data types.

Many business databases encounter a different problem. Monetary values often require a large number of digits, and users cannot tolerate round-off errors. Even if you use long integers, you would be restricted to values under 2,000,000,000 (20,000,000 if you need two decimal point values). Double-precision floating-point numbers would enable you to store numbers in the billions even with two decimal values. However, floating-point numbers are often stored with round-off errors, which might upset the accountants whose computations must be accurate to the penny. To compensate for these problems, database systems offer a currency data type, which is stored and computed as integer values (with an imputed decimal point). The arithmetic is fast, large values in the trillions can be stored, and round-off error is minimized. Most systems also offer a generic fixed-precision data type. For example, you could specify that you need 4 decimal digits of precision, and the database will store the data and perform computations with exactly 4 decimal digits.

Dates and Times

All databases need a special data type for dates and times. Most systems combine the two into one domain; some provide two separate definitions. Many beginners try to store dates as string or numeric values. Avoid this temptation. Date types have important properties. Dates (and times) are actually stored as single numbers. Dates are typically stored as integers that count the number of days or seconds from some base date. This base date may vary between systems, but it is only used internally. The value of storing dates by a count is that the system can automatically perform date arithmetic. You can easily ask for the number of days between two dates, or you can ask the system to find the date that is 30 days from today. Even if that day is in a different month or a different year, the proper date is automatically computed. Although most systems need 8 bytes to store date/time columns, doing so removes the need to worry about any year conversion problems.

A second important reason to use internal date and time representations is that the database system can convert the internal format to and from any common format. For example, in European nations, dates are generally displayed in day/month/year format, not the month/day/year format commonly used in the United States. With a common internal representation, users can choose their preferred method of entering or viewing dates. The DBMS automatically converts to the internal format, so internal dates are always consistent.

Databases also need the ability to store time intervals. Common examples include a column to hold years, months, days, minutes, or even seconds. For instance, you might want to store the length of time it takes an employee to perform a task. Without a specific interval data type, you could store it as a number. However, you would have to document the meaning of the number—it might be hours, minutes, or seconds. With a specified interval type, there is less chance for confusion.

Binary Objects

A different type of domain is a category for objects or **binary large object (BLOB)**. It enables you to store any type of object created by the computer. A useful example is to use a BLOB to hold images and files from other software packages. For example, each row in the database could hold a different spreadsheet, picture, or graph. An engineering database might hold drawings and specifications for various components. The advantage is that all of the data is stored together, making it easier for users to find the information they need and simplifying backups. Similarly, a database could hold several different revisions of a spreadsheet to show how it changed over time or to record changes by many different users.

On the other hand, BLOBs can quickly eat up space in the database. The free versions of commercial software all place limits on the size of the database files. This limit tends to be around 2 gigabytes. If your application loads thousands of BLOBs into the database, it will quickly reach this upper limit; requiring you to move up to a paid version of the DBMS. Increasing the size of the data files also complicates the backup process and might slow down other operations. So, many developers store binary files as regular operating system files and store the filename within the DBMS—which requires only a few dozen text characters for each file.

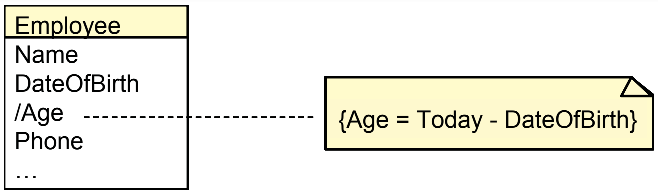


Figure 2.29

Derived values. The Age attribute does not have to be stored, since it can be computed from the date of birth. Hence, it should be noted on the class diagram. Computed attribute names are preceded with a slash.

Computed Values

Some business attributes can be computed. For instance, a sales form typically computes SalePrice times Quantity. Or an employee's age can be computed as the difference between today's date and the DateOfBirth. At the design stage, you should indicate which data attributes could be computed. The UML notation is to precede the name with a slash (/) and then describe the computation in a note. For example, the computation for a person's age is shown in Figure 2.29. The note is displayed as a box with folded corner. It is connected to the appropriate property with a dashed line.

User-Defined Types (Domains/Objects)

A relatively recent object-relational feature is supported by a few of the larger database systems. You can build your own domain as a combination of existing types. This domain essentially becomes a new object type. The example of a geocode is one of the easiest to understand. You can define a geographic location in terms of its latitude and longitude. You also might include altitude if the data is available. In a simple relational DBMS, this data is stored in separate columns. Anytime you want to use the data, you would need to look up and pass all values to your code. With a user-defined data type, you can create a new data type called *geolocation* that includes the desired components. Your column definition then has only the single data type (geolocation), but actually holds two or three pieces of data. These elements are treated by the DBMS as a single entry. Note that when you create a new domain, you also have to create functions to compare values so that you can sort and search using the new data type.

Events

What are events, and how are they described in a database design? Events are another important component of modern database systems that you need to understand and document. Many database systems enable you to write programming code within the database to take action when some event occurs. In general, three different types of events can occur in a database environment:

1. Business events that trigger some function, such as a sale triggering a reduction in inventory.
2. Data changes that signal some alert, such as an inventory that drops below a preset level, which triggers a new purchase order.
3. User interface events that trigger some action, such as a user clicking on an icon to send a purchase order to a supplier.

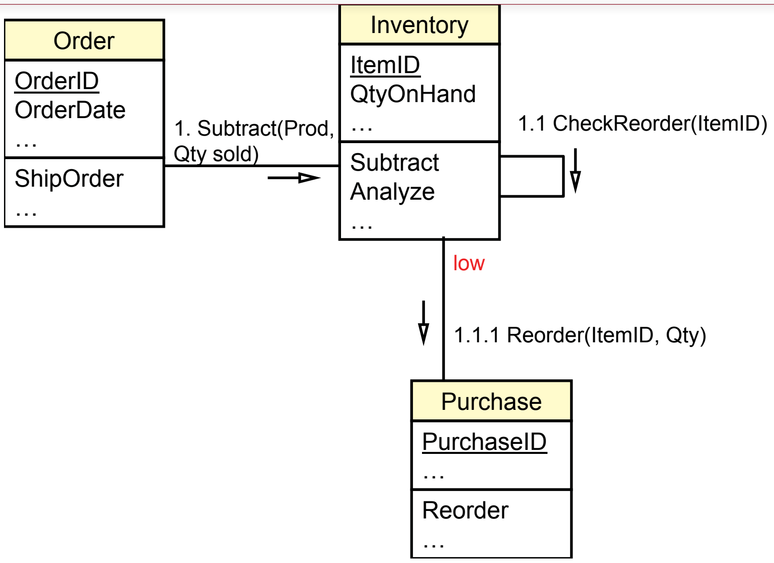


Figure 2.30

Collaboration diagram shows inventory system events. An Order shipment triggers a reduction of inventory quantity on hand which triggers an reorder-point analysis routine. If necessary, the analysis routine triggers a new purchase order for the specified item.

Events are actions that are dependent on time. UML provides several diagrams to illustrate events. The collaboration diagram is the most useful for recording events that happen within the database. Complex user interface events can be displayed on sequence diagrams or statechart diagrams. These latter diagrams are beyond the scope of this book. You can consult an OO design text for more details on how to draw them.

Database events need to be documented because (1) the code can be hard to find within the database itself, and (2) one event can trigger a chain that affects many tables and developers often need to understand the entire chain. Handling business inventory presents a useful example of the issues. Figure 2.30 is a small collaboration diagram that shows how three classes interact by exchanging messages and calling functions from other classes. Note that because the order is important, the three major trigger activities are numbered sequentially. First, when a customer places an order, this business event causes the Order class to be called to ship an order. The shipping function triggers a message to the Inventory class to subtract the appropriate quantity. When an inventory quantity changes, an automatic trigger calls a routine to analyze the current inventory levels. If the appropriate criteria are met, a purchase order is generated and the product is reordered.

The example represents a linear chain of events, which is relatively easy to understand and to test. More complex chains can be built that fork to alternatives based on various conditions and involve more complex alternatives. The UML sequence diagram can be used to show more detail on how individual messages are handled in the proper order. The UML statechart diagrams highlight how a class/object status varies over time. Details of the UML diagramming techniques are covered in other books and online tutorials. For now you should be able to draw simple collaboration diagrams that indicate the primary message events.

```
ON (QuantityOnHand < 100)
THEN Notify_Purchasing_Manager
```

Figure 2.31

Sample trigger. List the condition and the action.

In simpler situations you can keep a list of important events. You can write events as triggers, which describe the event cause and the corresponding action to be taken. For example, a business event based on inventory data could be written as shown in Figure 2.31. Large database systems such as Oracle and SQL Server support triggers directly. Microsoft Access added a few data triggers with the introduction of the 2010 version. You define the event and attach the code that will be executed when the condition arises. These triggers can be written in any basic format (e.g., pseudocode) at the design stage, and later converted to database triggers or program code. UML also provides an Object Constraint Language (OCL) that you can use to write triggers and other code fragments. It is generic and will be useful if you are using a tool that can convert the OCL code into the database you are using.

Large Projects

How are teams organized on large projects? If you build a small database system for yourself or for a single user, you will probably not take the time to draw diagrams of the entire system. However, you really should provide some documentation so the next designer who has to modify your work will know what you did. On the other hand, if you are working on large projects involving many developers and users, everyone must follow a common design methodology. What is a large project and what is a small project? There are no fixed rules, but you start to encounter problems like those listed in Figure 2.32 when several developers and many users are involved in the project.

Figure 2.32

Development issues on large projects. Large projects require more communication, adherence to standards, and project monitoring.

- Design is harder on large projects.
    - Communication with multiple users.
    - Communication between IT workers.
    - Need to divide project into pieces for teams.
    - Finding data/components.
    - Staff turnover-retraining.
- Need to monitor design process.
    - Scheduling.
    - Evaluation.
- Build systems that can be modified later.
    - Documentation.
    - Communication/underlying assumptions and model.

- Computer-Aided Software Engineering
  - Diagrams (linked)
  - Data dictionary
  - Teamwork
  - Prototyping
    - Forms
    - Reports
    - Sample data
  - Code generation
  - Reverse engineering

Figure 2.33

CASE tool features. CASE tools help create and maintain diagrams. They also support teamwork and document control. Some can generate code from the designs or perform reverse engineering.

Methodologies for large projects begin with diagrams such as the class and collaboration diagrams described in this chapter. Then each company or team adds details. For example, standards are chosen to specify naming conventions, type of documentation required, and review procedures.

The challenge of large projects is to split the project into smaller pieces that can be handled by individual developers. Yet the pieces must fit together at the end. Project managers also need to plan the project in terms of timing and expenses. As the project develops, managers can evaluate team members in terms of the schedule.

Several types of tools can help you design database systems, and they are particularly useful for large projects. To assist in planning and scheduling, managers can use project-planning tools (e.g., Microsoft Project) that help create Gantt and PERT charts to break projects into smaller pieces and highlight the relationships among the components. Computer-assisted software engineering (CASE) tools (like IBM's Rational set) can help teams draw diagrams, enforce standards, and store all project documentation. Additionally, groupware tools (like SharePoint or Lotus Notes/Domino) help team members share their work on documents, designs, and programs. These tools annotate changes, record who made the changes and their comments, and track versions.

As summarized in Figure 2.33, CASE tools perform several useful functions for developers. In addition to assisting with graphical design, one of the most important functions of CASE tools is to maintain the data repository for the project. Every element defined by a developer is stored in the data repository, where it is shared with other developers. In other words, the data repository is a specialized database that holds all of the information related to the project's design. Some CASE tools can generate databases and applications based on the information you enter into the CASE project. In addition, reverse-engineering tools can read files from existing applications and generate the matching design elements. These CASE tools are available from many companies, including Rational Software, IBM, Oracle, and Sterling Software. CASE tools can speed the design and development process by improving communication among developers and through generating code. They offer the potential to reduce maintenance time by providing complete documentation of the system.

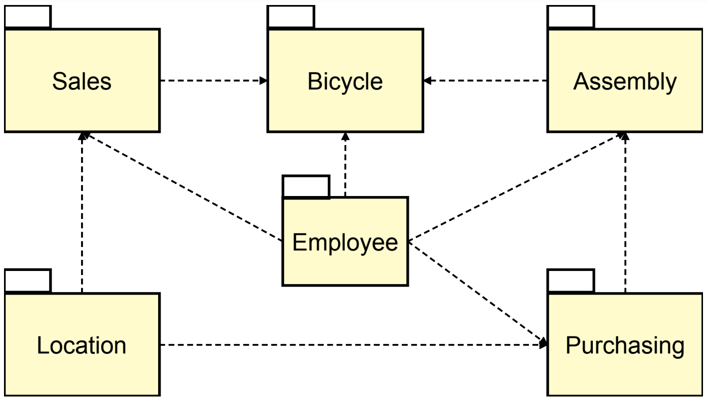


Figure 2.34

Rolling Thunder Bicycles—top-level view. The packages are loosely based on the activities of the firm. The goal is for each package to describe a self-contained collection of objects that interacts with the other packages.

Good CASE tools have existed for several years, yet many firms do not use them, and some that have tried them have failed to realize their potential advantages. Two drawbacks to CASE tools are their complexity and their cost. The cost issue can be mitigated if the tools can reduce the number of developers needed on a given project. But their complexity presents a larger hurdle. It can take a developer several months to learn to use a CASE tool effectively. Fortunately, some CASE vendors provide discounts to universities to help train students in using their tools. If you have access to a CASE tool, use it for as many assignments as possible.

Rolling Thunder Bicycles

How does UML split a big project into packages? The Rolling Thunder Bicycle case illustrates some of the common associations that arise in business settings. Because the application was designed for classroom use, many of the business assumptions were deliberately simplified. The top-level view is shown in Figure 2.34. Loosely based on the activities of the firm, the elements are grouped into six packages: Sales, Bicycles, Assembly, Employees, Purchasing, and Location. The packages will not be equal: some contain far more detail than the others. In particular, the Location and Employee packages currently contain only one or two classes. They are treated as separate packages because they both interact with several classes in multiple packages. Because they deal with independent, self-contained issues, it makes sense to separate them.

Each package contains a set of classes and associations. The Sales package is described in more detail in Figure 2.35. To minimize complexity, the associations with other packages are not displayed in this figure. For example, the Customer and RetailStore classes have an association with the Location::City class. These relationships will be shown in the Location package. Consequently, the Sales package is straightforward. Customers place orders for Bicycles. They might use a RetailStore to help them place the order, but they are not required to do so. Hence the association from the RetailStore has a (0…1) multiplicity.

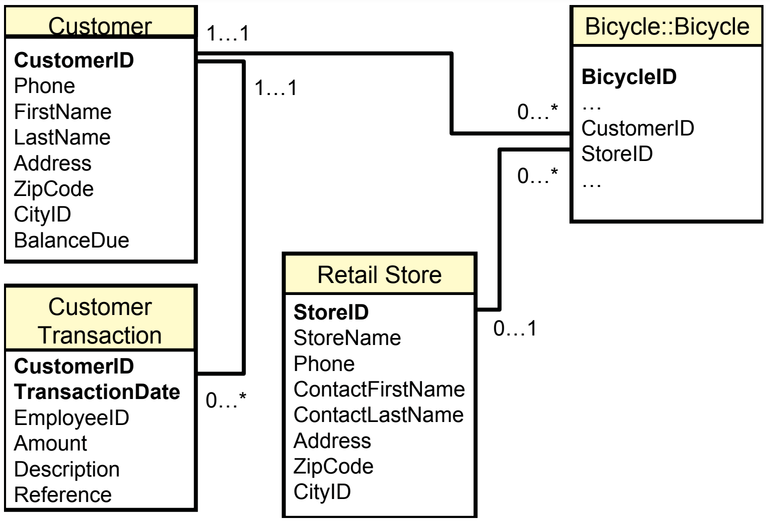
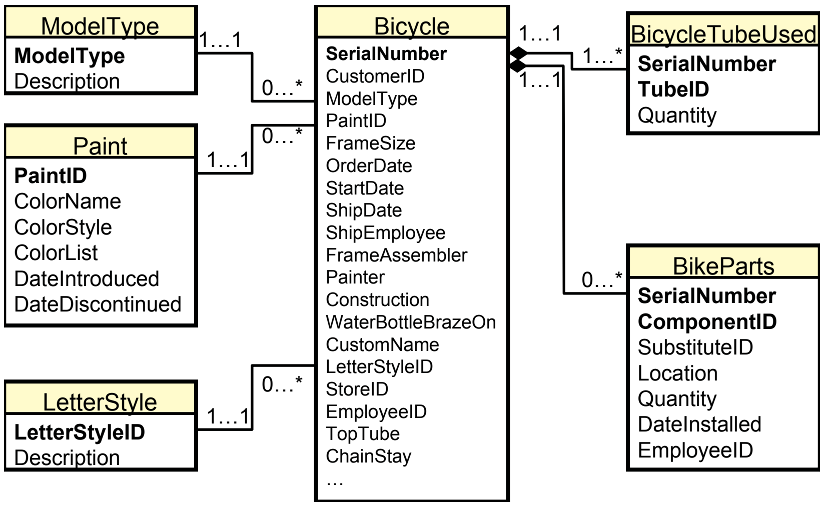


Figure 2.35

Rolling Thunder Bicycles—Sales package. Some associations with other packages are not shown here. (See the other packages.)

Figure 2.36

Rolling Thunder Bicycles—Bicycle package. Note the composition associations into the Bicycle class from the BikeTubes and BikeParts classes. To save space, only some of the Bicycle properties are displayed.



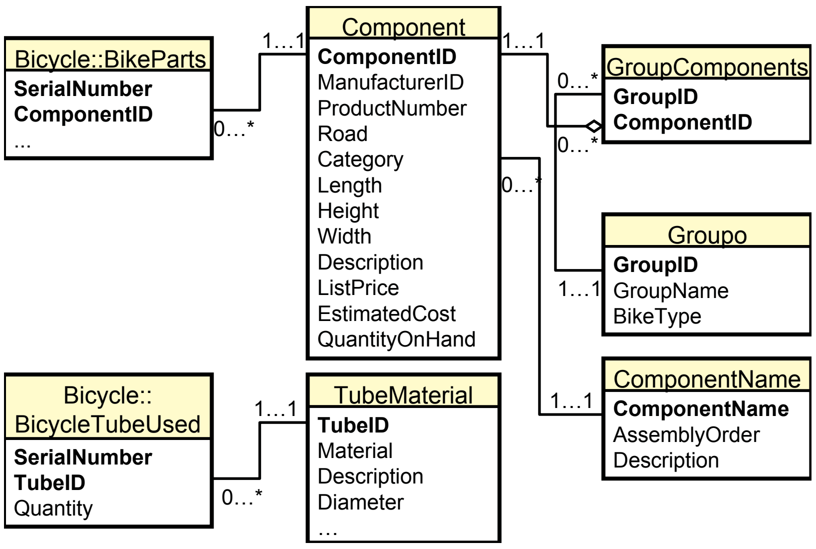
The Bicycle package contains many of the details that make this company unique. To save space, only a few of the properties of the Bicycle class are shown in Figure 2.36. Notice that a bicycle is composed of a set of tubes and a set of components. Customers can choose the type of material used to create the bicycle (aluminum, steel, carbon fiber, etc.). They can also select the components (wheels, crank, pedals, etc.) that make up the bicycle. Both of these classes have a composition association with the Bicycle class. The Bicycle class is one of the most important classes for this firm. In conjunction with the BicycleTubeUsed and BikeParts classes, it completely defines each bicycle. It also contains information about which employees worked on the bicycle. This latter decision was a design simplification choice. Another alternative would be to move the ShipEmployee, FrameAssembler, and other employee properties to a new class within the Assembly package.

As shown in Figure 2.37, the Assembly package contains more information about the various components and tube materials that make up a bicycle. In practice, the Assembly package also contains several important events. As the bicycle is assembled, data is entered that specifies who did the work and when it was finished. This data is currently stored in the Bicycle class within the Bicycle package. A collaboration diagram or a sequence diagram would have to be created to show the details of the various events within the Assembly package. For now, the classes and associations are more important, so these other diagrams are not shown here.

All component parts are purchased from other manufacturers (suppliers). The Purchase package in Figure 2.38 is a fairly traditional representation of this activity. Note that each purchase requires the use of two classes: PurchaseOrder and PurchaseItem. The PurchaseOrder is the main class that contains data about the

Figure 2.37

Rolling Thunder Bicycles—Assembly package. Several events occur during assembly, but they cannot be shown on this diagram. As the bicycle is assembled, additional data is entered into the Bicycle table within the Bicycle package.



order itself, including the date, the manufacturer, and the employee who placed the order. The PurchaseItem class contains the detail list of items that are being ordered. This class is specifically included to avoid a many-to-many association between the PurchaseOrder and Component classes.

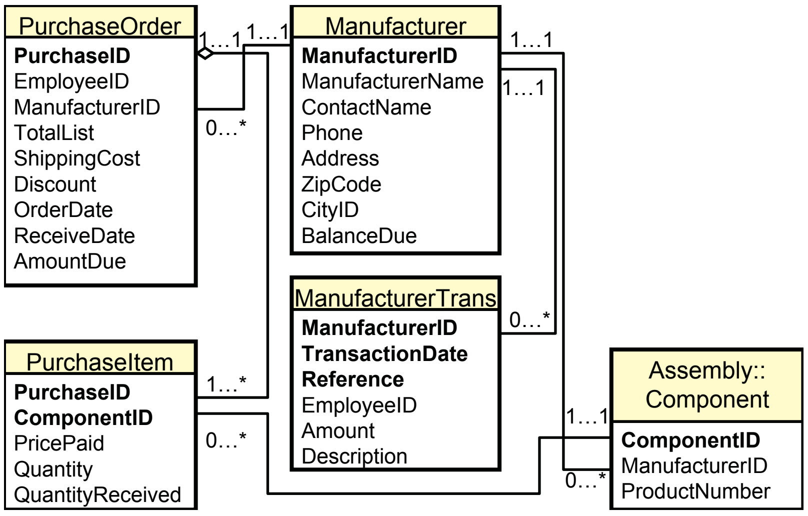
Observe from the business rules that a ManufacturerID must be included on the PurchaseOrder. It is dangerous to issue a purchase order without knowing the identity of the manufacturer. Chapter 10 explains how security controls can be imposed to provide even more safety for this crucial aspect of the business.

An additional class (ManufacturerTransactions) is used as a transaction log to record each purchase. It is also used to record payments to the manufacturers. On the purchase side, it represents a slight duplication of data (AmountDue is in both the PurchaseOrder and Transaction classes). However, it is a relatively common approach to building an accounting system. Traditional accounting methods rely on having all related transaction data in one location. In any case the class is needed to record payments to the manufacturers, so the amount of duplicated data is relatively minor.

The Location package in Figure 2.39 was created to centralize the data related to addresses and cities. Several classes have address properties. In older systems it was often easier to simply duplicate the data and store the city, state, and ZIP code in every class that referred to locations. Today, however, it is relatively easy to obtain useful information about cities and store it in a centralized table. This approach improves data entry, both in speed and data integrity. Clerks can simply choose a location from a list. Data is always entered consistently. For example, you do not have to worry about abbreviations for cities. If telephone area codes or ZIP codes are changed, you need to change them in only one table. You can also store additional information that will be useful to managers. For example, the

Figure 2.38

Rolling Thunder Bicycles—Purchasing package. Note the use of the Transaction class to store all related financial data for the manufacturers in one location.



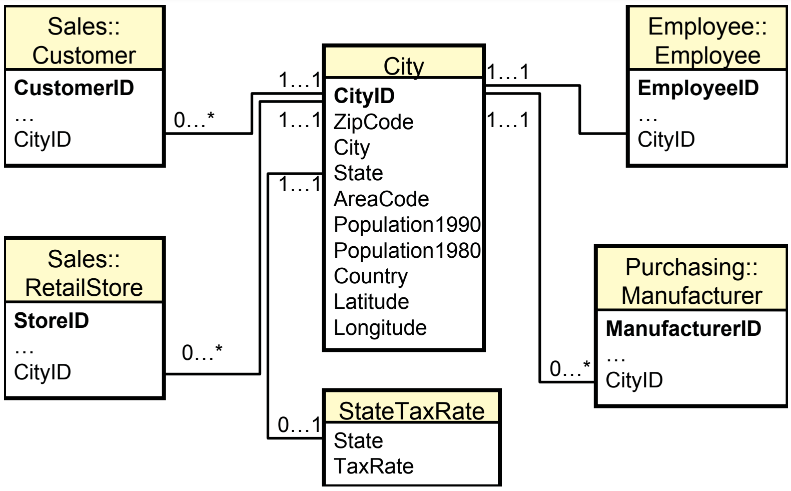


Figure 2.39

Rolling Thunder Bicycles—Location package. By centralizing the data related to cities, you speed clerical tasks and improve the quality of the data. You can also store additional information about the location that might be useful to managers.

population and geographical locations can be used to analyze sales data and direct marketing campaigns.

The Employee package is treated separately because it interacts with so many of the other packages. The Employee properties shown in Figure 2.40 are straightforward. Notice the reflexive association that denotes the management relationship. For the moment there is only one class within the Employee package. In actual practice this is where you would place the typical human resources data and associations. For instance, you would want to track employee evaluations, assignments, and promotions over time. Additional classes would generally be related to benefits such as vacation time, personal days, and insurance choices.

A detailed, combined class diagram for Rolling Thunder Bicycles is shown in Figure 2.41. Some associations are not included—partly to save space. A more important reason is that all of the drawn associations are enforced by Microsoft Access. For example, once you define the association from Employee to Bicycle, Access will only allow you to enter an EmployeeID into the Bicycle class that already exists within the Employee class. This enforcement makes sense for the person taking the order. Indeed, financial associations should be defined this strongly. On the other hand, the company may hire temporary workers for painting and frame assembly. In these cases the managers may not want to record the exact person who painted a frame, so the association from Employee to Painter in the Bicycle table is relaxed.

Application Design

What is an application? The concept of classes and attributes seems simple at first, but can quickly become complicated. Practice and experience make the process easier. For now, learn to focus on the most important objects in a given project. It is often easiest to start with one section of the problem, define the basic

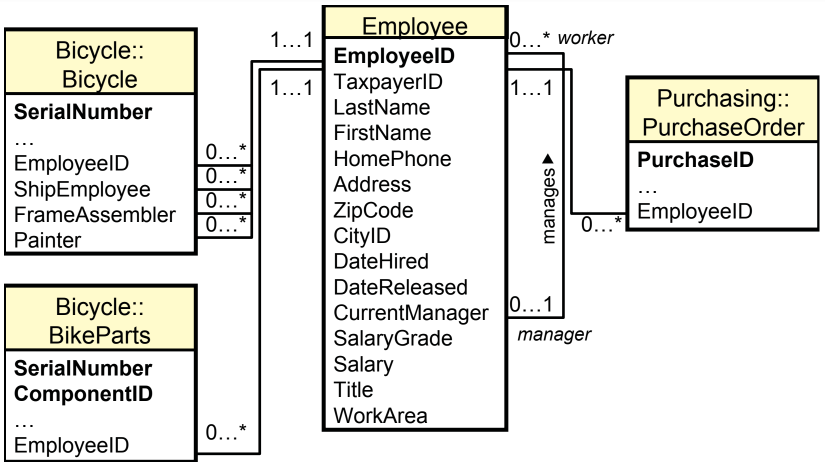


Figure 2.40

Rolling Thunder Bicycles—Employee package. Note the reflexive association to indicate managers.

elements, add detail, then expand into other sections. As you are designing the project, remember that each class becomes a table in the database, where each attribute is a column, and each row represents one specific object.

You should also begin thinking about application design in terms of the forms or screens that users will see. Consider the simple form in Figure 2.42. On paper, this form would simply have blanks for each of the items to be entered. Eventually, you could build the same form with blanks as a database form. In this case, you might think only one table is associated with this form; however, you need to think about the potential problems. With blank spaces on the form, people can enter any data they want. For example, do users really know all of the breed types? Or will they sometimes leave it blank, fill in abbreviations, or misspell words? All of these choices would cause problems with your database. Instead, it will be better to give them a selection box, where users simply pick the appropriate item from a list. But that means you will need another table that contains a list of possible breeds. It also means that you will have to establish a relationship between the Breed table and the Animal table. In turn, this relationship affects the way the application will be used. For example, someone must enter all of the predefined names into the Breed table before the Animal table can even be used.

At this point in the development, you should have talked with the users and collected any forms and reports they want. You should be able to sketch an initial class diagram that shows the main business objects and how they relate to each other, including the multiplicity of the association. You should also have a good idea about what attributes will be primary keys, or keys that you will need to create for some tables. You also need to specify the data domains of each property.

Corner Med

What process is followed when starting a project? Before you can design tables and relationships, you need to talk with the users and determine what data needs to be collected. It is easier to understand the users if you have some knowledge of

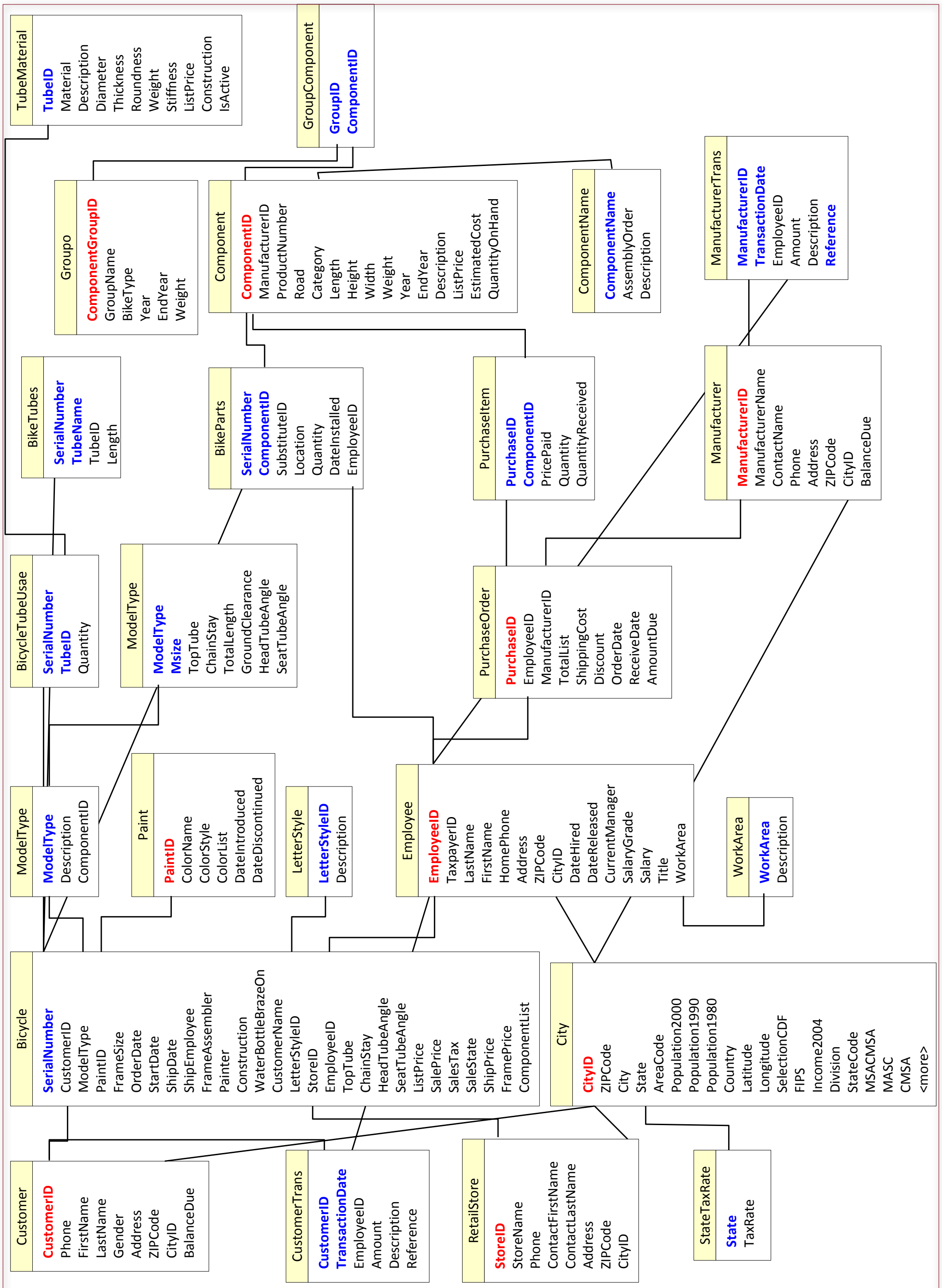
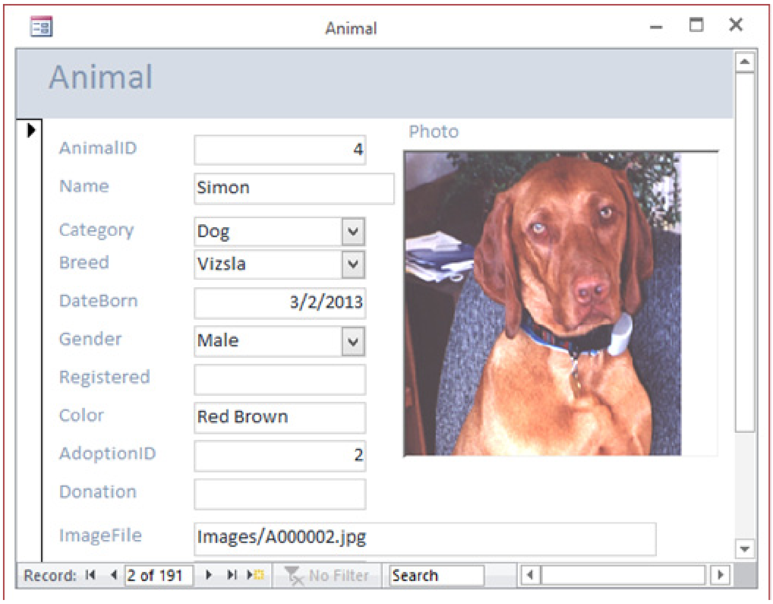


Figure 2.41

Rolling Thunder detailed class diagram. The detail class diagram is a nice reference tool for understanding the organization, but for many organizations this diagram will be too large to display at this level of detail.


Animal

Animal

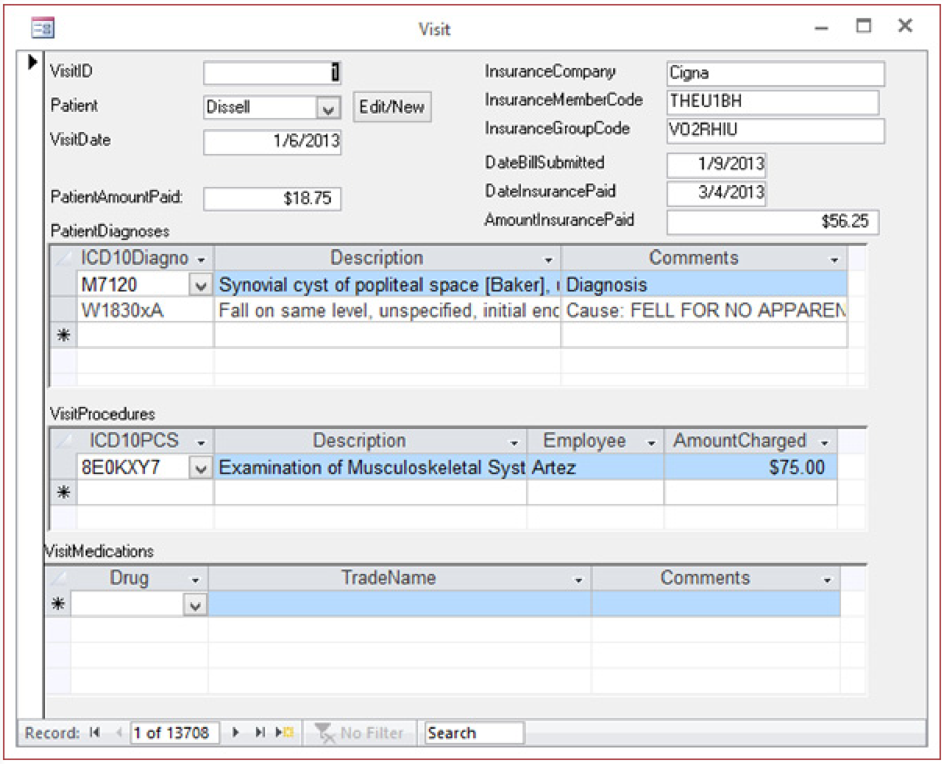
AnimalID	4
Name	Simon
Category	Dog
Breed	Vizsla
DateBorn	3/2/2013
Gender	Male
Registered	
Color	Red Brown
AdoptionID	2
Donation	
ImageFile	Images/A000002.jpg

Figure 2.42

Basic Animal form. Initially this form seems to require one table (Animal). But to minimize data-entry errors, it really needs a table to hold data for Category and Breed, which can be entered via a selection box.

their field. You probably do not need a medical degree to build a business system for physicians; however, you will have to learn some of the basic terminology to understand the various data relationships. This is a good place to point out that the sample Corner Med database is merely a start of an application. None of the components should be used in an actual medical situation. It is designed purely as a demonstration project to highlight some of the issues in database design.

In a family-practice physician office, the patient visit is going to be a key element in any business administration system. Figure 2.43 shows a simple version of a form to record data about a patient visit. The first thing to note is that the main form contains two subforms. Note that each subform represents repeating data or a one-to-many relationship. A useful question to ask the managers at this point would be to confirm the insurance data. In particular, can patients have more than one insurance plan? If this data is important, the form would have to be modified to add a repeating section for insurance data. It might be tempting to argue that almost all data could potentially be repeating, so perhaps there should be dozens of repeating sections on the form. Given the state of health insurance in the U.S., it is possible that you will need to add this repeating section. However, be cautious with other items. One-to-many relationships add flexibility to collecting and storing data, but they make the data form considerably more complex. If patients rarely have more than one insurance provider, it will be cumbersome for the clerks to deal with the extra repeating section when it is rarely used. On the other hand, the patient diagnoses and treatments sections are required because most patient visits will require multiple entries. The patient visit form also illustrates one of the key steps in starting a database project: Collect input forms and reports from users so you can identify the data that needs to be stored.


Visit

VisitID	1	InsuranceCompany	Cigna
Patient	Dissell	InsuranceMemberCode	THEU1BH
VisitDate	1/6/2013	InsuranceGroupCode	V02RHIU
		DateBillSubmitted	1/9/2013
PatientAmountPaid:	$18.75	DateInsurancePaid	3/4/2013
		AmountInsurancePaid	$56.25

PatientDiagnoses

ICD10Diagno	Description	Comments
M7120	Synovial cyst of popliteal space [Baker], ι	Diagnosis
W1830xA	Fall on same level, unspecified, initial enc	Cause: FELL FOR NO APPAREN

VisitProcedures

ICD10PCS	Description	Employee	AmountCharged
8E0KXY7	Examination of Musculoskeletal Syst	Artez	$75.00

VisitMedications

Drug	TradeName	Comments

Record: 1 of 13708 No Filter Search

Figure 2.43

Patient Visit form. This form has two repeating sections: One for diagnoses and one for treatment. Many more details can be added but it is possible to start with these key data elements.

When you look at the patient visit form, you should start thinking about the tables that will be needed to hold the data. At the start, you should quickly identify three starting tables: (1) PatientVisit, a table that represents the form itself, (2) PatientDiagnoses, a table that arises because of the first repeating section, and (3) PatientProcedures, a table representing the second repeating section.

When you identify a new table, you should also think about the possible key columns. The PatientVisit table will most likely need a generated key—a value that the DBMS will create whenever a new visit is added to the database. Call the column VisitID. It is the best way to guarantee a unique value for every visit. A generated VisitID value also makes it easier to identify the keys for the repeating sections. Each of these tables will need two key columns. For instance, VisitID, ICD10Diagnosis will be the two key columns for the PatientDiagnoses table. It is easy to verify that both columns need to be keyed because on a specific visit, a patient could be diagnosed with many different problems, requiring ICD10Diagnosis to be part of the key. In reverse, a specific diagnosis could be applied to many different visits (either for one patient or different patients), requiring VisitID to be keyed. The same analysis reveals the two keys required for the PatientProcedures table.

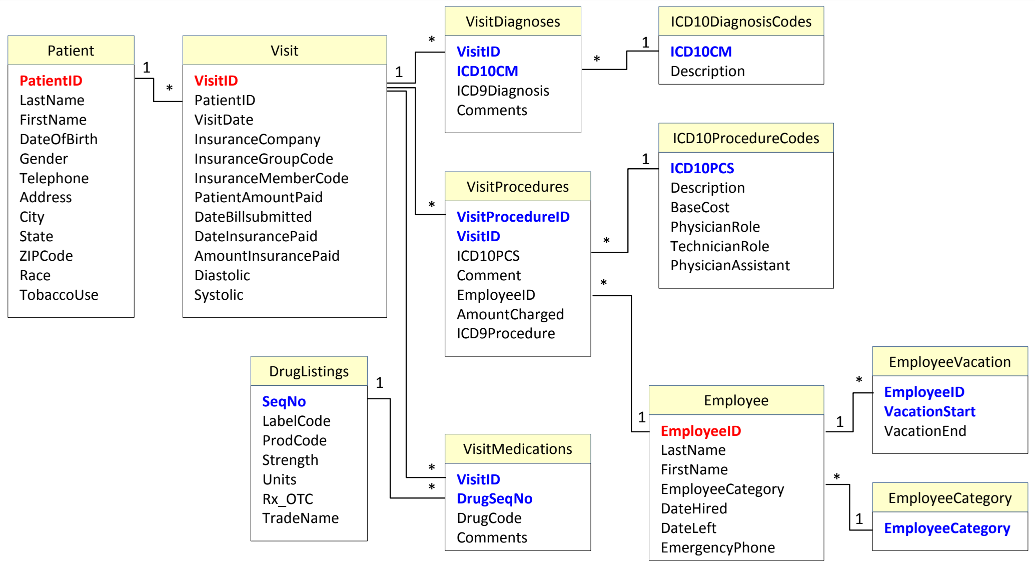


Figure 2.44

Corner Med basic tables. Ultimately, all of these tables will contain more data columns.

The next step is to ask where the ICD10Diagnosis and ICD10Procedure columns will be defined. These are slightly trickier in the context of the medical world. Ideally, you would create a table of standard codes for each of these values. The best approach would be to purchase a complete list of codes. For example, you could buy the current ICD10 codes from the United Nation's World Health Organization. Enabling physicians to pull the codes from a standard list will reduce errors. However, it would also require physicians to become familiar with the codes and to take the time to read through the list to find the specific code for every diagnosis and procedure. In practice, large healthcare institutions find it more efficient to have physicians enter written descriptions of diagnoses and procedures and hire medical coders to identify the specific codes later. This decision is an example of a complex business problem that you will have to solve early in the design process. In many cases, you will have to outline the options and present them to senior management for the final decision.

Figure 2.44 shows the basic tables used for the Corner Med case. In a real case, all of these tables will contain more data columns. However, the strength of the relational data model is that the basic structure will remain the same. It is relatively easy to add more columns to each table later. Notice that data for all employees is handled in a single class. That is, physician, nurse, and clerical data are all stored in the same table. The employees are identified by EmployeeCategory which is stored in a lookup list. However, you might want to think about this decision. The company might want to keep considerably more data for physicians. This data could be highly specialized, such as license number and date. If the amount of data gets large, it will be more efficient to store data for physicians in a table separate from the other employees. Otherwise, you will waste space and complicate the data-entry form for employees where you do not need this extra data.

By now, you should be able to make a first pass at creating a class diagram for a specific problem. You should also recognize that the final structure of the diagram depends on the business rules and assumptions. You can often resolve these questions by talking with users, but some decisions have to be passed up to senior management.

Summary

Managing projects to build useful applications and control costs is an important task. The primary steps in project management are the feasibility study, systems analysis, systems design, and implementation. Although these steps can be compressed, they cannot be skipped.

The primary objective is to design an application that provides the benefits needed by the users. System models are created to illustrate the system. These models are used to communicate with users, communicate with other developers, and help us remember the details of the system. Because defining data is a crucial step in developing a database application, the class diagram is a popular model.

The class diagram is created by identifying the primary entities in the system. Entities are defined by classes, which are identified by name and defined by the properties of each entity. Classes can also have functions that they perform.

Associations among classes are important elements of the business design because they identify the business rules. Associations are displayed as connecting lines on the class diagram. You should document the associations by providing names where appropriate, and by identifying the multiplicity of the relationship. You should be careful to identify special associations, such as aggregation, composition, generalization, and reflexive relationships.

Designers also need to identify the primary events or triggers that the application will need. There are three types of events: business events, data change events, and user events. Events can be described in terms of triggers that contain a condition and an action. Complex event chains can be shown on sequence or collaboration diagrams.

Designs generally go through several stages of revision, with each stage becoming more detailed and more accurate. A useful approach is to start with the big picture and make sure that your design identifies the primary components that will be needed in the system. Packages can be defined to group elements together to hide details. Detail items are then added in supporting diagrams for each package in the main system diagram.

Models and designs are particularly useful on large projects. The models provide a communication mechanism for the designers, programmers, and users. CASE tools are helpful in creating, modifying, and sharing the design models. In addition to the diagrams, the CASE repository will maintain all of the definitions, descriptions, and comments needed to build the final application.

A Developer's View

Like any developer, Miranda needs a method to write down the system goals and details. The feasibility study documents the goals and provides a rough estimate of the costs and benefits. The class diagram identifies the main entities and shows how they are related. The class diagram, along with notes in the data dictionary, records the business rules. For your class project, you should study the case. Then create a feasibility study and an initial class diagram.

Key Terms

aggregation	generalization
association	inheritance
association role	method
attribute	multiplicity
binary large object (BLOB)	n-ary association
class	null
class diagram	polymorphism
class hierarchy	primary key
collaboration diagram	property
composition	rapid application development (RAD)
data normalization	reflexive association
data type	relational database
derived class	relationship
encapsulation	table
entity	Unified Modeling Language (UML)

Review Questions

1. How do you identify user requirements?
2. What is the purpose of a class diagram (or entity-relationship diagram)?
3. What is a reflexive association and how is it shown on a class diagram?
4. What is multiplicity and how is it shown on a class diagram?
5. What are the primary data types used in business applications?
6. How is inheritance shown in a class diagram?
7. How do events and triggers relate to objects or entities?
8. What problems are complicated with large projects?
9. How can computer-aided software engineering tools help on large projects?
10. What is an application?

Exercises

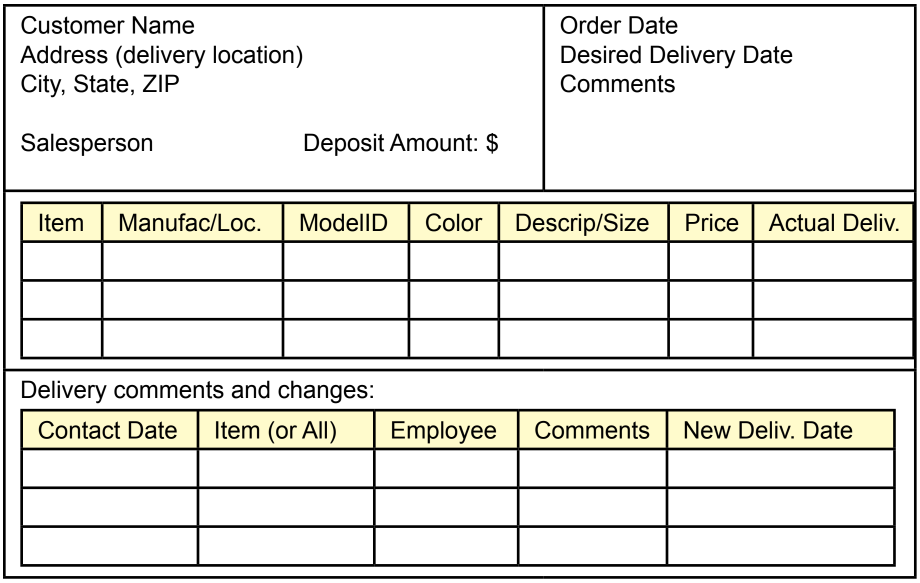
1. Most medical practices use turn-key systems to handle billing data and basic electronic medical records. But a local physician wants a system to help him perform deeper statistical analyses on basic patient data. Specifically, he wants to track basic test results for patients and wants to compare them by families. For example, he wants to see if families where parents have high blood pressure also affect the blood pressure of the children and at what age those values change. Initially, he just wants to track basic medical values including heart rate, pressure, and basic blood test. Notes:

Patient Last name, First Name		Father:
Date of Birth		
Gender		Mother:
Race		
Tobacco y/n	Alcohol y/n	Family history notes, particularly if data is not available.
Marital Status		
Phone, e-mail		
Address, City, State, ZIP		

Test date:
Last meal time:

Test	Meas.	Value	Low	High	Comments
Albumin	*g/dL*		*3.9*	*5.0*	
Alkaline phosphatase	*IU/L*		*44*	*147*	
ALT	*IU/L*		*8*	*37*	
AST	*IU/L*		*10*	*34*	
BUN	*mg/dL*		*7*	*20*	
Calcium	*mg/dL*		*8.5*	*10.9*	
Chloride	*mmol/L*		*96*	*106*	
CO2	*mmol/L*		*20*	*29*	
Creatinine	*mg/dL*		*0.8*	*1.4*	
Glucose	*mg/dL*		*100*		
Potassium	*mEq/L*		*3.7*	*5.2*	
Sodium	*mEg/L*		*136*	*144*	
Total bilirubin	*mg/dL*		*0.2*	*1.9*	
Total protein	*g/dL*		*6.3*	*7.9*	
Blood Pressure-systolic	*mm Hg*		*90*	*140*	
Pressure-diastolic	*mm Hg*		*60*	*90*	
Heart rate	*bpm*		*50*	*90*	

2. A local store that sells household appliances wants a database to track special orders. Most of the items ordered are large and from high-end vendors so they are too expensive to stock in the display room. Also, customers tend to order them when they are remodeling their houses so they do not want the items immediately. Instead, they need to be ordered and scheduled for delivery on a specified date. Of course, construction delays are common so the managers also need the ability to delay delivery by a few days or weeks when necessary. Most of the items are ordered directly from the manufacturers and they are good at scheduling deliveries, but sometimes highly-customized items need to be tracked down at other stores across the nation.



Customer Name
Address (delivery location)
City, State, ZIP

Salesperson Deposit Amount: $

Order Date
Desired Delivery Date
Comments

Item	Manufac/Loc.	ModelID	Color	Descrip/Size	Price	Actual Deliv.

Delivery comments and changes:

Contact Date	Item (or All)	Employee	Comments	New Deliv. Date

If the item is not available from the manufacturer, the location is specified in terms of the store and contact information. Typically, it is ordered immediately and held in storage until needed; because it is too hard to find a new version so safer to buy it now.

3. A friend of yours lives in a town with many older houses and he repairs antique lights for homeowners. Many of the lights use crystals and colored panels that were designed by artists. Fortunately, most of the electrical components are relatively standard and compatible with today's parts. The most common problems involve the wiring, because old wires used paper and plastic insulation that tends to crack and disintegrate over time. He often has to rewire the entire light and he usually replaces the bulb sockets at the same time. When possible, he takes down the lights and brings them to his shop, other times he has to set up appointments to do the work in place because the light cannot be removed easily. He needs an application to track appointments, the work done, and a basic billing system that includes his time and the parts used.

Contact Last Name, First Name Phone Address City, State, ZIP	Contact Date Referred by: Date Paid:

Description of problem
Description of light, style, est. year
In-place or shop

Date	Work Performed	Hours

Part No.	Description	Quantity	Cost	Source

Total Hours: ______ Total Parts: ______ Amount Due: ______	Rate: _______ Value: _______

4. A local day spa wants you to build an application that can be used to track services provided by the various employees. You do not need to handle reservations and scheduling—which are currently handled by an online provider based primarily on number of slots available during the day. Instead, you want to focus on billing and payments for an application that will be used when clients arrive. In the past, a paper card would be created for guests and services listed on the cards. Payment methods often include gift cards. Staff members often write up comments regarding treatments of clients so they can refer to them when the client returns in the future. Most treatments have a set amount of time, such as 50 minutes for a massage. At the end of the visit, clients are asked to evaluate the staff members in terms of the service quality, knowledge, and friendliness. For most people, the owner simply talks to the guests and then fills out a form later. In a modern twist, the staff members are also asked to rate the clients—largely in terms of dealing with special requests; which might lead to changes in the treatment offerings.

Guest Name Cell phone, Address, City, State, ZIP Health issues or concerns	Date Payment method

Room & Time	Treatment	Staff Specialty, Phone	Staff comments	Amount & Tip

	Subtotal Tax Total

Guest, Gender, Approx. Age Facility comments/overall	Comments

Staff member	Treatment	Quality	Knowledge	Friendliness	Staff rate Guest

Suggested changes in treatments

Treatment	Change	Est. Time

5. A small company makes winter gloves for men and women. Originally, the gloves were woven wool, but recently the company has also added leather gloves and might consider synthetic materials in the future. The woolen gloves come in a variety of colors. Sizes are typically small, medium, and large which are slightly different for men and women (largely in terms of finger length). The factory also produces different styles which tend to be variations in length of the glove, cuffs, or designs in the stitching or emblems. The company needs a database to track production and shipments.

   Production runs emphasize a single style, material, and size. Changes in yarn or material color do not require reconfiguring the machine so colors are tracked within the same production run. The IDs from the input material are tracked and an employee inspects the output batch and adds any comments.

Production Run ID Machine ID Material Glove Style Men/Women Size	Date Start Time Employee Last/First Name Job Title InspectorID

Color	Quantity Made	Material Batch	Qty Rejected	Comments

Order ID Customer/Store Contact Person Address City, State, ZIP	Order Date Ship Date

ItemID	Description	Size	Color	Style	Gender	Quantity	Sale Price

6. A start-up company is assembling customized stereo ear-buds. Instead of shipping dozens of different sizes and shapes of in-ear pieces, the company distributes a clear piece of plastic with various marks. Customers hold the piece on their ear and take a photograph. A system at the company reads the markings and determines the best ear-piece for each customer. The assembly team then builds the custom ear-buds for the customer allowing them to set additional specifications including colors, logos, number of "armatures" or speaker cones, and control switches for android or Apple devices. The company needs a database to track orders, assembly, and shipping.

Customer Last name, First name Phone, e-mail Address City, State, Postal Code Country		Referrer/Source
Measure Photo Horizontal Distance Vertical Distance Depth	Left Ear	Right Ear
Color Wire color/type Logo Armatures Apple/Android		

Assembly Date
Work station #
Employee Name, Date Hired, Title

Customer Order ID Comments/ Changes	Item ID	Quantity	Comments	Start Time End Time

Inspector ID, Supervisor

Test	Value	Pass/Fail
Audio low		
Audio high		
Stretch		

Inspection outcome:

7. A rich uncle owns about a dozen classic automobiles. He keeps them in several garages around town and actually drives them for different events. At a recent family holiday get-together, he mentioned that he struggles to remember the maintenance schedules for all of the vehicles. He has records for all of the service work, including oil changes, tune-ups, and other repairs. But currently, they are just paper receipts and he needs a way to track and schedule the maintenance so that any of the cars will be available for use when he wants it. He also would like to plan his usage so that he doesn't have to get all of the cars serviced at the same time. You suggested that it would be a good application for a database. Actually, you are really hoping he will let you borrow one of them for a date; but first you have to design the database.

Car Make, Model, Year, Color VIN Storage Location, Name, Address Size, Heat/Cool Monthly Cost	Date Acquired Exterior Condition Interior Condition Amount Paid Source

Manufacturer Service Intervals

Miles Interval	
3000	*Oil change*
	Grease frame fittings
	Tire pressure
15000	*Replace spark plugs,…, Tune-up*
	Rotate tires
	Grease door fittings

Actual Service Records

Miles	Date	Location	Service	Comments	Cost	
					Labor	Parts

8. Experience exercise: Talk to a friend, relative, or local manager to identify a basic job and create a class diagram for the problem.

9. Identify the typical relationships between the following entities. Write down any assumptions or comments that affect your decision. Be sure to include minimum and maximum values. Use the Internet to look up terms and examples.

   a) Company, CEO

   b) Restaurant, Cook

   c) TV Show, commercial ad

   d) E-mail address, computer user

   e) Item, List price

   f) Car, Car wash

   g) House, Painter

   h) Dog, Owner

   i) Manager, Worker

   j) Doctor, Patient

10. For each of the entities in the following list (left side), identify whether each of the items on the right should be an attribute of that entity or a separate entity.

a)	Employee	Name, Date Hired, Manager, Spouse, Job
b)	Factory	Manager, Address, Supplier, Machine, Size
c)	Boat	Dock, Length, Passenger, Captain, Weight
d)	Dentist	Patient, Graduate School, Emergency Phone, Drill
e)	Library	Book, Librarian, Number of Books, Visitor

Sally's Pet Store

11. Do some initial research on retail sales and pet stores. Identify the primary benefits you expect to gain from a transaction processing system for Sally's Pet Store. Estimate the time and costs required to design and build the database application.

12. Extend the class diagram by adding comments about each animal, beginning with adoption group remarks and including comments by employees and customers.

13. Write classes for the pet store case to track special sales events. Every couple of months the store has clearance sales and places specific items on sale. Eventually, Sally wants to evaluate the sales data to see how customers respond to the reduced prices.

14. Extend the pet store class diagram to include scheduling of appointments for pet grooming.

Rolling Thunder Bicycles

15. The Bicycle table includes entries for several employees who worked on the bike. The advantage to this approach is that it leaves all the work in one table and identifies the work performed, making it easier to enter the data. The drawback is that it is more difficult to query (and would require several links to the Employee table). Redesign the table to eliminate these problems.
16. Rolling Thunder Bicycles is thinking about opening a chain of bicycle stores. Explain how the database would have to be altered to accommodate this change. Add the proposed components to the class diagram.
17. If Rolling Thunder Bicycles wants to add a Web site to sell bicycles over the Internet, what additional data needs to be collected? Extend the class diagram to handle this additional data.

Corner Med

18. One of the first things Corner Med needs for the database is the ability to enter multiple numbers for the physicians, such as pager and cell phone. Add the necessary class.
19. Corner Med needs more information about insurance companies. Each company requires claims to be submitted to a specific location. Today, much of the data can be submitted electronically, so there will be an electronic address as well as a physical address. There will also be an account number and password, as well as a phone number and contact person. Add these elements to the class diagram.
20. In theory, prescriptions could be handled as ICD10 procedures. However, because of various drug laws, including pharmacy verification and tracking needs, it is easier to store the data separately. Add the class(es) to the diagram to handle drug prescriptions. Be sure to include the drug name, the dosage, instructions for taking the drug, and the time period. Note that you do not need to add a Drug table because it would be too large and change too often; although the physicians might want to add the Physician's Desk Reference (PDR) on CD later.

Web Site References

http://www.rational.com/uml/	The primary site for UML documentation and examples.
http://www.iconixsw.com	UML documentation and comments.
http://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm	Oracle data type description.
http://msdn.microsoft.com/en-us/library/ms187752(SQL.90).aspx	SQL Server data types.
http://msdn.microsoft.com/en-us/library/ms130214.aspx	SQL Server Books Online documentation.
http://JerryPost.com/DBDesign	Database design system.

Additional Reading

Codd, E. F., “A Relational Model of Data for Large Shared Data Banks,” Communications of the ACM, 13 no. 6, (1970), pp. 377-387. [The paper that initially described the relational model.]

Constantine, L., “Under Pressure,” Software Development, October 1995, pp. 111-112. [The importance of design.]

Constantine, L., “Re: Architecture,” Software Development, January 1996, pp. 87-88. [Update on a design competition.]

McConnell, S., Rapid Development: Taming Wild Software Schedules, Redmond: Microsoft Press, 1996. [An excellent introduction to building systems, with lots of details and examples.]

Penker, M. and H. Eriksson, Business Modeling with UML: Business Patterns at Work, New York: John Wiley & Sons, 2000. [Detailed application of UML to business applications.]

Silverston, Len, The Data Model Resource Book, Vol 1 and 2, 2001, New York: John Wiley & Sons. [A collection of sample models for a variety of businesses.]

Appendix: Database Design System

Many students find database design to be challenging to learn. The basic concept seems straightforward: define a table that represents one basic entity with columns that describe the properties to hold the necessary data. For example, a Customer table will have columns for CustomerID, LastName, FirstName, and so on. But it is often difficult to decide exactly which columns belong in a table. It is also difficult to identify the key columns, which are used to establish relationships among tables. The design is complicated by the fact that the tables reflect the underlying business rules, so students must also understand the business operations and constraints in order to create a design that provides the functionality needed by the business.

In addition to reading Chapters 2 and 3 closely, one of the most important steps in learning database design is to work as many problems as possible. The catch is that students also need feedback to identify problems and improve the design. An online expert system is available to instructors and students to provide this immediate feedback. This online system is available at: http://JerryPost.com/DBDesign. This appendix uses the DB Design system to highlight a graphical approach to designing a database. However, even if you do not use the DB Design system, this appendix provides a useful summary of how to approach database design.

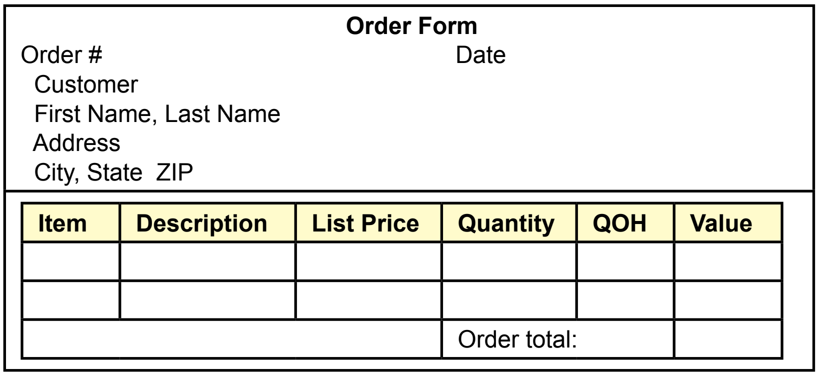
The design process in this appendix is illustrated with a generic sales order form. If you are unfamiliar with order forms and the entire ordering process, check out the Universal Business Language on the Oasis Web site at http://docs.oasis-open.org/ubl/cd-UBL-1.0. This organization has defined a generic purchasing process that applies to any organization. The goal is to create a standard means of transferring data among businesses. The specification includes several XML schema definitions. Because the goal is to create a generic format, the specification is considerably more complex than the example presented here, but the document also defines the common terms, processes, and business rules.

Sample Problem: Customer Orders

It is easiest to understand database design and the DB Design system by following an example. Customer orders are a common situation in business databases, so

Figure 2.1A

Typical order form. Each order can be placed by one customer but can contain multiple items ordered as shown by the repeating section.



Order Form

Order # Date

Customer
First Name, Last Name
Address
City, State ZIP

Item	Description	List Price	Quantity	QOH	Value
			Order total:		

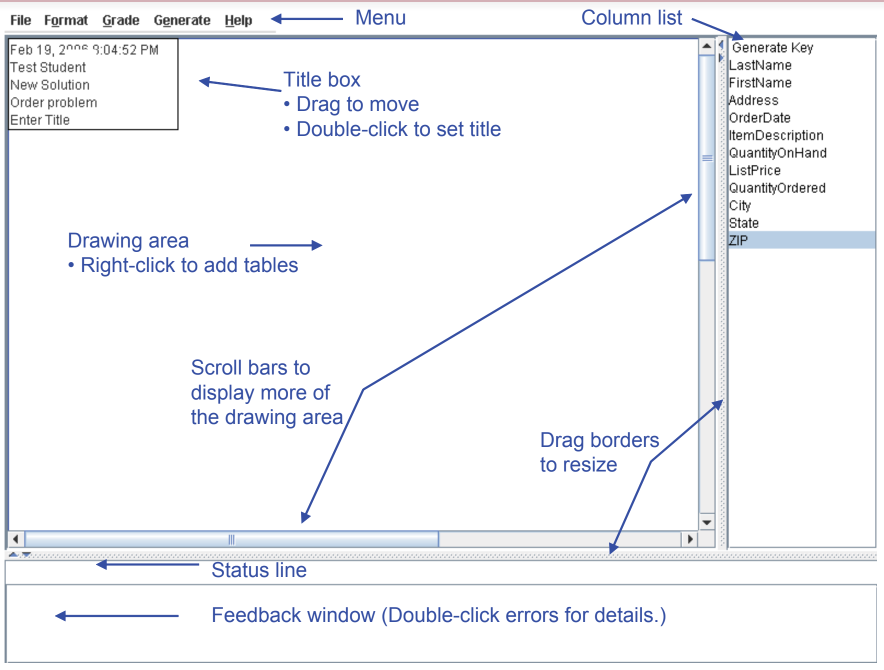


Figure 2.2A

DB Design screen. Once you log in, use the menu option File/Open to choose the Order Problem. The Help menu has an option to View the Problem. The right-hand window contains a list of the available columns that will be placed into tables. Selecting the Grade menu option generates comments in the feedback window.

consider the simple sales order form displayed in Figure 2.1A. The layout of the form generally provides information about the business rules and practices. For example, there is space for only one customer on the order, so it seems reasonable that no more than one customer can participate in an order. Conversely, the repeating section shows multiple rows to allow several items to be ordered at one time. These one-to-many relationships are important factors in the database design.

Getting Started: Identifying Columns

One of the first steps in creating the database design is to identify all of the properties or items for which you need to collect data. In the example, you will need to store customer first name, last name, address, and so on. You will also need to store an order number, order date, item description, and more. Basically, you identify each item on the form and give it a unique name. Note that some items can be easily computed and will not need to be stored. For instance, value is list price times quantity, and the order total is the sum of the value items. In a business environment, you will have to identify these items yourself and write them down. The DBDesign system handles this step for you and displays all of the columns in a list.

As shown in Figure 2.2A, after you have opened a problem, the DB Design system provides you with a list of items from the form. This list is presented in the

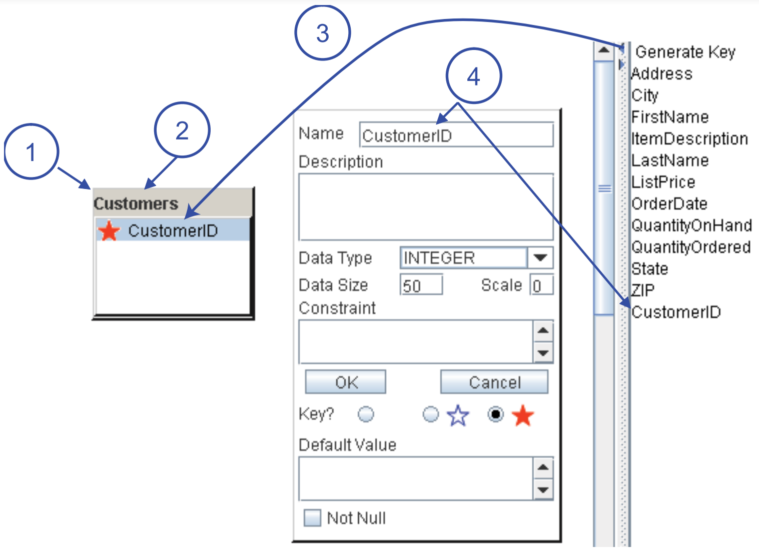


Figure 2.3A

Adding a table and key. (1) Right click and select Add table. (2) Enter a new name (Customer) in the title box. (3) Drag the Generate Key item onto the table. (4) Enter a new name (CustomerID) in the edit box and click the OK button.

right-hand column. The list of columns is the foundation for the database design. Your job is to create tables and then select the columns that belong in each table. You can rename the columns by right clicking the column name and selecting the Rename option, but be careful to use names that represent the data. Also, key columns should have unique names. To get a better grasp of the columns available, you can sort the list by right clicking the list and selecting the Sort option. You can also double-click a column to see more details about it, including a brief description. If two columns have the same name (such as LastName), you will have to look at the description to see which entity it refers to (such as employee or customer).

Creating a Table and Adding Columns

The main objective is to create tables and specify which columns belong in each table. It is fairly clear that the sale order problem will need a table to hold customer data, so begin by right-clicking the main drawing window and selecting the option to add a table. The system enters a default name for the table, but you should change it by typing in a new name. Later, you can change the name by right-clicking the name and selecting the rename option. For this demonstration, enter "Customer" to provide the new name.

Each table must have a primary key—one or more columns that uniquely identify each row in the table. Looking at the order form and the column list you will not see a column that can be used as a primary key. You might consider using the customer phone number, but that presents problems when customers change their numbers. Instead, it is best to generate a new column called CustomerID. To ensure each customer is given a different ID value, the data for this column will

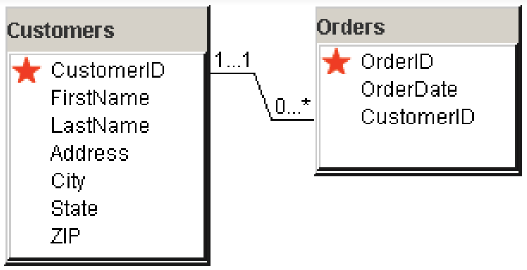


Figure 2.4A

Two tables. Each table represents a single entity, and all columns are data collected for that entity. The Orders table contains the CustomerID, which provides a method to obtain the matching data in the Customer table. Build the Customer table first, followed by the Orders table and the relationship line will probably be added automatically for you.

be generated by the DBMS whenever a new customer is added. To create a new key column that is generated by the DBMS, drag the Generate Key item from the column list and drop it on the Customers table. The column-edit form will pop up with a temporary name. Type a new name for the column (CustomerID). You can enter a description if you want. Click the OK button when you are ready. Notice that CustomerID will be displayed in the Customers table and as a new column in the column list. Also, notice in Figure 2.3A that the CustomerID is marked with a filled red star to indicate that it is part of the primary key in the Customers table. You can edit a column name and description later by double-clicking the column name.

A star in the DB Design system indicates that a column is part of the primary key for a table. But, there are two types of stars: (1) a filled red star, or (2) an open blue star. Both indicate that the column is part of the primary key. The filled red star additionally notes that the key values are generated in that table whenever a row is added. Because generated values must always be unique, any table that contains a generated key column can only have that column as the primary key. You can change the key attribute by opening the column-edit form or by double-clicking the space in front of a column name. As you double-click the space, the key indicator will rotate through the three choices: blank (no key), blue star (key), red star (generated key).

Now that the table and primary key are established, you can add other columns to the table. But which columns? The Customers table should contain columns that identify attributes specifically about a customer. So, find each column that is strictly identified by the new primary key CustomerID and drag it onto the Customers table.

Relationships: Connecting Tables

Almost all database problems will need multiple tables. In the sales order problem, it is fairly clear that the database design will need an Orders table. Add a new table, name it "Orders," and generate a key for OrderID. Once again, you need to identify the columns that belong in the Order table. Looking at the Order form, you should add the OrderDate column. Notice that the order form also contains

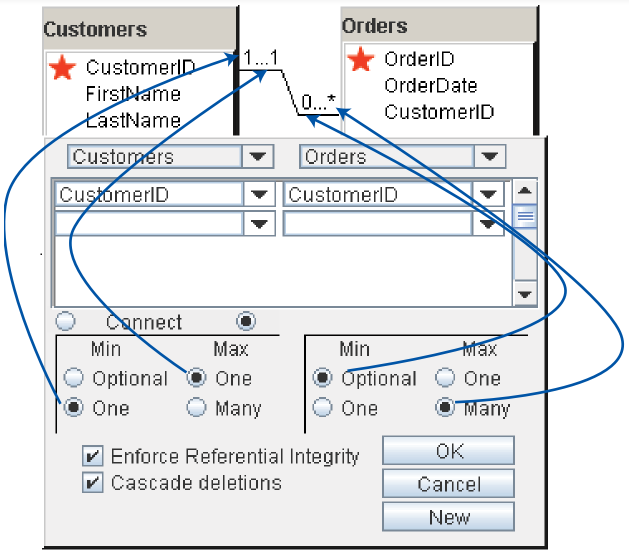


Figure 2.5A

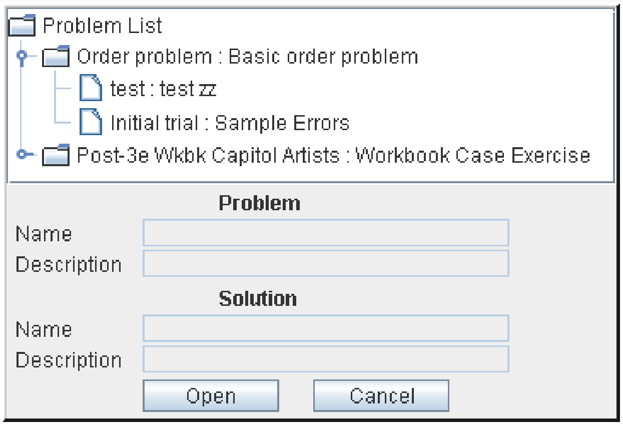
Relationships. Drag the CustomerID column from the Customer table and drop it onto the CustomerID column in the Orders table. Then set the minimum and maximum values for each side of the relationship. An order must have exactly one customer, and a customer can place from zero to many orders.

customer information. But it would seem to be a waste of effort to require clerks to enter a customer's name and address for every order. Instead, you need to add only the CustomerID in the Order table.

When you add the CustomerID to the Orders table, as shown in Figure 2.4A, the system will create a relationship back to the Customers table. It will even try to get the multiplicity correct. Actually, your instructor can turn off the automatic relationship and the multiplicity options, so there is a small chance that you will have to create the relationship by hand. You can delete a relationship by right-clicking the sloping line and choosing the Delete option. You edit a relationship by double-clicking the connecting line. You create a new relationship by dragging a column from one table and dropping it onto the matching column in a second table.

Remember that CustomerID will not be a primary key in the Order table, because for each order, there can be only one customer. If it were keyed, you would be indicating that more than one customer could take part in an order.

You often need to edit the multiplicity values when you create a relationship. If all key columns are specified correctly, the system does a good job of setting the values automatically. But, read that "if" condition again and you quickly realize that you will have to edit multiplicity values for many of your relationships. Double-click the connection line to open the relationship edit window. Figure 2.5A shows how the selections are displayed. Your form might be slightly different from the one shown because the form is dynamic. It looks at the diagram and displays the left-most table on the left. If your layout is different, the table names will change positions to match your diagram. Every relationship has four values: a minimum and maximum on each end of the relationship. These values are set with



Problem List
Order problem : Basic order problem
test : test zz
Initial trial : Sample Errors
Post-3e Wkbk Capitol Artists : Workbook Case Exercise

Problem

Name
Description

Solution

Name
Description

Figure 2.6A

Opening solutions. You can save multiple versions or solutions for any problem. To open a saved solution, you have to expand the list by clicking the handle icon in front of the problem name.

the option buttons. In this case, an order can be placed by exactly one customer, so the minimum customer value is one and the maximum value is also one. On the other side of the relationship, each customer can place from zero to many orders. Some might argue that if a customer has not placed any orders, then he or she is only a potential customer, but the difference is not critical to the database design.

The relationship-edit form has a couple of other options. The Connect option box is useful when two tables are displayed vertically (above and below, instead of the left and right used here). It enables you to specify the preferred side for the relationship line (left or right). Look at the boxes containing the CustomerID values, and you can use the drop-down lists to change the column matches if you made a mistake when you dropped a column while building the relationship. You can also create a relationship that connects tables on multiple columns by moving to a new row and choosing the matching columns. For more complex cases, you can click the New button to create multiple relationships between two tables. For example, you might need to connect a City.CityID column to both an Order. DeliveryCity and an Order.BillingCity column. These would be two separate, independent relationships. None of these more complicated options are needed for this example, but it is good to know they exist.

Saving and Opening Solutions

Be sure to save your work as you go. If you wait too long, the Internet connection will time-out and you might lose your changes. In most cases, if you lose your session, you can log in and try again. The first time you save your solution, you will be asked to give it a name and a brief description. You can use File/Save to create copies with different names—enabling you to save multiple versions of your work. Generally, you will only need this approach for complex problems.

Even if you save only one version of your solution, you need to understand the File/Open box shown in Figure 2.6A. First, note that you can resize the box by dragging its lower right-hand corner. This trick is useful when you have a long list of problems or solutions. Second, the list is stored and displayed in a tree hi-

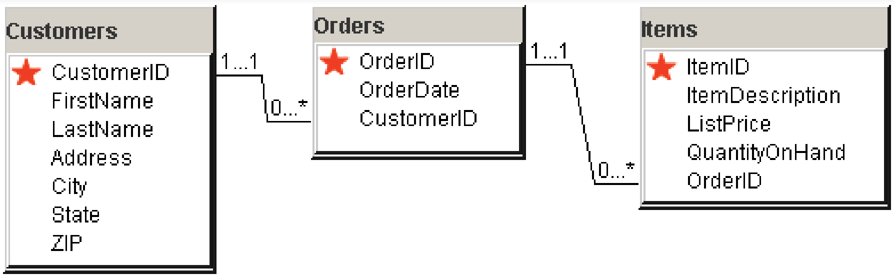


Figure 2.7A

Creating errors. To demonstrate a potential problem, add the OrderID column to the Items table and then link it to the Orders table.

erarchy that starts by listing each problem available to you. If you double-click a problem (or select one and click the Open button), you will get a blank problem where you start over. Sometimes this approach is useful if you really messed up an earlier solution. In most cases, you will want to click the handle icon in front of the problem name to open the list of solutions you saved for that problem. You can open any of the solutions you have saved.

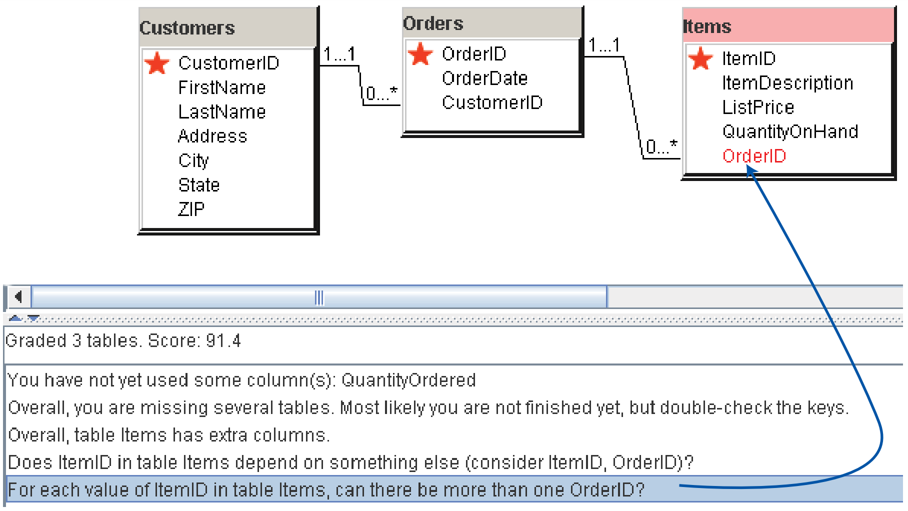
Grading: Detecting and Solving Problems

You will repeat these same steps to create the database design: add a table, set the primary key, add the data columns, and link the tables. The DB Design system makes the process relatively easy, and you can drag tables around to display them conveniently. You can save your work and come back at a later time to retrieve it and continue working on the problem. However, you still do not know if your design is good or bad.

Consider adding another table to the sample order problem. Add a table for Items and generate a new key column called ItemID. Add the columns for Item-Description, ListPrice, and QuantityOnHand. The problem you face now is that you need to link this new table with the Orders table. But, so far, they do not have any related columns. So, as an experiment, try placing the OrderID column into the Items table and build a relationship from Items to Orders by linking the Orde-rID columns, as shown in Figure 2.7A.

At any time, you can ask the server to grade the current design to see if there are problems. In fact, it is a good idea to check your work several times as you create the tables, so you can spot problems early. Use the Grade option on the menu to Grade and Mark the diagram. This option generates a list of comments in the bottom window. The Grade to HTML option generates the same list organized by tables in a separate window. Both options automatically save your work, so you do not need to worry about saving your solution as long as you continue to grade it.

As shown in Figure 2.8A, when you grade this problem, you get a reasonably good score (88.1). However, there are several important comments. When you select (click) a comment, the system highlights the error in the diagram whenever possible. Notice the first grade comment about the unused column. If you had others, they would also be listed in that message. Clicking that message will cause all of the column names to be highlighted in the right-hand side list—making them easy to find.



Graded 3 tables. Score: 91.4

You have not yet used some column(s): QuantityOrdered
Overall, you are missing several tables. Most likely you are not finished yet, but double-check the keys.
Overall, table Items has extra columns.
Does ItemID in table Items depend on something else (consider ItemID, OrderID)?
For each value of ItemID in table Items, can there be more than one OrderID?

Figure 2.8A

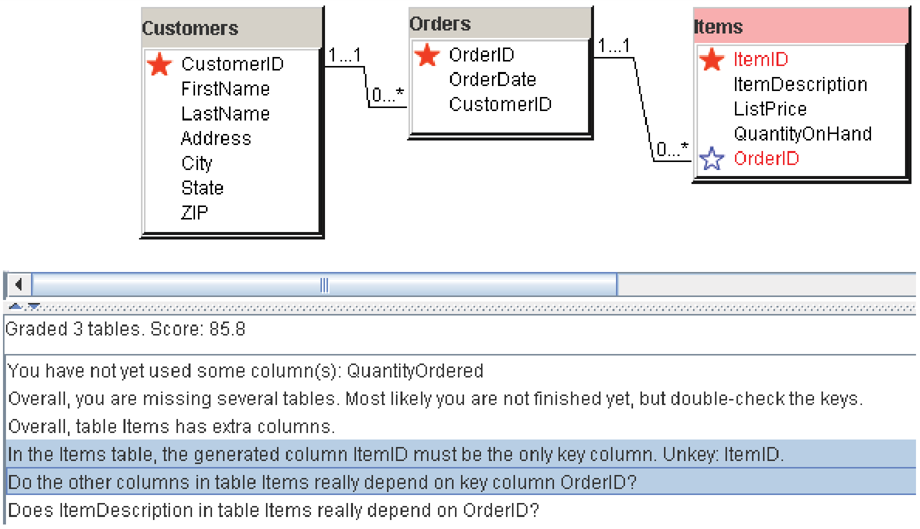
Grading the exercise. Click a comment to highlight the table and column causing problems. In this case, each ItemID can appear in many Orders, but OrderID is not part of the key. Double-click an error message to see more information about the error.

You use the error messages to help improve the design. In this case, most of the comments indicate there is a problem with the Items table. In particular, the OrderID column is presenting a problem. The first couple of questions ask whether the key values are correct. The question highlighted at the bottom is important because it tells you how to solve the problem. It is asking whether an item can be sold on more than one order. Currently, since OrderID is not part of the key, any item can be sold only one time. This assumption is extremely restrictive and probably wrong. The system is telling you that you need a table where both ItemID and OrderID are key columns.

At this point, you really should stop and think about this entire section of the design. But, see what happens if you just look at the one comment and leap ahead. Just make OrderID a key along with ItemID. Figure 2.9A shows the result of this change. First, notice that the score actually decreased! The DB Design system is still pointing out problems with the keys. In particular, note that ItemID was created as a generated key, so it is always guaranteed to be unique. If that is true, then you would never need a second key column in the same table. As a side note, observe that you can use the Ctrl+click approach to highlight several error messages at once. The basic problem is that you cannot include the OrderID column in the Items table.

The solution is to realize that a relational database cannot support a direct many-to-many relationship between two tables (Orders and Items). Instead, you must insert a new table between the two. In this case, call it an OrderItems table. Then be sure to add the key columns from both of the linked tables (OrderID and ItemID). As shown in Figure 2.10A, add both relationships as one-to-many links.

As indicated by the score, this four-table solution is the best database design for the typical order problem. The Customers table holds data about each customer.



Graded 3 tables. Score: 85.8

You have not yet used some column(s): QuantityOrdered
Overall, you are missing several tables. Most likely you are not finished yet, but double-check the keys.
Overall, table Items has extra columns.
In the Items table, the generated column ItemID must be the only key column. Unkey: ItemID.
Do the other columns in table Items really depend on key column OrderID?
Does ItemDescription in table Items really depend on OrderID?

Figure 2.9A

Trying to fix the problems. You could try making OrderID part of the key, but notice that the score decreased, so the fix actually made the situation worse. The problems with the OrderID and the relationship have not been solved. You can use Ctrl+Click to highlight several errors at the same time.

The Items table contains rows that describe each item for sale. The Orders table provides the order number, date, and a link to the customer placing the order. The OrderItems table represents the repeating section of the order form and lists the multiple items being purchased on each order. You should verify that all of the data items from the initial form appear in at least one of the tables.

Specifying Data Types

You need to perform one additional step before the database design is complete. Eventually, this design will be converted into database tables. When you create the tables, you will need to know the type of data that will be stored in each column. For example, names are text data, and key columns are often 32-bit integers. Make sure that all dates and times are given the Date data type. Be careful to check when you need floating point versus integer values: use single or double depending on how large the maximum value will be. Figure 2.11A shows that you set the data type by double-clicking to open the column-edit form.

The default value is text since it is commonly used. Consequently, many columns such as customer name will not need changes. Although there are standard names for data types, every DBMS uses its own terms. You can control which terms are displayed by setting the target DBMS under the Generate menu command. This choice makes it easier for you to choose the exact data type for a particular DBMS. Internally, the DB Design system assigns a generic definition. You can use the generic definitions, the SQL standard names, or switch to one of the common DBMSs to fine-tune the choice.

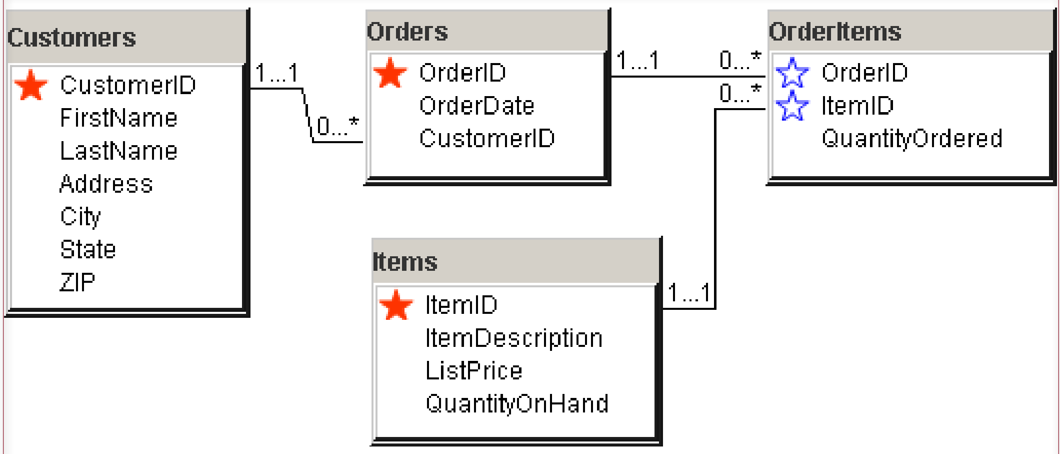


Figure 2.10A

A solution. Add the intermediate table OrderItems and include keys from both tables (OrderID and ItemID). Use one-to-many relationships to link it to both tables. Notice the difference in the key indicators. The solid red star shows where a key value is generated.

You can also set default values and constraint rules for the form. Default values are fairly standard, but the syntax of constraint rules depends heavily on the specific target DBMS. These options are provided primarily for when you want to generate complete table descriptions. Until you gain experience with your target DBMS, you should leave them blank.

Generating Tables

Once you are satisfied with your design, you can use the system to help create the tables in your DBMS. Almost all DBMSs support the SQL CREATE TABLE command. When you ask DB Design to generate tables, it writes a SQL script that you can run to generate the tables within your DBMS. Note that DB Design does not actually create the tables inside itself. You need to copy the SQL script and run it on your database server.

Use the Generate/Generate Tables menu command to open a new browser window with the SQL script. As shown in Figure 2.12A, you can scroll to the bottom of the window and change some of the options. For example, you might want to change the target DBMS. When you are satisfied with the script, click within the script window, press Ctrl+A to select all of the lines, and Ctrl+C to copy the text. Open a text editor or a script edit window in your DBMS management tool. Paste (Ctrl+V) the script and save it or execute it. If necessary, you can edit the script to fine-tune some DBMS-specific options.

If you are using Microsoft Access, read the notes at the top of the script file. While Access supports the CREATE TABLE command it does not support script files (at least through Office 2010). Consequently, you can only run one CREATE TABLE command at a time. Also, you need to hand-edit all of the final relationships inside Access because it does not support the cascade options.

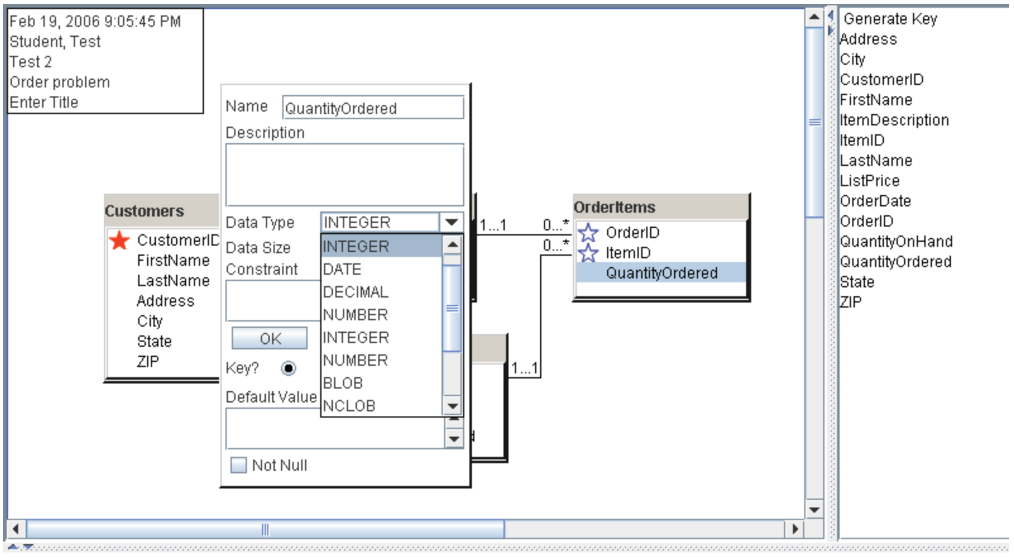


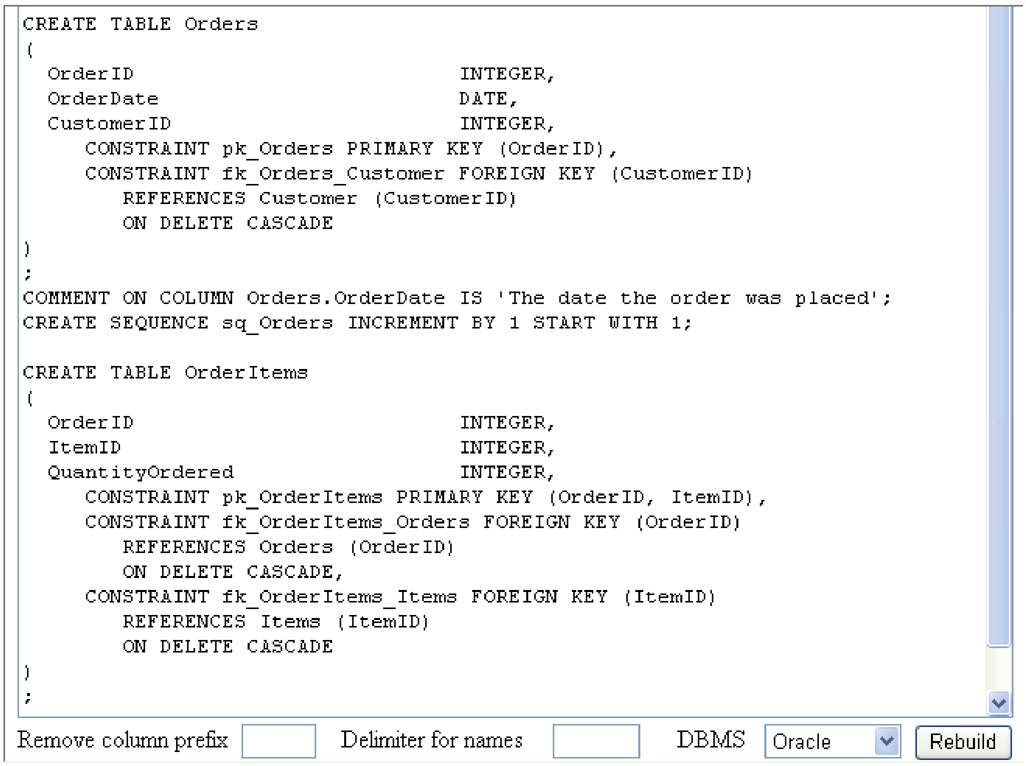
Figure 2.11A

Data types. Double-click a column name to open the edit window. Set the data type. The default is Text, so you do not have to change common columns like the customer name. You can also add a description and a default value setting. The Constraint setting has to match the format of the target DBMS.

The Generate form contains some additional options. The name delimiter is straightforward. You are not allowed to use reserved words or characters in table and column names. For instance, column names cannot include spaces. However, current DBMSs will allow you to violate these rules if you enclose the name in special delimiters. The delimiters vary by DBMS. For example, Microsoft Access and SQL Server use square brackets, while Oracle uses double quotes. If you enter a delimiter in the box (such as [or "), the generator will apply it to all table and column names. Why does the generator not apply delimiters by default? The answer is because delimiters sometimes have other consequences. In particular, if you use the double-quote delimiter (") in Oracle, the table and column names become case-sensitive. From that point, every time you reference a table or column name, you are required to enclose it in quotation marks. When you type SQL statements by hand, it is annoying to type all of those quotation marks, so it is easier to use well-formed names and avoid the delimiter completely.

Although it is not shown here, a checkbox option has been added to exclude the descriptive comments. Most of the time, you should keep the comments as documentation of the database design. However, if the comments are excessive or intrusive, you can tell the generator to leave them out.

The prefix option needs more explanation than the others. It is included because of the way DB Design works. In particular, since DB Design displays all of the columns in one list, it is helpful to ensure that the names are unique. For instance, if you see several columns called LastName, it is not immediately clear which entity or table is referenced. Consequently, it is helpful to add a prefix to the names to make them unique. For instance, you could have Emp_LastName and Cust_LastName. However, when you generate the tables in the DBMS, the



```
CREATE TABLE Orders
(
  OrderID                          INTEGER,
  OrderDate                        DATE,
  CustomerID                       INTEGER,
     CONSTRAINT pk_Orders PRIMARY KEY (OrderID),
     CONSTRAINT fk_Orders_Customer FOREIGN KEY (CustomerID)
        REFERENCES Customer (CustomerID)
        ON DELETE CASCADE
)
;
COMMENT ON COLUMN Orders.OrderDate IS 'The date the order was placed';
CREATE SEQUENCE sq_Orders INCREMENT BY 1 START WITH 1;

CREATE TABLE OrderItems
(
  OrderID                          INTEGER,
  ItemID                           INTEGER,
  QuantityOrdered                  INTEGER,
     CONSTRAINT pk_OrderItems PRIMARY KEY (OrderID, ItemID),
     CONSTRAINT fk_OrderItems_Orders FOREIGN KEY (OrderID)
        REFERENCES Orders (OrderID)
        ON DELETE CASCADE,
     CONSTRAINT fk_OrderItems_Items FOREIGN KEY (ItemID)
        REFERENCES Items (ItemID)
        ON DELETE CASCADE
)
;
```

Remove column prefix: (empty) Delimiter for names: (empty) DBMS: Oracle

Figure 2.12A

Generate Tables. Choose the Generate/Generate Tables menu option to create a set of SQL commands that can be run on your DBMS to build the tables created in the diagram. You can choose the target DBMS before running the command or select it on the generated page. Use Ctrl+A to select the entire text in the window, then open a text editor or a SQL editor and paste the commands with Ctrl+V.

column will gain the context of the table and the prefix is superfluous and just something extra to type (such as Employee.Emp_LastName). If you adopt a consistent naming convention, the generator can automatically remove the prefix. The easiest approach is to use an abbreviation of the entity followed by an underscore (e.g., Emp_Address, Cust_Address). When you enter the underscore character (_) into the prefix box and generate the SQL script, the generator will examine every column name and remove all characters that appear before the first underscore (and the underscore).

Chapter 3

Data Normalization

Chapter Outline

What You Will Learn in This Chapter

- Why is database design important?
- What is a table and how do you choose keys?
- What are the fundamental rules of database normalization?
- How do you begin analyzing a form to create normalized tables?
- How do you create a design in first normal form?
- What is second normal form?
- What is third normal form?
- What problems exist beyond third normal form?
- How does a database record constraints?
- How do business rules change the database design?
- What problems arise when converting a class diagram to normalized tables?
- What tables are needed for the Sally's Pet Store?
- How do you combine tables from multiple forms and many developers?
- How do you record the details for all of the columns and tables?

A Developer's View

Miranda: That was actually fun. I learned a lot about the company's procedures and rules. I think I have everything recorded properly on the class diagram; along with some notes in the data dictionary.

Ariel: Great! We should go to the concert tonight and celebrate.

Miranda: I could use a night off. Maybe giving my brain cells a rest will help me figure out what to do next.

Ariel: What do you mean? How much longer do you think the project will take?

Miranda: That's the problem. I put all this time in, and I don't really have a start on the application at all.

Ariel: Well, isn't the data the most important aspect to building a database application? I heard that database systems are touchy. You have to define the data correctly the first time; otherwise, you will have to start over.

Miranda: Maybe you're right. I'll take the night off; then I'll study these rules to see how I can turn the class diagram into a set of database tables.

Getting Started
Refine your table definitions by double-checking the primary keys. Then examine each non-key column to ensure that it depends on the whole key and nothing but the key. Each table should represent a single concept and all business rules should be explicit. There can be no hidden dependencies.

Introduction

Why is database design important? A database management system is a powerful tool. It provides many advantages over traditional programming and hierarchical files. However, you get these advantages only if you design the database correctly. Recall that a database is a collection of tables. The goal of this chapter is to show you how to design the tables for your database.

The essence of data normalization is to split your data into several tables that will be connected to each other based on the data within them. Mechanically, this process is not very difficult. There are perhaps four rules that you need to learn. On the other hand, the tables have to be created specifically for the business or application that you are dealing with. Therefore, you must first understand the business, and your tables must match the rules of the business. So the challenge in designing a database is to first understand how the business operates and what its rules are. Some of these rules were hinted at in Chapter 2 with the focus on relationships. Business relationships (one-to-one and one-to-many) form the foundation of data normalization. These relationships are crucial to determining how to set up your database. These rules vary from firm to firm and sometimes even depend on which person you talk to in the organization. So when you create your database, you have to build a picture of how the company works. You talk to many people to understand the relationships among the data. The goal of data normalization is to identify the business rules so that you can design good database tables.

By designing database tables carefully, you (1) save space, (2) minimize duplication, (3) protect the data to ensure its consistency, and (4) provide faster transactions by sending less data. One method for defining database tables is to use the graphical approach presented in Chapter 2 and build a class diagram. A related method is to collect the basic paperwork, starting with every form and every report you might use. Then take apart each collection of data and break it down into respective tables. Most people find that a combination of both approaches helps them find the answer. However, the discussion will begin by describing the two methods separately.

Two-Minute Chapter

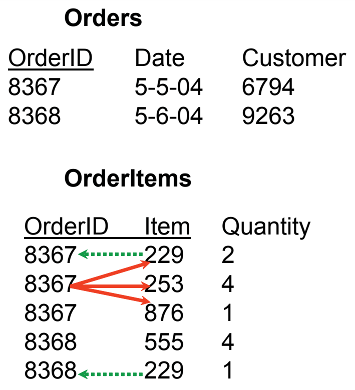
Database design takes practice and experience. In the end, the design encapsulates the rules and relationships in the underlying business problem. Chapter 2 emphasized that tables represent business objects and each table must have a primary key. Composite keys (multiple columns) in a table indicate many-to-many relationships. Tables are linked together by the data in keys. For example, The Sale table has a primary key of SaleID but CustomerID is a foreign key column in the Sale table. This approach saves space and prevents other problems because only the CustomerID number is stored for each Sale. Instead of repeating all Customer data for every Sale, the number refers back to the detailed information in the Customer table.

When deciding which columns belong in each table, the three primary rules of normalization are: (1) each entry must be atomic or single-valued not repeating, (2) each non-key column must depend on the whole key, and (3) each non-key column must depend on nothing but the key. Another general way to look at the problem is to note that there can be no hidden dependencies. If some business relationship or rule exists, it needs to be defined as its own table. The hard part is identifying these business rules. Some of them can be determined from existing forms and data. Others have to be elicited through interviews and discussions with managers.

This approach leads to tables that can efficiently store data with minimal problems. However, you have to carefully evaluate every table, every key, and every column. Watch for many-to-many relationships, and understand the concept of *dependence*.

Tables, Classes, and Keys

What is a table and how do you choose keys? Chapter 2 focuses on identifying the business classes and associations. Now, these classes need to be more carefully defined so they can be converted into database tables. Of course, as you modify the tables, you will also update the class diagram. The relationships among the classes are critical to determining the final form of the tables. These relationships are also expressed in terms of the primary keys of the tables. Remember that a primary key consists of a collection of columns that uniquely identify each row. Since the key must be guaranteed to always be unique, it is common to create a new key column that holds generated keys. But, in many cases, you will use multiple columns to make up the primary key. These situations are important enough to require a detailed explanation.



Orders

<u>OrderID</u>	Date	Customer
8367	5-5-04	6794
8368	5-6-04	9263

OrderItems

<u>OrderID</u>	<u>Item</u>	Quantity
8367	229	2
8367	253	4
8367	876	1
8368	555	4
8368	229	1

Figure 3.1

Composite keys. OrderItems uses a composite key (OrderID + Item) because there is a many-to-many relationship. Each order can contain many items (shown by the solid arrows). Each item can show up on many different orders (dotted arrows).

Composite Keys

In many cases, as you design a database, you will have tables that will use more than one column as part of the primary key. These are called **composite keys**. You need composite keys when the table contains a one-to-many or many-to-many relationship with another table.

As an example of composite keys, look at the OrderItems table in Figure 3.1. These two tables are common in business and they form a **master-detail** or parent-child relationship. The Orders table is straightforward. It has one column as a primary key, where you created the OrderID. This table contains the basic information about an order, including the date and the customer. The OrderItems table has two columns as keys: OrderID and Item. The purpose of the OrderItems table is to show which products the customers chose to buy. In terms of keys the important point is that each order can contain many different items. In the example OrderID 8367 has three items. Because each order can have many different items, Item must be part of the key. Reading the table description from left to right you can say that each OrderID may have many Items. The "many" says that Item must be keyed. What about the other direction in the OrderItems table? Do you really need to key OrderID? The answer is yes because the firm can sell the same item to many different people (or to the same customer at different times). For example, Item 229 appears on OrderIDs 8367 and 8368. Because each item can appear on many different orders, the OrderID must be part of the primary key. For comparison, reconsider the Orders table in Figure 3.1. Each OrderID can have only one Customer, so Customer is not keyed.

To be sure you understand how keys and relationships interact, look again at the OrderItems table. Looking at the ItemID column, ask yourself: For each OrderID, can there be one or many ItemIDs? If the answer is many, then ItemID must be keyed (underlined). Now, look at OrderID and ask yourself, Can an ItemID appear on one or many orders? Again, the answer is many, so OrderID must also be keyed.

Look at the CustomerID column in the Order table and ask, For each order, can there be one or many customers? The common business rule says there is only

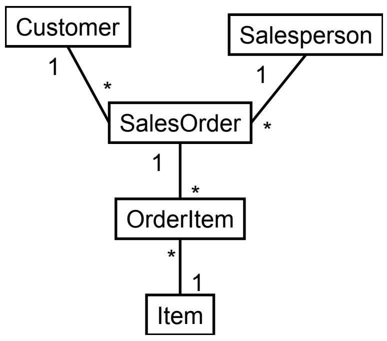


Figure 3.2

A small class diagram for a basic order system. The numbers indicate relationships. For instance, each customer can place many orders, but a given order can come from only one customer.

one customer per order, so CustomerID is not part of the primary key. On the other hand, because CustomerID is a primary key within the Customer table, it is known as a **foreign key** in the Order table. Think of it as a foreign dignitary visiting a different country (table). It is required in the Order table because it serves as a link to the rest of the customer data in the Customer table; but it does not have to be a key (king) in that table.

To properly normalize the data and store the data as efficiently as possible, you must identify keys properly. Your choice of the key depends on the business relationships, the terminology in the organization, and the one-to-many and many-to-many relationships within the company.

Surrogate Keys

It can be difficult to ensure that any real-world data will always generate a unique key. Consequently, you will often ask the database system to generate its own key values. These **surrogate keys** are used only within the database and are often hidden so users do not even know they exist. For example, the database system could assign a unique key to each customer, but clerks would look up customers by conventional data such as name and address. Surrogate keys are especially useful when there is some uncertainty with the business key. Think about SalesID or PurchaseOrderID which need to be assigned at the time of each sale or purchase. How can a person create these to guarantee they are unique? Numbers such as

Figure 3.3

Table notation. Column details are easier to see in a simple listing of the tables. This list is also useful when the tables are entered into the database.

Customer(<u>CustomerID</u>, Name, Address, City, Phone)
Salesperson(<u>EmployeeID</u>, Name, Commission, Datehired)
SaleOrder(<u>OrderID</u>, OrderDate, CustomerID, EmployeeID)
OrderItem(<u>OrderID</u>, <u>ItemID</u>, Quantity, SalePrice)
Item(<u>ItemID</u>, Description, ListPrice)

1. Each cell in a table contains atomic (single-valued) data.
2. Each non-key column depends on all of the primary key columns (not just some of the columns).
3. Each non-key column depends on nothing outside of the key columns.

Figure 3.4

The three main rules for data normalization. Essentially, each table has to accurately represent the business definitions. Each table represents a single entity and the keys accurately identify the entity and represent the one-to-many relationships among the attributes.

CustomerID and EmployeeID could be defined by the marketing or HRM departments, but it is simpler to just let the DBMS create unique values when they are needed.

The use of surrogate keys can be tricky when the database becomes large. With many simultaneous users, creating unique numbers becomes more challenging. Additionally, several performance questions arise involving surrogate keys in large databases. For example, a common method of generating a surrogate key is to find the largest existing key value and increment it. But what happens if two users attempt to generate a new key at the same time? A good DBMS handles these problems automatically.

Microsoft Access uses the **autonumber** data type to generate unique numbers for key columns. Similarly, SQL Server uses the Identity data type. Oracle has a SEQUENCES command to generate unique numbers, but it operates differently than the Microsoft approaches. As a programmer, you generate and use the new values when a new row is inserted—the process is not automatic (but you can automate it with a couple lines of code). There are advantages and drawbacks to both approaches. The biggest difficulty with the Microsoft approach is that it is sometimes difficult to obtain the new value that was generated when a row is inserted. The drawback to the Oracle approach is that you must ensure that all users and developers use the proper number generation commands throughout the application. Additionally, both approaches can cause problems when transferring data—particularly to other database systems.

Generated numbers are even trickier in distributed databases—where new numbers must be generated in multiple locations. One approach is to assign different ranges to each location so each location generates a different type of number. A second approach is the **globally-unique identifier (GUID)** which is essentially a very large random number. Microsoft software has tools for generating 128-bit GUIDs. Oracle also has functions for generating GUIDs. GUIDs are rarely sequential and they are large numbers. The point is that although they are useful for creating key values, think of generated keys as random values, and you almost always want to hide these numbers from the users.

Notation

A detailed class diagram can describe each table and include all properties within each class and marked key columns. The advantage to using class diagrams is that they highlight the associations among the classes. Additionally, some people understand the system better with a visual representation. Figure 3.2 shows a simple example class diagram, but it leaves out the properties.

The drawback to class diagrams is that they can become very large. By the time you get to 30 classes, it is hard to fit all the information on one page. Also, many of the association lines will cross, making the diagram harder to read. CASE tools help resolve some of these problems by enabling you to examine a smaller section of the diagram.

However, you can also use a shorter notation, as shown in Figure 3.3. The notation consists of a straight listing of the tables. Each column is listed with the table name. The primary keys are underlined and generally listed first. This notation is easy to write by hand or to type, and it can display many tables in a compact space. However, it is hard to show the relationships between the tables. You can draw arrows between the tables, but your page can become messy.

Designers frequently create both the class diagram and the list of tables. The list identifies all of the columns and the keys. The class diagram shows the relationships between the tables. The class diagram can also contain additional details, such as existence constraints and minimum requirements.

Database Normalization: Atomic Values and Dependency

What are the fundamental rules of database normalization? Database researchers have shown that if tables are not designed carefully, several serious problems can arise. These problems can be avoided by following some basic rules. The primary rules are written as the first three normal forms. Figure 3.4 lists the most important rules. These rules are explained in detail in the following sections. You need to understand each rule in detail because you use them to improve your designs and ensure that the tables you create accurately reflect the business rules. One way to think about the rules is that every database table represents the business rules so that there are no hidden relationships. Everything is spelled out correctly in the tables.

When you first read the rules, they seem slightly confusing because they rely on some special terms. In fact, understanding the rules basically comes down to understanding two specific concepts: atomicity and dependency. These terms are described in this section and the following sections describe how to apply them to real-world problems.

Atomic Data Values

The first rule is the easiest to understand and one of the most important. However, sometimes it can be tricky to apply. A table cell contains an **atomic** value if there is only a single non-repeating item. Consider a Customer table that shows up in most business databases. A Name column in a Customer table can contain only one name on each row. That example certainly seems simple. Why would anyone ever try to store multiple names in one cell? People have only one name anyway? Wait a second. What about first name and last (family) name? If you enter “John Doe” into a Name column, is that one name or two? The answer is that it depends on how the data will be used. If you want to sort the rows based on last name and then first name, you really need to create two columns: LastName and FirstName instead of just the Name column. Yes, you could write a special function that would split a single name into its two components. But, needing to write special functions to extract data from a cell is a major sign that your database is violating the rule of atomic data. Now, if the users always look at the entire name as a single thing, it is fine to store the entire name in one column. This example highlights one of the most important things you need to learn: the database design depends

CustomerID	LastName	FirstName	Phone	Fax	CellPhone
15023	Jones	Mary	222-3034	222-4094	223-0984
63478	Sanchez	Miguel	030-9693	403-4094	
94552	O'Reilly	Madelline	849-4948	292-3332	139-3831
45791	Stein	Marta	294-4421		
49004	Brise	Mer	764-5103		

Figure 3.5

Atomic values for phone numbers. Is phone number atomic (single-valued) or repeating? You might add a column for each possible type of phone number. But how many customers have each type of phone and what if more types are needed in the future?

heavily on the business rules and assumptions. In fact, the design is a model of the business because the tables, columns, and keys reflect the business rules.

The Name column is relatively easy. In practice, most designers do split Name into LastName and FirstName columns. Now, look at some other columns in the potential Customer table. Many companies want to store the customer's phone number, so you can add a Phone column to the table. But once again, you have to ask: Can a customer have more than one phone number? Twenty years ago, this question was easy to answer as "no," customers have only one phone number. Today, you might want to add a cell phone number or business number. Figure 3.5 shows how you might add three types of phone numbers to a table. It certainly appears that each cell contains a single value. However, notice that customers might not have all three numbers. Also, what will you do if a customer has a fourth or fifth phone number?

Figure 3.6

Repeating values for phone numbers. Each customer can have from one to many phone numbers. This table would be a bad design because it would cause problems with retrieving or editing an individual phone number.

CustomerID	LastName	FirstName	Phone
15023	Jones	Mary	222-3034 222-4094 223-0984
63478	Sanchez	Miguel	030-9693 403-4094
94552	O'Reilly	Madeline	849-4948 292-3332 139-3831 339-4040
45791	Stein	Marta	294-4421
49004	Brise	Mer	764-5103

CustomerID	LastName	FirstName
15023	Jones	Mary
63478	Sanchez	Miguel
94552	O'Reilly	Madeline
45791	Stein	Marta
49004	Brise	Mer

CustID	PhoneType	Phone
15023	Land	222-3034
15023	Fax	222-4094
15023	Cell	223-0984
63478	Land	030-9693
63478	Fax	403-4094
94552	Land	849-4948
94552	Fax	292-3332
94552	Cell	139-3831
94552	Laptop	339-4040
45791	Land	294-4421
49004	Land	764-5103

Figure 3.7

Repeating values for phone numbers. Split the phone numbers into a separate table. Include the key (CustomerID) to link back to the original Customer table.

Today, it is possible that phone number is a repeating (or multi-valued) entity. Figure 3.6 shows another possible way of looking at the phone number data. At least with this approach, you would not have to guess at the number of phone number columns. However, it would be difficult to find or edit individual phone numbers, so you would never actually store the data this way. You could make do with the multiple columns shown in Figure 3.5, but it will waste space when you get beyond two or three types of numbers.

So what is a better answer? The solution to each of the normalization steps is always the same: Split the table into two tables. In this case, the phone number is causing the problem, so you need to put the phone numbers into a separate table. The one catch is that you must bring along the key column (CustomerID) so you will be able to join the phone numbers back to the customers.

Figure 3.7 shows the resulting two tables. Look at the sample data to see how the problems have been solved. Each phone number is listed in a separate row in the CustomerPhones table. A customer with a single phone number takes only one row of space. Yet, the table can hold data for as many phone numbers per person as you will need—even if new phone types are added later.

The example of phone numbers as repeating data is tricky. Most designers would probably choose to go with the method in Figure 3.5 by using multiple columns. However, as the number of phone types proliferates, you might need to switch to the version in Figure 3.7 to reduce the wasted space. In every case, the overtly repeating version in Figure 3.6 would be wrong. Fortunately, most situations of repeating data are considerably more obvious. Usually, you can identify repeating sections of data on a form because multiple lines are provided for entering data.

Dependency

Although researchers have defined dependency in mathematical terms, the concept is truly an issue of business rules. You can read the formal definitions in

CustomerID	Company Name
City	
Contact Last Name, First Name	
Phone	

Figure 3.8

A portion of a form for a firm that sells products to other companies. The dependencies among the attributes have to be identified based on common business practices.

the appendix, but the discussion in the chapter focuses on the business rules and uses language to explain dependency. If one attribute always identifies a specific value for another attribute, the second attribute is said to **depend** on the first. For instance, if someone gives you the CustomerID of 15023, you know that the LastName will always be Jones. It can never be anything else. Hence, LastName depends on CustomerID. Which would mean that CustomerID is a good candidate for becoming the primary key. On the other hand, if you were given the LastName of Jones, you probably would not be able to identify just one phone number. Jones is a fairly common name and the company probably has many customers with that name, so many phone numbers would show up for that name. Consequently, it is wrong to say that Phone depends on LastName. LastName would be a bad choice for a key column.

How do you know when one column depends on another? This question is the most difficult problem you face when designing a database. In fact, if someone were to tell you every single dependency in a case, it is easy to create the appropriate database design. In fact, you could create a program to build the design automatically (it has been done already). In other words, when you build a database design, you are really just identifying the business rules that specify how attributes (columns) depend on each other. Did you notice that the original question has not been answered? The answer is that you must talk with the users, study the forms and reports, and identify the dependency rules specifically for each situation.

To create an example, you need to know the basic business rules. Writing the business rules out would give away the answers, so usually you will be asked to study forms and reports to identify dependencies. However, you still need to use your business judgment. (It truly is important to take all of those other business courses—they will help you identify common business rules.) Consider a firm that sells products to other companies. Figure 3.8 shows a portion of an order form. You need to use your knowledge of common business rules to determine the dependencies among the attributes.

As a first attempt, you might try putting all of the attributes together into a single table. The CustomerID looks like it would make a good primary key. But, now you have to look at each potential column and ask yourself a basic question: (1) For a given value of the CustomerID, can there be more than one value of this attribute? If the answer is "yes," you must move the questioned attribute into a new table.

In the customer example, it is clear that the CompanyName depends on the CustomerID. The City attribute might be trickier. A customer could have offices in several cities. And, it might be critically important to your application to store the data this way. However, in many situations you can simply assume that City refers

Customer(<u>CustomerID</u>, CompanyName, City)
Contact(<u>ContactID</u>, CustomerID, LastName, FirstName)

Figure 3.9

The two main tables for the customer case. The key columns in the Contact table reveal that there can be many ContactIDs at each customer, but each contact works for only one customer. Where does the phone number belong?

to the corporate headquarters in a single city. You should record this assumption in your design notes so others can understand your design decisions. The Contact last and first names are more important. Although you might currently have only one contact at a customer firm, it is more reasonable to assume that in the future you will have multiple contacts at a company. From the repeating rule, you need to create a new table for the contacts. Figure 3.9 shows the initial tables. Notice that CustomerID is not part of the key in the Contact table because each contact person works for only one customer.

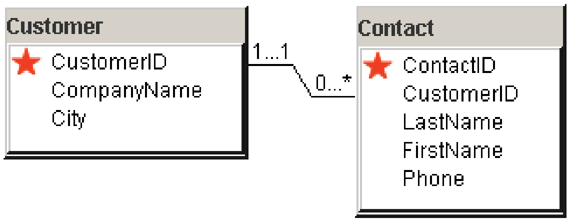
Now, where do you put the phone number column? This question illustrates the concept of dependency. Does the phone depend on (refer to) the customer or a contact? Technically, it could refer to either one. Most companies have a primary switchboard number that you could call. However, it is a generic number and requires you to go through several steps to find the person you want. It is more likely that the users of the system want the phone number of the specific contact person so that person can be reached directly. Consequently, Phone depends on ContactID and belongs in the Contact table instead of the Customer table. Figure 3.10 shows the resulting database design section.

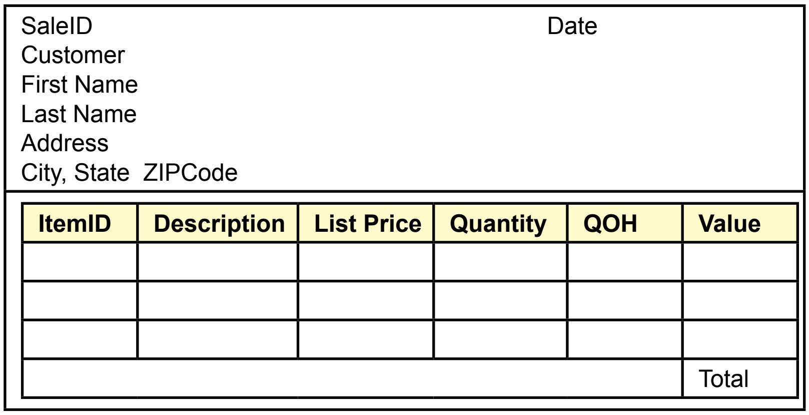
Sample Database for Typical Sales

How do you begin analyzing a form to create normalized tables? The best way to illustrate data normalization is to examine a sample problem. Remember that the results you get (the tables you create) depend heavily on the specific example and the assumptions you make. The following example uses a basic Sales Order form. The sample data are from fictional sales at a SCUBA dive shop, but the actual items could be anything and the principles will remain the same.

Figure 3.10

The final design with the Customer and Contact tables. The most reasonable assumption is to decide that users will want to call contacts directly, so Phone depends on ContactID instead of CustomerID.





SaleID Date
Customer
First Name
Last Name
Address
City, State ZIPCode

ItemID	Description	List Price	Quantity	QOH	Value
					Total

Figure 3.11

Sample sales form. First look for possible keys, keeping in mind that repeating sections (one-to-many relationships) will eventually need composite keys.

Figure 3.11 shows a basic sales order form. The main components of the sample form are the customer and the items being sold. When the form is built in the database, it will automatically keep track of the total amount due. It should also automatically assign a SaleID that is unique. The database form will also have buttons and drop-down lists to help the user enter data with a minimum of effort. For now, as you talk with the manager, you should sketch the desired features of the form. Values that can be computed (e.g., subtotals) should be marked, and the appropriate equations provided if needed. For the most part you do not want to store computed values in the data tables.

Initial Objects

One way to begin the design process is to identify the primary objects on a form. This step helps you think about the overall design and provides a start at identifying the tables. A form generally has several obvious entities. In this case, the obvious ones are customers and items. In real life you would also have employees. Managers also need to keep track of who purchased specific items. For example, if a manufacturer finds a problem with a piece of diving gear, the manager wants to send a notice to any customer who purchased that item. Hence you need two additional objects. The first is a transaction that records the date and the customer. It represents the overall form itself. The second is a list of the items purchased by that customer at that time. It represents the repeating section of the form.

Examine the initial objects in Figure 3.12. You need a primary key for customers (and items). Clearly, Name will not work, but you might consider using the Phone number. This approach would probably work, but it might cause some minor difficulties down the road. For example, if a customer gets a new phone number, you would have to change the corresponding phone number in every table that referred to it. As a primary key, it could appear in several different tables. A bigger problem would arise if a customer (Adams) moves, freeing up the phone number, which the phone company reassigns to another person (Brown) several months later. If Brown opens an account at your store, your database might mis-

Initial Object	Key	Sample Properties
Customer	Assign CustomerID	Name Address Phone
Item	Assign ItemID	Description List Price Quantity On Hand
Sale	Assign SaleID	Sale Date
SaleItems	SaleID + ItemID	Quantity

Figure 3.12

Initial objects for the sale form. Note that the transaction has two parts, Sale and SaleItems because many items can be sold at many different times.

takenly identify customer Brown as the customer Adams. The safest approach is to have the database create a new number for every customer.

The Item object also needs a key. In practice you might be able to use the product identifiers created by the manufacturer. For now, it is easiest to assign a separate number. Basic properties include the item description, list price and quantity on hand (QOH). More attributes (such as size) can be added later if necessary.

Every transaction must be recorded. The transaction in this case is the entire sale. This object refers to the overall sale form and is also assigned a unique key value. Remember this approach. Almost all of the problems you encounter will end up with a table to hold data for the base form or report.

An important issue in many situations is the presence of a repeating section, which can cause problems for storing data. Hence, the section is split from the main transaction and stored in its own table. Keys here include the SaleID from the Sale table and the ItemID. Note that the key is composite because a many-to-many relationship exists. A customer can buy many products at one time and a product can be purchased (at different times) by more than one customer.

Initial Form Evaluation

Practice is required to identify all of the tables needed for a form or report. In the Sales Order example, most people should be able to identify the Customer and Item tables. Some will recognize the need for a Sale table. However, the purpose of the SaleItems table is not as clear. Fortunately, there is a method to derive the individual tables by starting with the entire form and breaking it into pieces. This method is the data normalization approach, and it is a mechanical process that follows from the business assumptions.

Figure 3.13 shows the first step in the evaluation. As you learn normalization, you should be careful to write out this first step. As you gain experience, you might choose to skip this step. The procedure is to look through the form or report and write down everything that you want to store. The objective is to write it in a structured format. Give the form a name and list the items as column names. You can generally start at the top left of the form and write a column name for each data element. Try to list items together that fall into natural groupings—such as all customer data. The SaleForm begins with the SaleID, which looks like it would make a good key. The SaleDate and CustomerID are listed next, followed by the

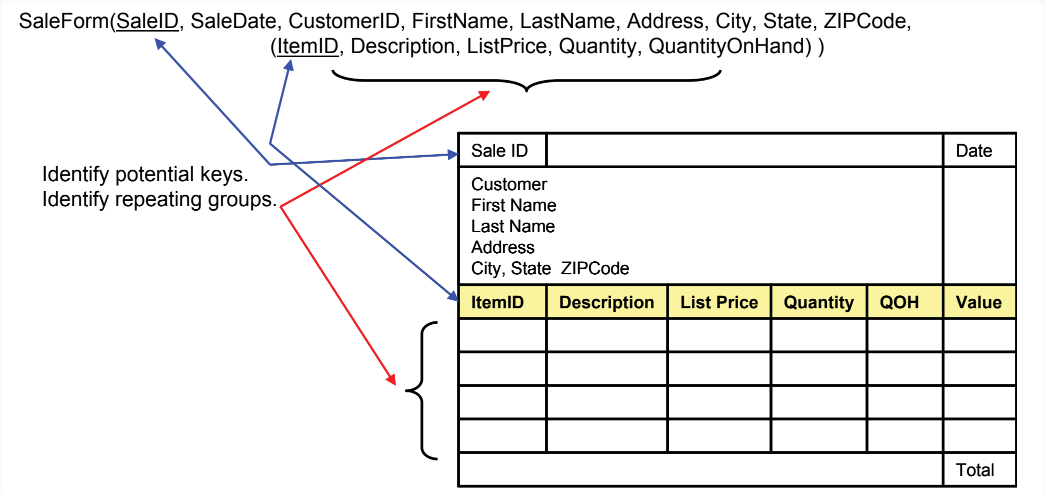

SaleForm(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode,
(ItemID, Description, ListPrice, Quantity, QuantityOnHand))
Identify potential keys.
Identify repeating groups.
Sale ID
Date
Customer
First Name
Last Name
Address
City, State ZIPCode
ItemID
Description
List Price
Quantity
QOH
Value
Total

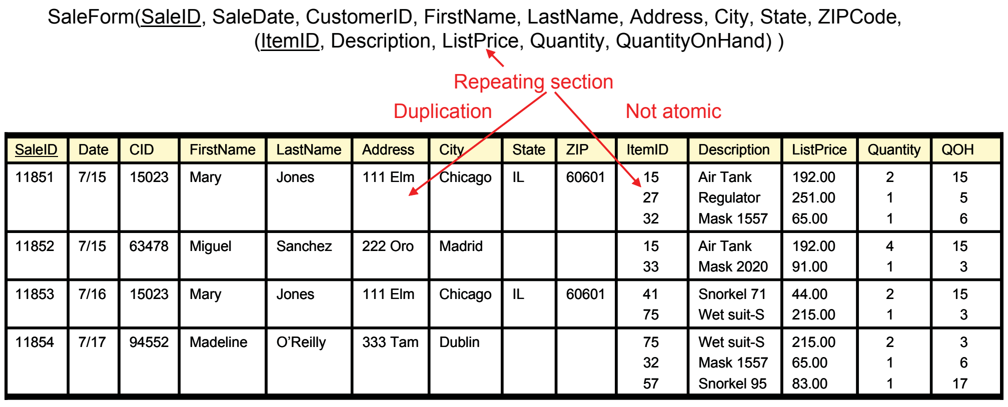
Figure 3.13

Initial form evaluation. Once you have collected basic user forms, you can convert them into a more compact notation. The notation makes the normalization steps easier by highlighting potential issues.

basic customer data. The next step is slightly more complicated because you have to signify that the section with the items contains repeating data. That is, it has multiple lines of data or the potential for several similar entries. Repeating data represents a one-to-many relationship that must be handled carefully. An easy way to signify the repeating section is to put it inside another set of parentheses. Some people also list it on a new line.

Observe that the computed total was not included, since it can be recalculated as needed. However, in some cases you might want to store computed data. For instance, if you compute a sales tax with each order, it is convenient to store the computed value. Even though the tax could be recomputed later, changing tax rates and round-off differences might lead to errors in the later calculations. The reduction in risk is worth the small extra storage of data. However, you should mark the items or add a description so you remember they are computed values.

While you are working on the first step, be sure to write down every item that you want to store in the database. In addition, make sure to identify every repeating section. Here you have to be careful. Sometimes repeating sections are obvious: They might be in a separate section, highlighted by a different color, or contain sample data so you can see the repetition. Other times, repeating sections are less obvious. For example, on large forms repeating sections might appear on separate pages. Other times, some entries might not seem to be repeating. You should also try to mark potential keys at this point, both to indicate repeating sections and to highlight columns that you know will contain unique data. On some DBMSs, you can create a **pseudo column** to define a computed value. This value is not actually stored, but recomputed as needed. For instance, the Value column could be computed as price times quantity. Finally, when you write down a column name, you should add it to a data dictionary and record attributes such as the data type and which person is responsible for the item.


Repeating section
Duplication
Not atomic

SaleForm(<u>SaleID</u>, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode, (<u>ItemID</u>, Description, ListPrice, Quantity, QuantityOnHand))

<u>SaleID</u>	Date	CID	FirstName	LastName	Address	City	State	ZIP	ItemID	Description	ListPrice	Quantity	QOH
11851	7/15	15023	Mary	Jones	111 Elm	Chicago	IL	60601	15 27 32	Air Tank Regulator Mask 1557	192.00 251.00 65.00	2 1 1	15 5 6
11852	7/15	63478	Miguel	Sanchez	222 Oro	Madrid			15 33	Air Tank Mask 2020	192.00 91.00	4 1	15 3
11853	7/16	15023	Mary	Jones	111 Elm	Chicago	IL	60601	41 75	Snorkel 71 Wet suit-S	44.00 215.00	2 1	15 3
11854	7/17	94552	Madeline	O'Reilly	333 Tam	Dublin			75 32 57	Wet suit-S Mask 1557 Snorkel 95	215.00 65.00 83.00	2 1 1	3 6 17

Figure 3.14

Problems with repeating data. Storing repeating data with the main form results in either non-atomic cells or substantial duplication of data.

Problems with Repeating Sections

The reason you have to be so careful in identifying repeated sections or one-to-many relationships is that they can cause problems in the database. The situation in Figure 3.14 shows what happens when you try to store the data from the form exactly the way it is written now. In particular, the repeating section causes problems. As it is displayed, it results in non-atomic data in the cells because of the need to store multiple items for each order. You might try to avoid the problem by storing each item in a separate row, but then you would have to duplicate the sale and customer data for each item being sold. The other problems with this attempt are explained in the next sections, but you might as well solve the problem now.

Several other problems arise because of this weak design. What do you know about products that have not been sold yet? Conversely, what if you delete old data, such as all of last year's sales? As you delete sales, you also delete item and customer data. Suddenly, you notice that you deleted half of the customer base. Technically, these problems are known as an **insertion anomaly** and a **deletion anomaly**; that is, when the data is not stored in a proper format, you encounter difficulties as you try to add or delete data. These problems arise because you tried to store all the data in one table.

First Normal Form

How do you create a design in first normal form? The answer to the problem with repeating sections is to put them into a separate table. When all cells contain atomic data, (for example, a table has no repeating groups), it is said to be in **first normal form (1NF)**. That is, for each cell in a table (one row and one column), there can be only one value. This value is atomic in the sense that it cannot be decomposed into smaller pieces.

Repeating Groups

As shown in some of the prior examples, some **repeating groups** are obvious. Others are more subtle and deciding whether to split them into a separate table is more difficult. The first normalization rule is clear: If a group of items repeats,

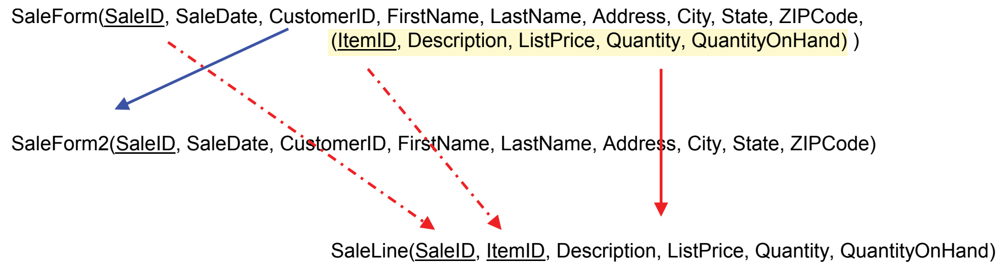

SaleForm(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode,
(ItemID, Description, ListPrice, Quantity, QuantityOnHand))
SaleForm2(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)
SaleLine(SaleID, ItemID, Description, ListPrice, Quantity, QuantityOnHand)

Figure 3.15

First normal form. All repeating (non-atomic) groups must be split into new tables. Be sure that the new table includes a copy of the key from the original table. The table holding the repeating group must have a composite key so that the data can be recombined in queries.

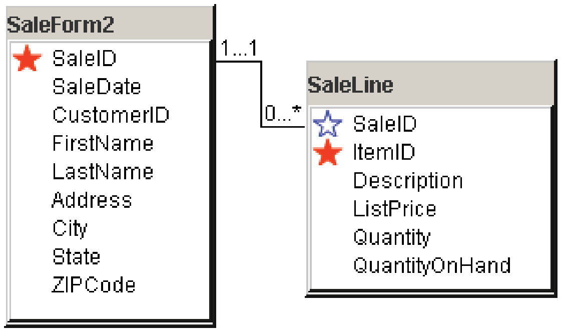
it should be split into a new table. The solution in all cases is the same: Split the design into two tables. If repeating groups remain, split the design again.

Return to the scuba store example, as shown in Figure 3.15, and notice the repeating section that is highlighted by the parentheses. To split this form, first separate everything that is not in the repeating group. These columns might need other changes later, but the section contains no repeating groups. Second, put all the columns from the repeating item sales section into a new table. However, be careful. When you pull out a repeating section, you must bring down the key from the original table. The SaleForm table has SaleID as a primary key. This key, along with the ItemID key, must become part of the new table SaleLine. You need the Sale key so that the data from the two tables can be recombined later. Note that the new table (SaleLine) will always have a composite key—signifying the many-to-many relationship between sales and items.

Figure 3.16 shows the current design in the database design system. Keep in mind that this design is merely the first step. Just glancing at the figure should tell you there is a serious problem—because the SaleLine table contains both a generated key and a non-generated key column. Remember, because there is a many-to-many relationship between Sale and Item, you eventually need a table

Figure 3.16

Current design. Splitting the repeating items into the SaleLine table helps but it does not solve all of the problems.



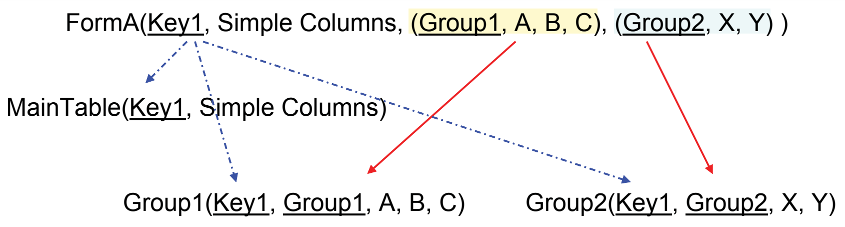

FormA(Key1, Simple Columns, (Group1, A, B, C), (Group2, X, Y))
MainTable(Key1, Simple Columns)
Group1(Key1, Group1, A, B, C)
Group2(Key1, Group2, X, Y)

Figure 3.17

Independent groups. In this example, two groups are repeating independently of each other. They are split separately into new tables. Remember to include the original key (Key1) in every new table.

that contains both SaleID and ItemID as keys. The point to remember is that this design is in 1NF, which means it is better than putting everything into a single table—but not much better.

Splitting off the repeating groups solves several basic problems. First, it reduces the duplication: You no longer have to enter customer data for every item that is sold. In addition, you do not have to worry about allocating storage space: Each item sold will be allocated to a new row.

Multiple Repeating Groups

Before looking at the next step in normalization, you should realize that repeating sections can be considerably more complicated. Remember that you pick up the initial design from forms and reports, and you will be amazed at the complexity that can arise on business forms. Two common situations are: (1) independently repeating groups, and (2) nested repeating groups.

Many forms will have several different groups that repeat. As shown in Figure 3.17, if they repeat independently of each other, the split is straightforward; each group becomes a new table. Just be careful to include the original key in every new table so the tables can be linked together later. Using the base notation, groups are independent if the parentheses do not overlap. For example, a more complex sales case could have a second repeating group of items that are being leased. The leased items would often be stored separately because additional lease data is needed for each item.

Nested Repeating Groups

More complicated situations arise when several different repeating groups occur within a table—particularly when one repeating group is nested inside another group. The greatest difficulty lies in identifying the nested nature of the groups. As illustrated in Figure 3.18, after you identify the relationships, splitting the tables is straightforward. Just go one step at a time, pulling the outermost groups first. Always remember to bring along the prior key each time you split the tables. So when you pull the second group (Key2 … (Key3 …)) from the first group (Key1 …), the new TableA must include Key1 and Key2. When you pull Table3 from TableA, you must bring along all the prior keys (Key1 and Key2) and then add the third key (Key3).

A more sophisticated sale problem could encounter nested repeating groups. For example, the store might sell (and ship) items to several departments for the

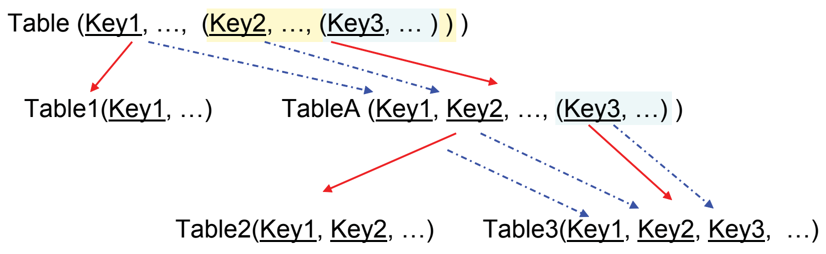


Table (Key1, …, (Key2, …, (Key3, …)))
Table1(Key1, …)
TableA (Key1, Key2, …, (Key3, …))
Table2(Key1, Key2, …)
Table3(Key1, Key2, Key3, …)

Figure 3.18

Nested repeating groups. Groups are nested when they repeat within another group (key3 inside key2 inside key1). Split them in steps: Pull all of group2 from group1, then pull group3 from group2. Note that every table will contain the original key (key1). With three levels, the final table (Table3) must contain three columns in the key.

customer. The Sale would be the top level, departments would be next, and individual sale items would be nested within the departments. The table resulting from the innermost nesting would have a primary key consisting of the SaleID, Department, and ItemID columns.

Second Normal Form

What is second normal form? It was straightforward to reach first normal form: Just identify the repeating groups and put them into their own table that is linked to the main table through the initial key. The next step is a little more complicated because you have to look at relationships between the key value and the other (nonkey) columns in the table. Correct specification of the keys is crucial. At this point it would be wise to double-check all the keys to make sure they are unique and that they correctly identify many-to-many relationships. In particular, focus on tables where the primary key consists of more than one column. Second normal form is concerned with the situation where a nonkey column depends on only part of the key.

Problems with First Normal Form

It is fairly clear that 1NF is not the final answer. You still face a couple of major design issues. The DB Design system highlights some of the issues with the keys in the SaleLine group. A generated key must always be the only key column in a table, but the SaleLine table needs both SaleID and ItemID as keys. One solution might be to just remove the generator from ItemID so it is a simple key column. But, if you make that change, where would you get values for the ItemID? The problem is that the SaleLine table is trying to do two things: show which items were sold and describe individual items.

You can guess by the temporary names of the tables in Figure 3.15 that first normal form might still have problems storing data efficiently. Consider the situation in Figure 3.19 that illustrates the current Sale Item table. Every time someone buys item 15, the database stores the *Air Tank* description. The problem is that the description depends on only part of the key (ItemID). If you know the ItemID, you always know the corresponding description. The description does not change with every transaction. Beyond the waste of space and clerical time, there is an

SaleLine(<u>SaleID</u>, <u>ItemID</u>, Description, ListPrice, Quantity, QuantityOnHand)

Duplication for columns that depend only on ItemID

<u>SaleID</u>	<u>ItemID</u>	Description	ListPrice	Quantity	QOH
11851	15	Air Tank	192.00	2	15
11851	27	Regulator	251.00	1	5
11851	32	Mask 1557	65.00	1	6
11852	15	Air Tank	192.00	4	15
11852	33	Mask 2020	91.00	1	3
11853	41	Snorkel 71	44.00	2	15
11853	75	West suit-S	215.00	1	3
11854	75	Wet suit-S	215.00	2	3
11854	32	Mask 1557	65.00	1	6
11854	57	Snorkel 95	83.00	1	17

Figure 3.19

Problems with first normal form. This design is in 1NF but it still contains duplicated data. Every time an item is sold, the clerk has to reenter its description, list price, and quantity on hand. Also, if an item has not yet been sold, what is its ListPrice? The problems arise because these columns depend only on the ItemID, not on the SaleID.

additional problem: If an item has not yet been sold, what is its description (and price)? Because items are only entered into the database with a transaction, this data will not be stored in the database. Similarly, if all the rows for item 15 are deleted, you will lose all the associated information about that product.

Second Normal Form Definition

The problem with the preceding example is that once you know the ItemID, you always know the description. A one-to-one relationship exists between the ItemID and the Description (perhaps many-to-one). As shown in Figure 3.20, the important point is that the sale transaction does not matter. If someone buys item 15 in June, the description is *Air Tank*. If someone buys item 15 in December, the description is still *Air Tank*. Hence, the description depends on only part of the key (the ItemID and not the SaleID). A table is in **second normal form (2NF)** if every nonkey column depends on the entire key (not just part of it). Note that this issue arises only for composite keys (with multiple columns).

The solution is to split the table. Pull out the columns that depend on part of the key. Remember to include that part of the key in the new table. The new tables (SaleItems and Item) are shown in Figure 3.21. Note that ItemID must be in both tables. It stays in the SaleItems table to indicate which items have been purchased at each time. It is the primary key in the Item table because it is the unique identifier. Including the column in both tables enables you to link the data together later.

In creating the new Item table, you are faced with the interesting question of where to put the price. There are two choices: in the SaleItems table or in the Item table. The answer depends on the operations and rules used in the business. From a technical standpoint you can choose either table. However, from a business standpoint there is a big difference. Consider the case where the price is in

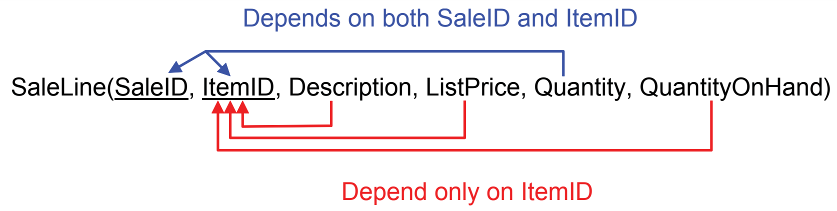

Depends on both SaleID and ItemID
SaleLine(SaleID, ItemID, Description, ListPrice, Quantity, QuantityOnHand)
Depend only on ItemID

Figure 3.20

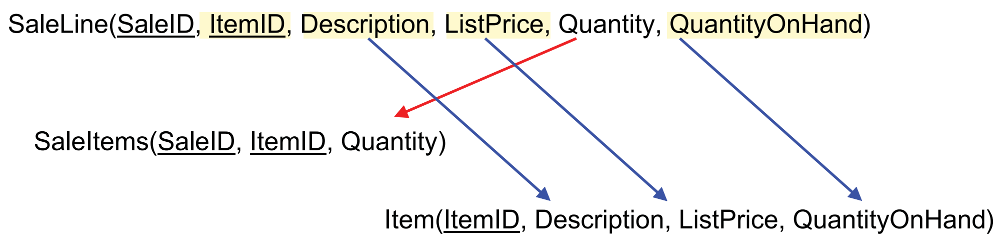
Second normal form definition. Each nonkey column must depend on the entire key. It is only an issue with composite keys. The solution is to split off the parts that only depend on part of the key.

the Item table. This model of the firm says that if you know the ItemID, you always know the price. In other words, the price is fixed for each item and does not change over time. Now consider the interpretation when the price is stored in the Item table. Here you are explicitly saying that the price depends on both the ItemID and on the specific sale. In other words, for one customer the price for *Air Tank* might be $192, whereas another customer might pay only $175. The price difference might arise because you give end-of-season discounts, or if someone purchases several items at one time. Most business database designers quickly encounter the problem of where to store prices. One solution is to store prices in both tables. That is, the price in the Items table (ListPrice) would be the list price that the business intends to charge. The price in the SaleItems table would be the actual price paid that incorporates various discounts (SalePrice). The key point is that the final list of tables depends not just on mechanical rules but is also determined by the operations of the business. The assumptions you make about how a particular business operates determine the tables you get. For now, you will stick with the simpler assumption that assigns a fixed ListPrice to each item.

Figure 3.22 gives sample data for the new tables. Notice that 2NF resolves the problem of repeating the description each time an item is sold. The base product data is stored one time in the Item table. It is referenced in the SaleItem table by the ItemID. Looking through the SaleItem table, you can easily get the corresponding description by finding the matching ID in the Item table. Chapter 4 explains how the database query system handles this link automatically.

Figure 3.21

Creating second normal form. Split the original table so that the items that depend on only part of the key are moved to a separate table. Note that both tables must contain the ItemID key.


SaleLine(SaleID, ItemID, Description, ListPrice, Quantity, QuantityOnHand)
SaleItems(SaleID, ItemID, Quantity)
Item(ItemID, Description, ListPrice, QuantityOnHand)

SaleID	ItemID	Quantity
11851	15	2
11851	27	1
11851	32	1
11852	15	4
11852	33	1
11853	41	2
11853	75	1
11854	75	2
11854	32	1
11854	57	1

SaleItems(SaleID, ItemID, Quantity)

ItemID	Description	ListPrice	QOH
15	Air Tank	192.00	15
27	Regulator	251.00	5
32	Mask 1557	65.00	6
33	Mask 2020	91.00	3
41	Snorkel 71	44.00	15
57	Snorkel 95	83.00	17
75	Wet suit-S	215.00	3
77	Wet suit-M	215.00	7

Item(ItemID, Description, ListPrice, QuantityOnHand)

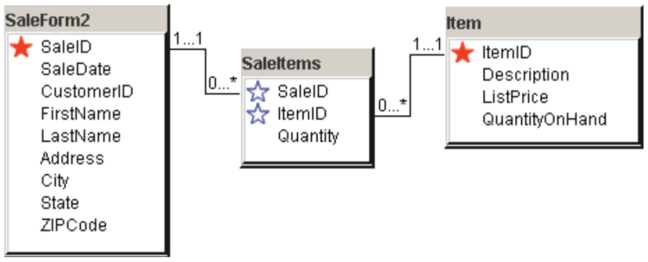
Figure 3.22

Second normal form data. Product items are now stored only one time. Other tables (SaleItems) can refer to an item just by its key (ItemID), which provides a link back to the Item table.

Figure 3.23 shows the current status of the tables in DB Design. The problem with the ItemID key has been solved. The ItemID key values are generated in the Item table whenever a new product is added to inventory. Every column in the Item table depends only on the ItemID key. The ItemID values are used along with the SaleID values in the SaleItems table to show exactly which items are purchased on each sale. Almost any time you have a many-to-many relationship, you will see a similar pattern. One table will be used to generate each key column, and the intermediate or junction table will use the two keys to handle the many-to-many relationship. The challenge is to identify exactly which columns depend on both key columns and belong in the intermediate table.

Figure 3.23

Second normal form in DB Design. ItemID is generated in a table that only refers to items. This value is used along with SaleID in a table that shows which items were purchased on each sale.



SaleForm2(<u>SaleID</u>, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)

<u>SaleID</u>	Date	CustomerID	FirstName	LastName	Address	City	State	ZIP
11851	7/15	15023	Mary	Jones	111 Elm	Chicago	IL	60601
11852	7/15	63478	Miguel	Sanchez	222 Oro	Madrid		
11853	7/16	15023	Mary	Jones	111 Elm	Chicago	IL	60601
11854	7/17	94552	Madeline	O'Reilly	333 Tam	Dublin		

Duplication

Figure 3.24

Problems with second normal form. The hidden dependency in the customer data leads to duplicating the customer address each time a customer rents videos from the store. Similarly, if old transaction rows are deleted, the firm might lose all of the data for some customers.

Third Normal Form

What is third normal form? The logic, analysis, and elements of designing for **third normal form (3NF)** are similar to those used in deriving 2NF. In particular, you still concentrate on the issue of dependence. With experience, most designers combine the derivation of 2NF and 3NF into a single step. Technically, a table in 3NF must also be in 2NF.

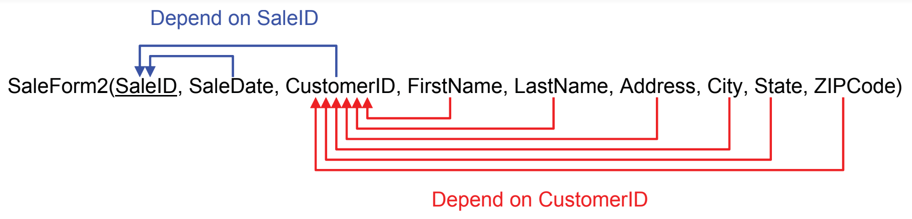
Problems with Second Normal Form

At this point, you need to examine the SaleForm2 table that was ignored in the earlier analysis. It is displayed in Figure 3.24. In particular, notice that SaleID is the key. The problem can be seen in the sample data. Every time a customer participates in a sale, the database stores his or her name, address, and phone number again. This unnecessary duplication is a waste of space and probably a waste of the clerk's data entry time. Consider what happens when a customer moves. You would have to find the address and change it for every transaction the customer had with the store. Likewise, if the customer has not yet purchased any items, you do not have a place to store the customer data. Similarly, if you delete old transactions from the database, you risk losing customer data. The problem arises because you have a hidden dependency. The solution is to make the dependency explicit.

Third Normal Form Definition

The problems in the previous section are fairly clear. The customer name, address, phone, and so on depend on the CustomerID. Given a specific value for CustomerID, you immediately know the rest of the customer data. The problem with the design at this point is that CustomerID is not part of the key for the table. In other words, some nonkey columns do not depend on the key. So why are they in this table? The question also provides the solution. If columns do not depend on the primary key, they should be placed in a separate table.

To be in 3NF a table must already be in 2NF, and every nonkey column must depend on nothing but the key. In the video example in Figure 3.25, the problem



Depend on SaleID
SaleForm2(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)
Depend on CustomerID

Figure 3.25

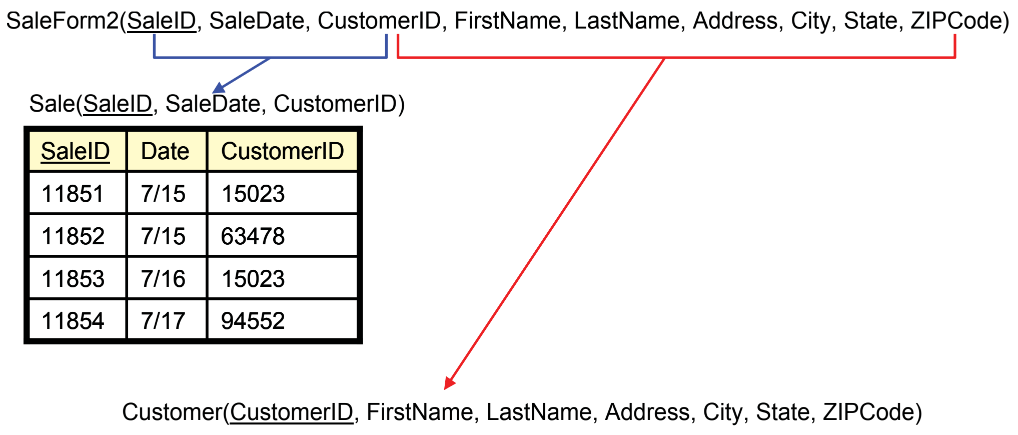
Third normal form definition. This table is not in 3NF since some of the columns depend on CustomerID, which is not part of the key.

is that basic customer data columns depend on the CustomerID, which is not part of the key.

At first glance, two solutions seem possible: (1) make CustomerID part of the key or (2) split the table. If the table is already in 2NF, option (2) is the only choice that will work. The problem with the first option is that making CustomerID part of the key is equivalent to stating that each transaction can involve many customers. This assumption is not likely to be true. However, even if it is, your table would no longer be in 2NF, since the customer data would then depend on

Figure 3.26

Third normal form. Putting customer data into a separate table eliminates the hidden dependency and resolves the problems with duplicate data. Note that CustomerID remains in both tables, but it is still not a key in the Sale table because only one customer participates in a given sale.



SaleForm2(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)
Sale(SaleID, SaleDate, CustomerID)
Customer(CustomerID, FirstName, LastName, Address, City, State, ZIPCode)

SaleID	Date	CustomerID
11851	7/15	15023
11852	7/15	63478
11853	7/16	15023
11854	7/17	94552

CustomerID	FirstName	LastName	Address	City	State	ZIP
15023	Mary	Jones	111 Elm	Chicago	IL	60601
63478	Miguel	Sanchez	222 Oro	Madrid		
94552	Madeline	O'Reilly	333 Tam	Dublin		

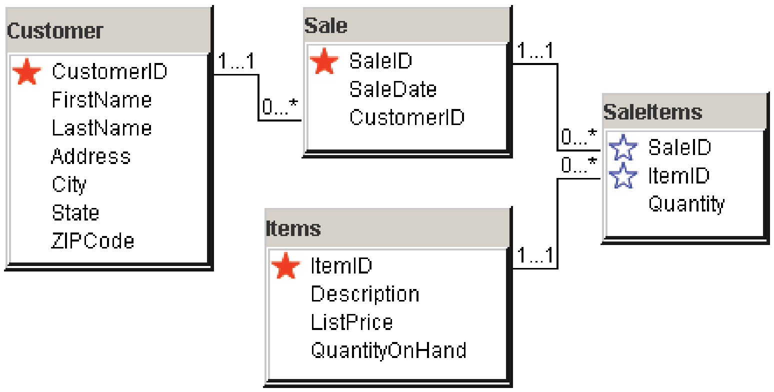


Figure 3.27

Third normal form tables. Each cell is atomic, with no no repeating groups within a table, and each nonkey column depends on the whole key and nothing but the key.

only part of the key (CustomerID and not TransID). Hence the correct solution is to split the table into two parts: the columns that depend on the whole key and the columns that depend on something else (CustomerID).

The solution in the sale example is to pull out the columns that are determined by the CustomerID. Remember to include the CustomerID column in both tables so they can be relinked later. The resulting tables are displayed in Figure 3.26. Notice that CustomerID is not a key in the Sale table because only one customer participates in any given sale. Figure 3.26 also illustrates how splitting the tables resolves the problems from the hidden dependency.

The final collection of tables is presented in Figure 3.27. This list is in 3NF: Each cell is atomic and there are no repeating groups within a table (1NF), and each nonkey column depends on the whole key (2NF) and nothing but the key (3NF). You can change the layout of the tables, but the relationships remain the same. As shown in Figure 3.28, you can also write the list of tables and their column names. This approach makes it easy to fit dozens of tables on one page; however, the relationships are more difficult to see.

The astute reader should raise a question about the address data. That is, City, State, and ZipCode have some type of dependent relationship. Perhaps the Customer table is not really in 3NF? In theory, it is true: ZIP codes were created as a means to identify locations. The catch is that at a five-digit level, the relationship is relatively weak. A ZIP code identifies an individual post office. Each city can have many ZIP codes, and a ZIP code can be used for more than one city. At the moment, it is true that a ZIP code always identifies one state. However, can you be certain that this relationship will always hold—even in an international setting? Hence it is generally acceptable to include all three items in the same table. On the other hand, as pointed out in the pet store discussion in Chapter 2, there are some advantages to creating a separate City table. The most important advantage is that you can reduce data entry time and errors by selecting a city from a predefined list.

Customer(CustomerID, FirstName, LastName, Address, City, State, ZIPCode)
Sale(SaleID, SaleDate, CustomerID)
SaleItems(SaleID, ItemID, Quantity)
Item(ItemID, Description, ListPrice, QuantityOnHand)

Figure 3.28

Third normal form table list. The list is an easy way to fit dozens of tables on a page but does not show the relationships.

Checking Your Work

At this critical point, you must double-check your work. In large projects it is beneficial to have several team members participate in the review to make sure the assumptions used in defining the data tables match the business operations.

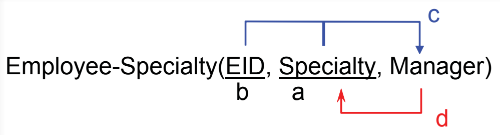
The essence of data normalization is to collect all the forms and reports and then to inspect each form to identify the data that will be stored. Writing the columns in a standard notation makes the normalization process more mechanical, minimizing the potential for mistakes. In particular, look for keys and highlight one-to-one and one-to-many relationships. To check your work, you need to examine each table to make sure it demonstrates the assumptions and operations of the firm.

To check your tables, you essentially repeat the steps in normalization. First, make sure that you have pulled out every repeating group. While you are at it, double-check your keys. Be sure you know exactly where each key value is generated. To verify the key columns, start with the first key column in a table and ask yourself if there is a one-to-one or a one-to-many relationship with each of the other columns. If it is a one-to-many relationship (or many-to-many), you need to underline the column title. If it is one-to-one (or many-to-one), the column in question should not be underlined. The second step is to look at each nonkey column and ask yourself if it depends on the whole key and nothing but the key. Third, verify that the tables can be reconnected. Try drawing lines between each table. Tables that do not connect with the others are probably wrong. Fourth, ask yourself if each table represents a single object. Try giving it a name. If you cannot find a good single name for the table, it probably represents more than one object and needs to be split. Finally, enter sample data for each table and make sure that you are not entering duplicate rows. Some underlying problems may become obvious when you begin to enter data. It is best to enter test data during the design stage, instead of waiting until the final implementation.

Beyond Third Normal Form

What problems exist beyond third normal form? In designing relational database theory, E. F. Codd first proposed the three normalization rules. On examining real-world situations, he and other writers realized that additional problems could occur in some situations. In particular, Codd's initial formal definition of 3NF was probably too narrow. Hence he and Boyce defined a new version, which is called **Boyce-Codd normal form (BCNF)**.

Other writers eventually identified additional problems that could arise and created further "normal forms." If you are careful in designing your database—particularly in creating keys—you should not have too many problems with these



a. Each employee has many specialties
b. Each specialty has many managers
c. Employee + manager is one specialty
d. Each manager has one specialty

Figure 3.29

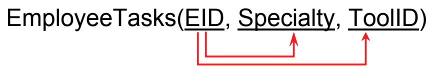
Boyce-Codd normal form. There is a hidden dependency (d) between manager and specialty. If we delete rows from the original table, we risk losing data about our managers. The solution is to add a table to make the dependency explicit.

issues. However, occasionally problems arise, so a good database designer will check for the problems described in the following sections. In particular, in large projects with many designers, one member of the team should check the final list of tables.

Boyce-Codd Normal Form

You have already seen how problems can arise when hidden dependencies occur within a table. A secondary relationship between columns within a table can cause problems with duplication and lost data. Consider the example in Figure 3.29, which contains data about employees. From the business rules, it is clear that the table is in 3NF. The keys are correct, and from rule (c) the nonkey column (Manager) depends on the entire key. That is, each employee can have a different manager for each specialty. The problem arises because of business rule (d): Each manager has only one specialty. The manager determines the specialty, but since Manager can never be a key for the entire table, you have a hidden dependency (Manager → Specialty) in the table. A **hidden dependency** arises when there is a functional rule that is not part of the primary key. What if you delete old data rows and delete all references to one manager? Then you lose the data that revealed that manager's specialty. BCNF prevents this problem by stating that any dependency must be explicitly shown in the keys.

The solution is to add a table to make the dependency explicit. Because each specialty can have many managers, the best solution is to add the table Manager(Manager, Specialty). Note that technically, you can now remove the Specialty column from the original table (and key Manager). Because a manager can have only one specialty, as soon as you know the manager, you can use a link to obtain the specialty. However, as a designer, you have to question to rules. This situation requires some unusual rules. If the manager-specialty rule is relaxed in the future, allowing managers to have multiple specialties, you would have to redesign the tables (and forms and reports). In the example it is not very realistic to believe the firm will always have managers with only one specialty. It is better to leave the original table and add the new Manager table. Then if the assumptions change, you simply need to make Specialty a key in the Manager table. The main point is that you have solved the BCNF problem by explicitly recording the hidden relationship—so you no longer need to worry about losing important relationships when you delete rows.



Business rules.

(a) Each employee has many specialties.
(b) Each employee has many tools.
(c) Tools and specialties are unrelated.

EmployeeSpecialty(EID, Specialty)
EmployeeTools(EID, ToolID)

Figure 3.30

Fourth normal form. The original table is 3NF because there are no nonkey columns. The keys are legitimate, but there is a hidden (multivalued) dependency because Specialty and ToolID are unrelated. The solution is to create two tables—one to show each of the two dependencies.

Fourth Normal Form

Fourth normal form (4NF) problems arise when there are two binary relationships, but the modeler attempts to show them as one combined relationship. An example should clarify the situation.

In Figure 3.30, employees can have many specialties, and they perform many tasks for each specialty. Because all three columns are keyed, the table must be in 3NF. From the business rules, you can see that the keys are legitimate. However, there are really two binary relationships instead of one ternary relationship: Employee → Specialty and Employee → Tool.

Since the third business rule specifies that Specialty and ToolID are not directly dependent on each other, you need to break up the original table into two tables to remove the hidden dependency. The problem you would face with the original is that there could be considerable duplication of data if for every employee you have to list each tool for every specialty. It is more efficient to list specialties and tools separately.

Fourth normal form problems can occur and they can cause problems, so you should be able to spot them. The main trick is to watch for hidden dependencies, and make sure they are made explicit.

Domain-Key Normal Form

In 1981 Fagin described a different approach to normalized tables when he proposed the **domain-key normal form (DKNF)**. DKNF describes the ultimate goal in designing a database. If a table is in DKNF, Fagin proved that it must also be in 4NF, 3NF, and all of the other normal forms. The catch is that there is no defined method to get a table into DKNF. In fact, it is possible that some tables can never be converted to DKNF.

Despite these difficulties, DKNF is important for application developers because it is a goal to work toward when designing applications. Think of it as driving to the mall when you do not have exact directions. You can still get there as long as you know how to start (1NF, 2NF, and 3NF are well-defined) and can recognize the mall when you arrive (DKNF).

The goal of DKNF is to have each table represent one topic and for all the business rules to be expressed in terms of domain constraints and key relationships.

EmployeeTask(EmployeeID, TaskID, ToolID)

Defined business rules
(a) Each employee performs many tasks with many tools.

But, maybe you need a second rule.
(b) Each task has commonly used tools.
RequiredTools(TaskID, ToolID)

Figure 3.31

DKNF example. With the stated rule, the tables are in BCNF but might not be in DKNF. The initial table combines information about tasks and tools. Maybe there is an additional undefined dependency between task and tool, where employees commonly use the same tools for each task.

That is, all business rules are explicitly described by the table rules. Domain constraints are straightforward—they represent limitations placed on the data held in a column. For example, prices cannot be negative.

All other business rules must be expressed in terms of relationships with keys. In particular, there can be no hidden relationships. Consider the example in Figure 3.31, which shows a table that records the tasks performed by employees and the tools they used. The primary business rule you were given states that each employee performs many tasks with many tools, so all three columns need to be part of the primary key. The key columns are legitimate and the table is in 3NF. Since only one rule (dependency) has been specified, the table is also in BCNF. However, if you think about the business problem for a few minutes, you can see that the table might be used to cover two topics: (1) The tools that employees actually used, and (2) The tools that are commonly used for a specific task. Think about the problem from the perspective of a novice employee who needs to know which tools to pick up for a specific task. You could query the database to see what tools other employees used in the past, but there could be considerable variation. Perhaps there needs to be a second dependency that lists the minimum set of tools required for each task. If you know about this rule at the start, you can see that the single EmployeeTask table violates BCNF because it ignores a hidden rule. But, since the rule was not explicitly stated, you used the DKNF approach to realize the initial table was trying to cover two different facts. With this insight, you can look harder to identify formal rules.

This example also shows you the challenge of DKNF. There is no formal method to arrive at DKNF. To define a set of tables in DKNF, you can start by working through the 3NF rules. Then look carefully for hidden dependencies and add tables to reach BCNF. Then, verify keys and ensure that each table describes a single fact, and that facts are stored in only one location. Domain-key normal form returns to the beginning of Chapter 2. The goal in designing the database is to build a model of the organization, and DKNF clarifies this goal by stating that the best database design is one that explicitly states all business rules as database rules.

In theory, there can be no normal forms beyond DKNF. That is a nice theory, but since there is no well-defined way to put a set of tables in DKNF, it is not always helpful. Several authors have identified other potential problems and derived additional versions of normal forms, such as fifth normal form. For the most part these definitions are not very useful in practice; they will not be described here. You can consult C. J. Date's textbooks for details and examples of more theoretical concepts.

Summary

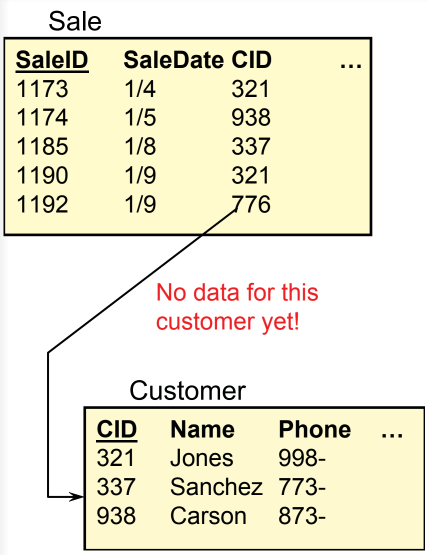
You can review the technical definitions in the appendix for a formal statement of the normalization conditions. However, the most important thing to remember is that normalization ultimately comes down to properly understanding the business rules (dependencies). The first rule is straightforward: Each cell contains atomic, non-repeating data. The second and third rule can be summarized by remembering that: Each nonkey column depends on the whole key and nothing but the key. (So help me Codd.) BCNF seems slightly trickier, but a simple rule can be used to represent the entire process: There must be no hidden dependencies. All dependencies should be explicitly stated within the primary keys.

Data Rules and Integrity

How does a database record constraints? As you talk to users and managers to design reports and tables, you also need to think about what business rules need to be enforced. One of the goals of a database designer is to ensure that the data remains accurate. Many cases have straightforward business rules. For example, you typically want to make sure that price is greater than zero. Similarly, you may have a constraint that salaries should not exceed some number like $100,000 or that the date hired has to be greater than the date the company was founded. These **data integrity** constraints are easy to assign in most databases. Typically, you can go to the table definitions and add the simple constraints along with a message. The advantage of storing these constraints with the tables is that the DBMS enforces the conditions for every operation on the table, regardless of the source or method of data entry. No programming is necessary, and the constraint is stored in one location. If you need to change the condition, it is readily accessible (to authorized users).

A second type of constraint is to choose data from a set of predefined options. For example, gender may be listed as male, female, or unavailable. Providing a list helps clerks enter data, and it forces them to enter only the choices provided. For instance, you do not have to worry whether someone might enter *f*, *F*, or *fem*. The data is more consistent.

A third type of data integrity is a bit more complicated but crucial in a relational database. The tables are nicely organized with properties that ensure efficient storage of the data. Yet you need to be able to reconnect the data in the tables to get the reports and forms the users need. Consider the sale example in Figure 3.32 when a clerk enters a customer number in the Sale table. What happens if the clerk enters a customer number that does not exist in the Customer table? If you want to check later on customer purchases, you will be unable to find matching data for that customer. Hence you need a constraint to ensure that when a customer number is entered into the Sale table that number must already exist in the Customer table. The CustomerID in the Sale table is a foreign key in that table, and the constraint you need is known as referential integrity. **Referential integrity** exists



Sale

<u>SaleID</u>	SaleDate	CID	…
1173	1/4	321	
1174	1/5	938	
1185	1/8	337	
1190	1/9	321	
1192	1/9	776	

Customer

<u>CID</u>	Name	Phone	…
321	Jones	998-	
337	Sanchez	773-	
938	Carson	873-	

Simple business rules
- Limits on data ranges
  - Price > 0
  - Salary < 100,000
  - DateHired > 1/12/1995
- Choosing from a set
  - Gender = M, F, Unknown
  - Jurisdiction=City, County, State, Federal

Referential Integrity
- Foreign key values in one table must exist in the master table.
- Sale(SaleID, SaleDate, CID,…)
- CID must exist in the customer table.

Figure 3.32

Data integrity. Integrity can be maintained by simple rules. Relational databases rely on referential integrity constraints to ensure that customer data exists before the customer number can be entered in the Rental table.

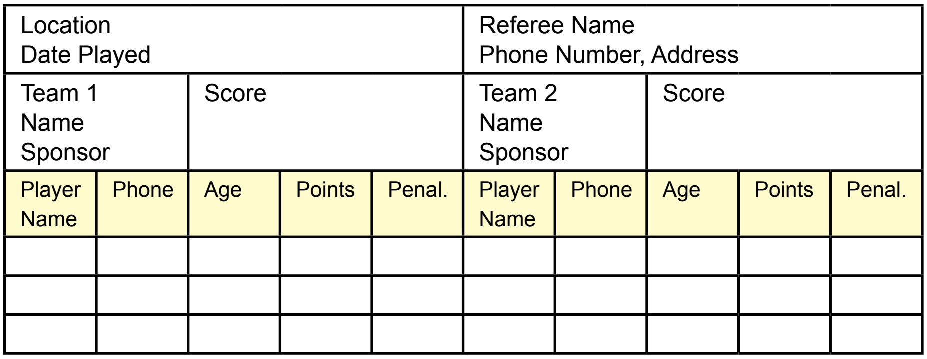
when a value for a foreign key can be entered only if the corresponding value already exists in the originating table.

Essentially, once you define the relationship between tables, you can tell the DBMS to enforce referential integrity. The method for defining referential integrity depends on the specific DBMS. Generally, the constraint is specified in the CREATE TABLE command. Most relational databases also support **cascading delete**, which uses the same concepts. If a user deletes a row in the Customer table, you also need to delete the related entries in the Sale table. Then you need to delete the corresponding rows in the SaleItems table. If you build the relationships and specify cascade on delete, the database will automatically delete the related

Figure 3.33

SQL referential integrity definition. In the Sale table, declaring a column as a foreign key tells the DBMS to check each value in this table to find a matching value in the referenced (e.g., Customer) table.

```
CREATE TABLE Sale
(  SaleID        Integer NOT NULL,
   SaleDate      Date,
   CustomerID    Integer,
     CONSTRAINT pk_Sale PRIMARY KEY (SaleID),
     CONSTRAINT fk_SaleCustomer FOREIGN KEY (CustomerID)
       REFERENCES Customer (CustomerID)
       ON DELETE CASCADE
)
```



Location Date Played					Referee Name Phone Number, Address				
Team 1 Name Sponsor		Score			Team 2 Name Sponsor		Score		
Player Name	Phone	Age	Points	Penal.	Player Name	Phone	Age	Points	Penal.

Figure 3.34

Database design for a soccer league. The design and normalized tables depend on the business rules. Some of the rules are shown on the form.

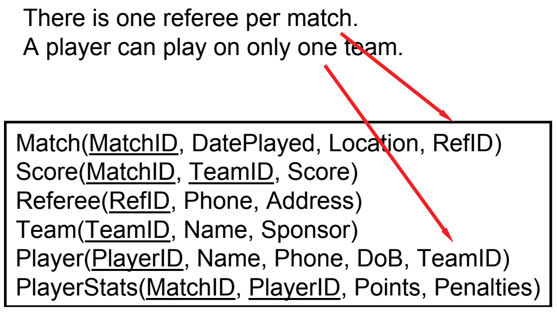
rows when a user deletes an entry in the Customer table. These actions maintain the consistency of the database by ensuring that links between the tables always refer to legitimate rows.

Oracle and SQL Server support referential integrity by declaring a foreign key when you create a table. Figure 3.33 shows the command that can be used to create a Sale table with three columns. The company wants to make sure that all orders are sent to legitimate customers, so the customer number (CustomerID) in the Sale table must exist in the Customer table. The foreign key constraint enforces this relationship. The constraint also specifies that the relationship should handle cascading deletes. Oracle and SQL Server use the standard SQL language to create tables.

When you start to enter data into a DBMS, you will quickly see the role played by referential integrity. Consider two tables: Sale(SaleID, SaleDate, CustomerID) and Customer(CustomerID, Name, Address, etc.). You have a referential integrity constraint that links the CustomerID column in the Sale table to the CustomerID column in the Customer table. As you enter sample data, begin with the Sale table. The DBMS will not accept any data—because the corresponding CustomerID must already exist in the Customer table. That is, the referential integrity rules force you to enter data in a certain order. Clearly, these rules would present problems to users, so you cannot expect users to enter data directly into tables. Chapters 6, 7, and 8 explain how forms and applications will automatically ensure that the user enters data in the proper sequence.

The Effects of Business Rules

How do business rules change the database design? It is important to understand how different business rules affect the database design and the normalization process. As a database designer, you must identify the basic rules and build the database to match them. However, be careful because business rules can change. If you think a current business rule is too restrictive, you should design the database with a more flexible structure.



There is one referee per match.
A player can play on only one team.

Match(MatchID, DatePlayed, Location, RefID)
Score(MatchID, TeamID, Score)
Referee(RefID, Phone, Address)
Team(TeamID, Name, Sponsor)
Player(PlayerID, Name, Phone, DoB, TeamID)
PlayerStats(MatchID, PlayerID, Points, Penalties)

Figure 3.35

Restrictive rules. With only one referee per match, the referee key is added to the Match table. Similarly, the TeamID column is placed in the Player table.

Consider the example shown in Figure 3.34. The local parks and recreation department runs a soccer league and collects basic statistics at the end of every match. You need to design the data tables for this problem.

To illustrate the effect of different rules, consider the two main rules and the resulting tables displayed in Figure 3.35. The first rule states that there can be only one referee per match. Hence the RefID can be placed in the Match table. Note that it is not part of the primary key. The second rule states that a player can play on only one team; therefore, the appropriate TeamID can be placed in the Player table.

Now consider what happens if these two rules are relaxed as shown in Figure 3.36. The department manager believes that some day there might be several referees per match. Also, the issue of substitute players presents a problem. A substitute might play on several different teams in a season—but only for one team during a match. To handle these new rules, the key values must change. You might be tempted to make the simple changes indicated in Figure 3.36; that is, make RefID part of the key in the Match table and make TeamID part of the primary key in the Player table. Now each Match can have many Referees, and each Player can play on many teams. The problem with this approach is that the Match and Player

Figure 3.36

Relaxing the rules to allow many-to-many relationships. You might try to make the RefID and TeamID columns part of the primary key, but the resulting tables are not in 3NF. Location does not depend on RefID, and Player Name does not depend on TeamID.

There can be several referees per match.
A player can play on several teams (substitute), but only one team per match.

Match(MatchID, DatePlayed, Location, RefID)
Score(MatchID, TeamID, Score)
Referee(RefID, Phone, Address)
Team(TeamID, Name, Sponsor)
Player(PlayerID, Name, Phone, DoB, TeamID)
PlayerStats(MatchID, PlayerID, Points, Penalties)

There can be several referees per match.
A player can play on several teams (substitute), but only one team per match.

Match(MatchID, DatePlayed, Location)
RefereeMatch(MatchID, RefID)
Score(MatchID, TeamID, Score)
Referee(RefID, Phone, Address)
Team(TeamID, Name, Sponsor)
Player(PlayerID, Name, Phone, DoB)
PlayerStats(MatchID, PlayerID, TeamID, Points, Penalties)

Figure 3.37

Relaxing the rules and normalizing the tables. The RefereeMatch table enables the department to have more than one referee per match. Moving the TeamID to the PlayerStats table indicates that someone can play for more than one team—but for only one team during a given match.

tables are no longer in 3NF. For example, DatePlayed does not depend on RefID. Likewise, Name in the Player table does not depend on the TeamID. For example, Paul Ruiz does not change his name every time he plays on a different team.

The solution is displayed in Figure 3.37. A new table is added to handle the many-to-many relationship between referees and matches. Similarly, the player's TeamID is moved to the PlayerStats table, but it is not part of the primary key. In this solution, each match has many players, and players can participate in many matches. Yet, for each match, each player plays for only one team. This new database design is different from the initial design. More importantly it is less restrictive. As a designer, you must look ahead and build the database so that it can handle future needs of the department.

Which of these database designs is correct? The answer depends on the needs of the department. In practice, it would be wiser to choose the more flexible design that can assign several referees to a match and allows players to substitute for different teams throughout the season. However, in practice you should make one minor change to this database design. If no matches have been played, how do you know which players are on each team? As it stands, the database cannot answer this question. The solution is to add a BaseTeamID to the Player table. At the start of the season, each team will submit a roster that lists the initial team members. Players can be listed on only one initial team roster. If someone substitutes or changes teams, the data can be recorded in the PlayerStats table.

Converting a Class Diagram to Normalized Tables

What problems arise when converting a class diagram to normalized tables? Each normalized table represents a business entity or class. Hence a class diagram can be converted into a list of normalized tables. Likewise, a list of normalized tables can be drawn as a class diagram. Technically, the entities in a class diagram do not have to be in 3NF (or higher). Some designers use a class diagram as an overview, or big picture, of the business, and they leave out some of the normalized details. In this situation you will have to convert the classes into a list of normalized tables. As noted in Chapter 2, some features commonly arise on a class diagram, so you should learn how to handle these basic conversions.

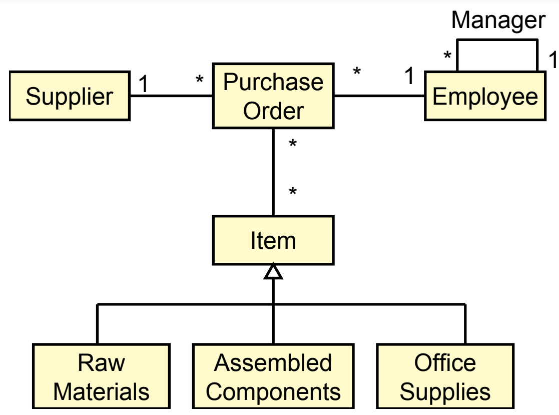


Figure 3.38

Converting a class diagram to normalized tables. Note the four types of relationships: (1) one-to-many, (2) many-to-many, (3) subtype, and (4) recursive.

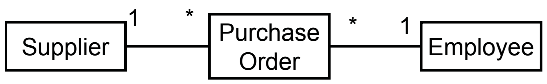
The most challenging problems you will encounter are from class diagrams that utilize object-oriented features, such as subclasses and composition. Some relational database systems have added object-oriented features to make it easier to handle these issues. For example, you can store object data in a cell. However, storing object data (including XML) in a cell often violates the first rule of normalization. From the database perspective, the data is no longer atomic, but requires special routines to examine and compare the data within a cell. Sometimes it makes sense to use these extensions, but you will have to weigh the tradeoffs. For example, most systems enable you to define customized data types. A common use is for spatial data where a location is stored as a single GPS (latitude, longitude, and altitude) coordinate instead of three separate columns. For the approach to be successful, you (or the DBMS vendor) need to write customized functions to use this new object type. Some objects, such as location, are commonly used by many organizations, so it does make sense. Other, highly customized objects, could require considerable additional effort, and you need to evaluate the tradeoffs before creating the custom objects.

Figure 3.38 illustrates a typical class diagram for a purchase order with four basic types of relationships: (1) a one-to-many relationship between supplier and the purchase order, (2) a many-to-many relationship between the purchase order and the items, (3) a subtype relationship that contains different attributes, and (4) a recursive relationship within the Employee entity to indicate that some employees are managers of others.

One-to-Many Relationships

The most important rule in converting class diagrams to normalized tables is that relationships are handled by placing a common column in each of the related tables. This column is usually a key column in one of the tables. This process is easy to see with one-to-many relationships.

The purchase order example has two one-to-many relationships. (1) Many different purchase orders can be sent to each supplier, but only one supplier appears on a purchase order. (2) Each purchase order is created by only one employee, but



Supplier(SID, Name, Address, City, State, Zip, Phone)
Employee(EID, Name, Salary, Address, …)

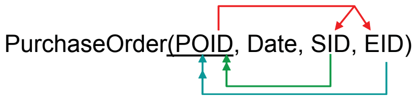
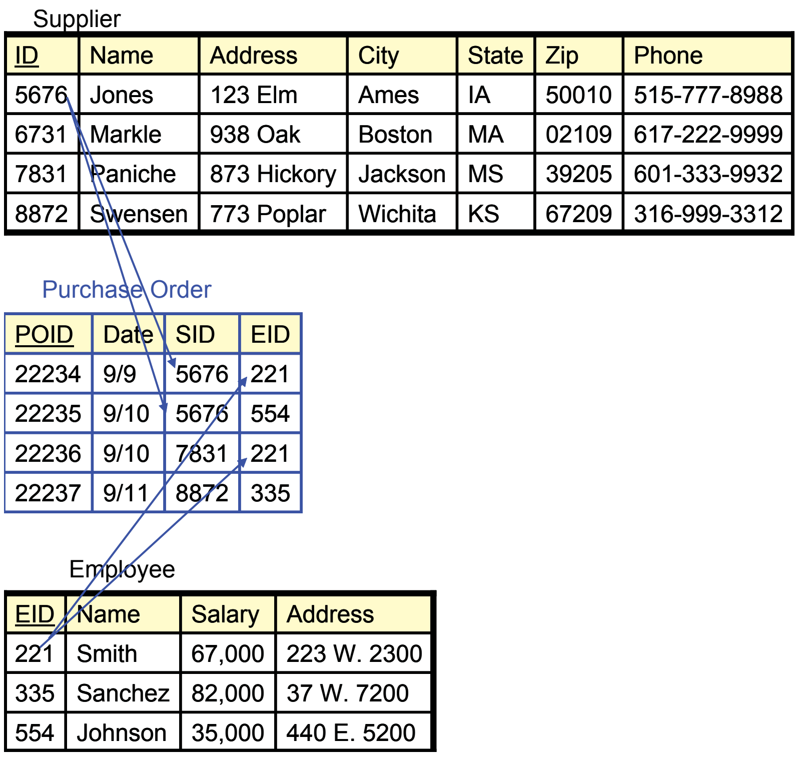

PurchaseOrder(POID, Date, SID, EID)

Figure 3.39

Converting one-to-many relationships. Add the primary key from the one-side into the many-side table. In the example SID and EID are added to the PurchaseOrder table. Note that they are not primary keys in the PurchaseOrder table.

Figure 3.40

Sample data for one-to-many relationships. The Supplier and PurchaseOrder tables are linked through the SID column. Similarly, the Employee table is linked through the data in the EID column. Both the SID and EID columns are foreign keys in the PurchaseOrder table, but they are not primary keys in that table.



Supplier

ID	Name	Address	City	State	Zip	Phone
5676	Jones	123 Elm	Ames	IA	50010	515-777-8988
6731	Markle	938 Oak	Boston	MA	02109	617-222-9999
7831	Paniche	873 Hickory	Jackson	MS	39205	601-333-9932
8872	Swensen	773 Poplar	Wichita	KS	67209	316-999-3312

Purchase Order

POID	Date	SID	EID
22234	9/9	5676	221
22235	9/10	5676	554
22236	9/10	7831	221
22237	9/11	8872	335

Employee

EID	Name	Salary	Address
221	Smith	67,000	223 W. 2300
335	Sanchez	82,000	37 W. 7200
554	Johnson	35,000	440 E. 5200

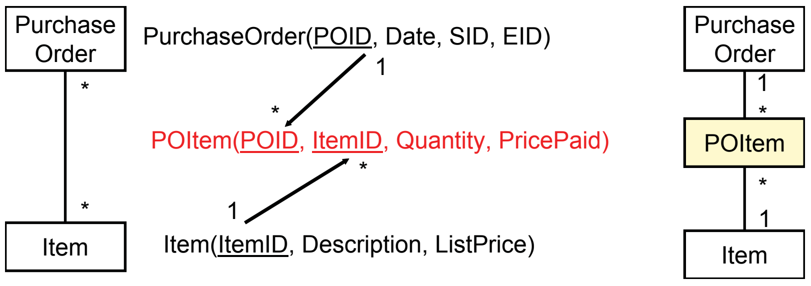

Purchase Order
*
*
Item
PurchaseOrder(POID, Date, SID, EID)
1
*
POItem(POID, ItemID, Quantity, PricePaid)
*
1
Item(ItemID, Description, ListPrice)
Purchase Order
1
*
POItem
*
1
Item

Figure 3.41

Converting a many-to-many relationship. Many-to-many relationships use a new, intermediate table to link the two tables. The new POItem table contains the primary keys from both the PurchaseOrder and Item tables.

an employee can create many purchase orders. To create the normalized tables, first create a primary key for each entity (Supplier, Employee, and PurchaseOrder). As shown in Figure 3.39, the normalized tables can be linked by placing the Supplier key (SID) and Employee key (EID) into the PurchaseOrder table. Note carefully that all class diagram associations are expressed as relationships between keys.

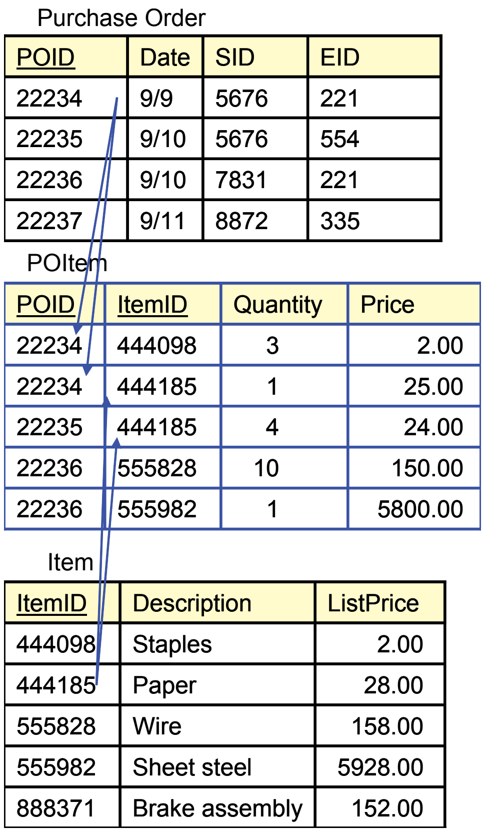
Note also that SID and EID are not key columns in the PurchaseOrder table. You can verify which columns should be keyed. Start with the POID column. For each PurchaseOrder (POID), how many suppliers are there? The business rule says only one supplier for a purchase order; therefore, SID should not be keyed, so do not underline SID. Now start with SID and work in the other direction. For each supplier, how many purchase orders are there? The business rule says many purchase orders can be sent to a given supplier, so the PID column needs to be a key. The same process indicates that EID should not be a key; it belongs in the PurchaseOrder table, since each Employee can place many orders. Figure 3.40 uses sample data to show how the tables are linked through the key columns.

Many-to-Many Relationships

Overview class diagrams often contain many-to-many relationships. However, in a relational database many-to-many relationships must be split into two one-to-many relationships to get to BCNF. Figure 3.41 illustrates the process with the PurchaseOrder and Item tables.

Each of the two initial entities becomes a table (PurchaseOrder and Item). The next step is to create a new intermediate table (POItem) that contains the primary keys from both of the other tables (POID and ItemID). This table represents the many-to-many relationship. Each purchase order (POID) can contain many items, so ItemID must be a key. Similarly, each item can be ordered on many purchase orders, so POID must be a key. Think of the PurchaseOrder and Item tables as the base tables that generate the purchase order and item data respectively. If you use generated key columns, new key values will be generated within those two tables when rows are added. The POItem table links the other two by using the existing key values. It indicates the individual items being purchased on a specific order.

You must have a table that contains both POID and ItemID as keys. Can you create this relationship without creating a third table? In most cases the answer is



Purchase Order

<u>POID</u>	Date	SID	EID
22234	9/9	5676	221
22235	9/10	5676	554
22236	9/10	7831	221
22237	9/11	8872	335

POItem

<u>POID</u>	<u>ItemID</u>	Quantity	Price
22234	444098	3	2.00
22234	444185	1	25.00
22235	444185	4	24.00
22236	555828	10	150.00
22236	555982	1	5800.00

Item

<u>ItemID</u>	Description	ListPrice
444098	Staples	2.00
444185	Paper	28.00
555828	Wire	158.00
555982	Sheet steel	5928.00
888371	Brake assembly	152.00

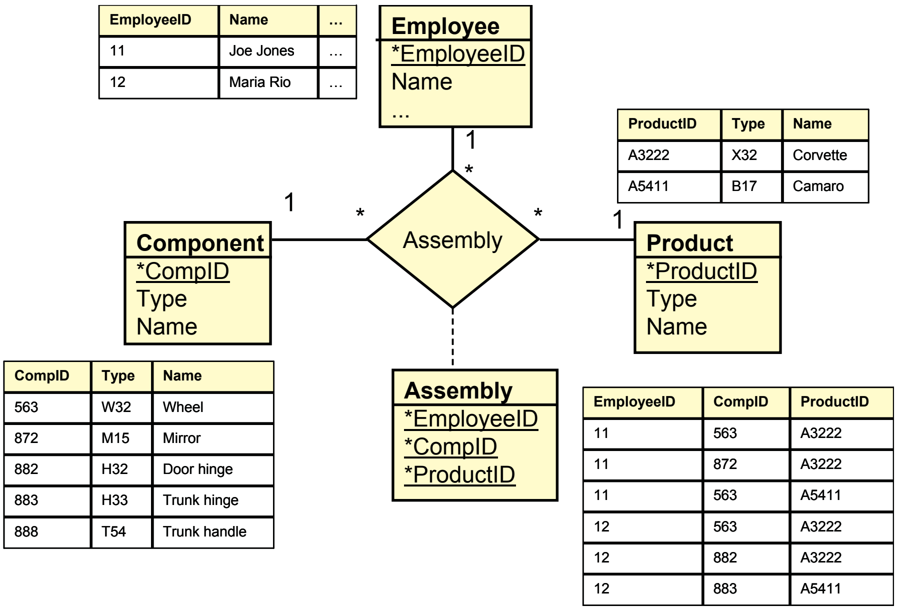
Figure 3.42

Sample data for the many-to-many relationship. Note that the intermediate POItem table links the other two tables. Verify that the three tables are in 3NF, where each nonkey column depends on the whole key and nothing but the key.

no. Consider what happens if you try to put the ItemID column into the Purchase-Order table and make it part of the primary key. The resulting entity would not be a 3NF table, because Date, SID, and EID do not depend on the ItemID. A similar problem arises if you try to place the POID key into the Item table. Hence the intermediate table is required. Figure 3.42 uses sample data to show how the three tables are linked through the keys.

N-ary Associations

As noted in Chapter 2, n-ary associations are denoted with a diamond. This diamond association also becomes a class. In a sense, an n-ary association is simply a set of several binary associations. As shown in Figure 3.43, the new association class holds the primary key from each of the other classes. As long as the binary associations are one-to-many, each column in the Assembly class will be part of the primary key. If for some reason a binary association is one-to-one, then the corresponding column would not be keyed.



EmployeeID	Name	...
11	Joe Jones	...
12	Maria Rio	...

ProductID	Type	Name
A3222	X32	Corvette
A5411	B17	Camaro

CompID	Type	Name
563	W32	Wheel
872	M15	Mirror
882	H32	Door hinge
883	H33	Trunk hinge
888	T54	Trunk handle

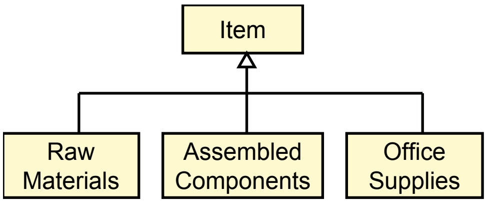
EmployeeID	CompID	ProductID
11	563	A3222
11	872	A3222
11	563	A5411
12	563	A3222
12	882	A3222
12	883	A5411

Figure 3.43

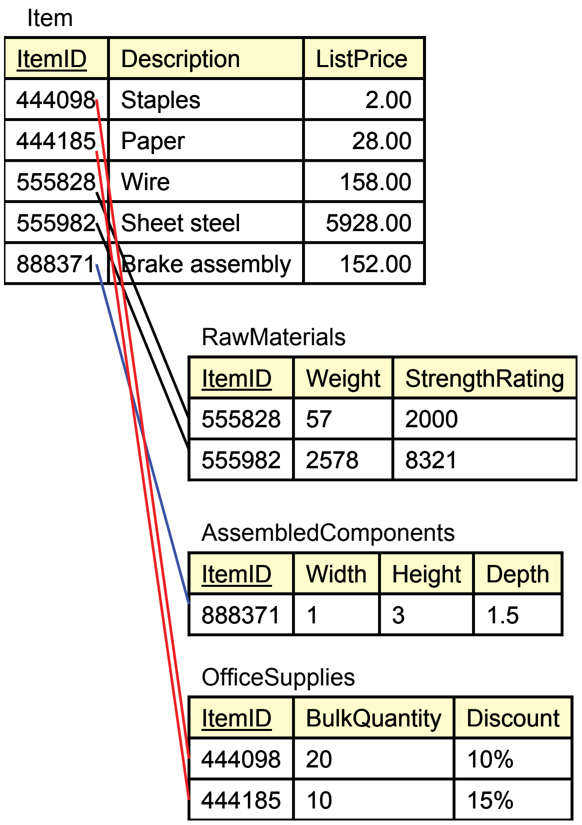
N-ary association. The Assembly association is also a class. It can be modeled as a set of binary (one-to-many) associations. The primary key from each of the main classes is included in the new Assembly class.

Figure 3.44

Converting subtypes. Every item purchased has basic attributes, which are recorded in the Item table. Each item can be placed in one of three categories, which have different attributes. To convert these relationships to 3NF, create new tables for each subtype. Use the same key in the new tables and in the generic table. Add attributes specific to each of the subtypes.



Item(<u>ItemID</u>, Description, ListPrice)
RawMaterials(<u>ItemID</u>, Weight, StrengthRating)
AssembledComponents(<u>ItemID</u>, Width, Height, Depth)
OfficeSupplies(<u>ItemID</u>, BulkQuantity, Discount)



Item

<u>ItemID</u>	Description	ListPrice
444098	Staples	2.00
444185	Paper	28.00
555828	Wire	158.00
555982	Sheet steel	5928.00
888371	Brake assembly	152.00

RawMaterials

<u>ItemID</u>	Weight	StrengthRating
555828	57	2000
555982	2578	8321

AssembledComponents

<u>ItemID</u>	Width	Height	Depth
888371	1	3	1.5

OfficeSupplies

<u>ItemID</u>	BulkQuantity	Discount
444098	20	10%
444185	10	15%

Figure 3.45

Sample data for the subtype relationships. Notice how each Item has an entry in the Item table and a row in one of the three subtype tables.

Generalization or Subtypes

Some business entities are created as subtypes. Figure 3.44 illustrates this relationship with the Item entity. An item is a generic description of something that is purchased. Every item has a description and a list price. However, the company deals with three types of items: raw materials, assembled components, and office supplies. Each of these subtypes has some additional properties that you wish to track. For example, the company tracks the weight of raw materials, the dimension of assembled components, and quantity discounts for office supplies.

Two basic approaches exist for converting this design to a relational database. (1) If subtypes are similar, you could ignore the subclasses and compress all the subclasses into the main class that would contain every property for all of the subclasses. In this case each item entry would have several null values. (2) In most cases a better approach is to create separate tables for each subclass. Each table will contain the primary key from the main Item class.

As shown in Figure 3.45, each item has an entry in the Item table. The Item table contains attributes that apply to all of the subtypes (Description and ListPrice). Each item also has an entry in one of the three subtype tables, depending on the specific type of item. For example, item 444098 is described in the Item table and

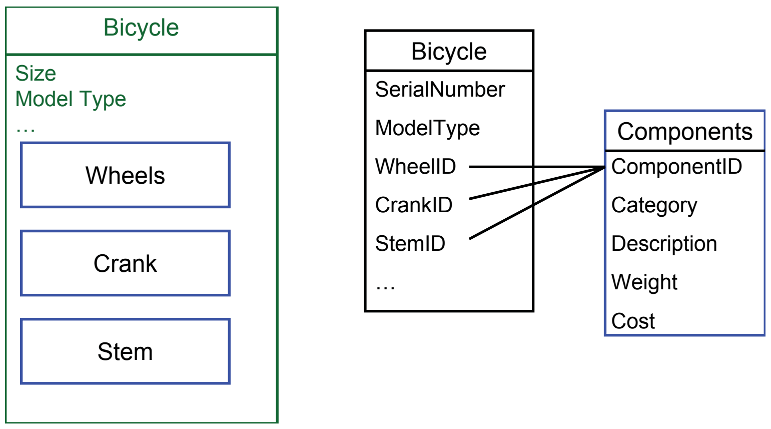


Figure 3.46

Normalizing a composition association. First decide how to handle the subclasses. In this case they are combined into one Components table. Second, handle the composition by storing the ComponentID and the SerialNumber as keys in a new junction table.

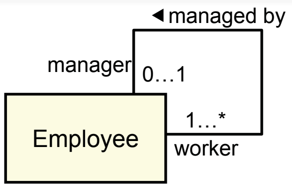
has additional data in the OfficeSupplies table. If the subclass relationships are not mutually exclusive, then each main item can have a matching row in more than one of the subclass tables.

In most cases, it is simpler to ignore the subtypes and put all of the columns into the single Item table. However, lumping the subtypes together could result in a large number of null values. If the amount of wasted space becomes a significant issue, or if you need to assign control of each subtype to a different department, you will need to create the separate tables. The major DBMSs have the ability to define subtables and can process queries with this structure automatically. If subtables are not available, or you choose not to use them, you will have to write queries to join the subtype tables back to the Item table.

Composition

In some ways composition is a combination of an n-ary association and subtypes. Consider the bicycle example in Figure 3.46, in which a bicycle is built from various components. The first decision to make is how to handle the many components. It is a question of subtypes. In this situation the business keeps almost identical data for each component (ID number, description, weight, cost, list price, and so on). Hence a good solution is to compress each subtype into a generic Component class. However, it would also make sense to handle wheels separately because they are a more complex component that is often built from other components.

You can solve the main composition problem by creating properties in the main Bicycle table for each of the component items (WheelID, CrankID, StemID, and so on). These columns are foreign keys in the Bicycle table (but not primary keys). When a bicycle is built, the ID values for the installed components are stored in the appropriate column in the Bicycle table. You can find more details by examining the actual Rolling Thunder database.



Employee(<u>EID</u>, Name, Salary, Address, Manager)

Employee

<u>EID</u>	Name	Salary	Address	Manager
221	Smith	67,000	223 W. 2300	335
335	Sanchez	82,000	37 W. 7200	
554	Johnson	35,000	440 E. 5200	335

Figure 3.47

Converting recursive relationships. An employee can have only one manager, so add a Manager column to the Employee table which contains the EID to point to the manager. In the example, Smith reports to Manager 335 (Sanchez).

Recursive (Reflexive) Associations

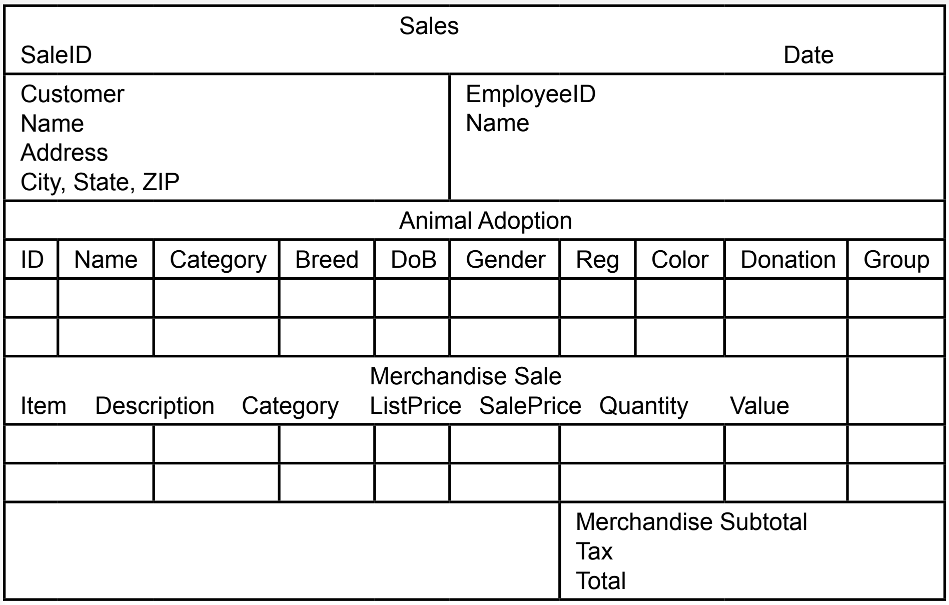
Occasionally, an entity may be linked to itself. A common example is shown in Figure 3.47, where employees have managers. Because managers are also employees, the entity is linked to itself. This relationship is easy to see when you create the corresponding table. Simply add a Manager column to the Employee table. The data in this column consists of an EID. For example, the first employee (Smith, EID 221) reports to manager 335 (Sanchez). Is the Manager column part of the primary key? No, because the business rule states that each employee can have only one manager.

How would you handle a situation in which an employee can have more than one manager? Key the Manager column? That would cause other problems, because the Employee table would not be in BCNF (an employee's address would not depend on the manager). The solution is to create a new table that lists EmployeeID and ManagerID—both part of the primary key. The new table would probably have additional data to describe the relationship between the employee and the manager, such as a project or task.

The Pet Store Example

What tables are needed for the Sally's Pet Store? To design the Sally's Pet Store database, you talk to the owner and investigate the way that other stores operate. In the process you collect ideas for various forms so you can learn the business rules. To expedite the development and hold down costs, you and Sally agree to begin with a simplified model and add features later. The sales form sketched in Figure 3.48 contains the primary data that will be needed when sales are made.

Sally wants you to create separate purchase orders for animals and products. She has repeatedly emphasized the importance of collecting detailed animal data. Eventually, Sally would also like to get medical records for the animals adopted through the store. Common data would include their shots, any illnesses, and any



Sales

SaleID Date

Customer Name Address City, State, ZIP	EmployeeID Name

Animal Adoption

ID	Name	Category	Breed	DoB	Gender	Reg	Color	Donation	Group

Merchandise Sale

Item	Description	Category	ListPrice	SalePrice	Quantity	Value

Merchandise Subtotal
Tax
Total

Figure 3.48

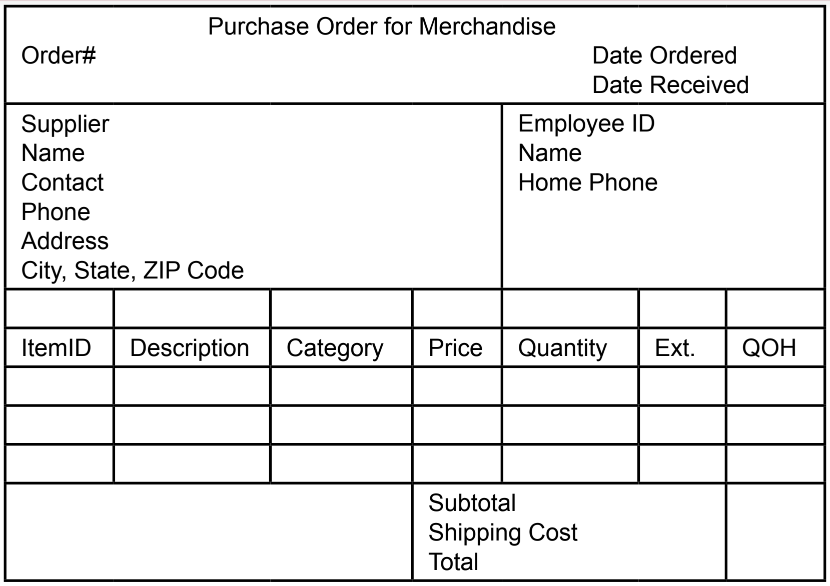
Pet Store sample sales form. Separate sections for selling animals and merchandise reflect a business rule to treat them differently.

medications or treatments they have received. For now, she is relying on the adoption groups to keep this information. However, once the sales and basic financial applications have been created, she wants to add these features to the database.

For the moment the most important job is to collect the transaction data. The design should make it easy to add new attributes for all of the major entities. It should also be easy to add new tables (such as health records) without making major alterations to the initial structure. In addition to sales, purchasing merchandise from suppliers is the other big transaction event.

A sample purchase order form is shown in Figure 3.49. Again, remember that Sally wants to start with a small database. Later you will have to collect additional data. For example, what happens if an order arrives and some items are missing? The current form can only record the arrival of the entire shipment. Similarly, each supplier probably uses a unique set of Item numbers. For example, a case of cat food from one supplier might be ordered with ItemID 3325, but the same case from a different supplier would be ordered with ItemID A9973. Eventually, Sally will probably want to track the numbers used by her major suppliers. That way, when invoices arrive bearing their numbers, matching the products to what she ordered will be easier.

The next step in designing the Pet Store database is to take each form and create a list of normalized tables that will be used to hold data for that form. Figure 3.50 shows the tables that were generated from the Sales form. Before examining the results in detail, you should attempt to normalize the data yourself. Then see whether you derived the same answer. You should also derive the normalized tables for the other two forms. Remember to double-check your work. First make sure the primary keys are correct, then check to see that each nonkey column depends on the whole key and nothing but the key.



Purchase Order for Merchandise

Order#

Date Ordered
Date Received

Supplier Name Contact Phone Address City, State, ZIP Code	Employee ID Name Home Phone

ItemID	Description	Category	Price	Quantity	Ext.	QOH
			Subtotal Shipping Cost Total			

Figure 3.49

Pet Store sample purchase order for merchandise. Note the similarities and differences between the two types of orders. Keep in mind that additional data will have to be collected later.

Note that because each animal is unique, there is no SaleAnimal table—unlike the SaleItem table for merchandise. Because an animal can be adopted only one time, it is possible to put the Donation amount and the SaleID into the Animal table. Think about the keys for a minute. Each AnimalID can be adopted only one time, so SaleID is not a key column. But, multiple animals could be adopted at the same time, so SaleID needs to be in the Animal table where AnimalID is a primary key.

Merchandise is different because an ItemID refers to a relatively generic item—such as a bag of dog food. A bag of dog food with that ItemID can be purchased many times. Physically, it is a different bag, but a specific brand/type/size of dog food always has the same ItemID. Consequently, you need a table that links SaleID and ItemID where each SaleID can have many ItemIDs (key ItemID); and each ItemID can appear on many SaleIDs (key SaleID). In the SaleItem table, both SaleID and ItemID are part of the primary key.

View Integration

How do you combine tables from multiple forms and many developers? Up to this point, database design and normalization have been discussed using individual reports and forms, which is the basic step in designing a database. However, most projects involve many reports and forms. Some projects involve teams of designers, where each person collects forms and reports from different users and departments. Each designer creates the normalized list of tables for the individual forms, and you eventually get several collections of tables related to the same topic. At this point you need to integrate all these tables into one complete, consistent set of table definitions.

```
Sale(SaleID, Date, CustomerID, EmployeeID)
SaleAnimal(SaleID, AnimalID, SalePrice)
SaleItem(SaleID, ItemID, SalePrice, Quantity)
Customer(CustomerID, Name, Address, City, State, Zip)
Employee(EmployeeID, Name)
Animal(AnimalID, Name, Category, Breed, DateOfBirth,
        Gender, Registration, Color, ListPrice)
Merchandise(ItemID, Description, Category, ListPrice)
```

Figure 3.50

Pet Store normalized tables for the basic sales form. You should do the normalization first and see if your results match these tables.

When you are finished with this stage, you will be able to enter the table definitions into the DBMS. Although you might end up with a large list of interrelated tables, this step is generally easier than the initial derivation of the 3NF tables. At this point you collect the tables, make sure everything is named consistently, and consolidate data from similar tables. The basic steps involved in consolidating the tables are as follows:

1. Collect the multiple views (documents, forms, etc.).
2. Create normalized tables for each document.
3. Combine the views into one complete model.

The Pet Store Example

Figure 3.51 illustrates the view integration process for the Pet Store case. The tables generated from the Sale and Purchase forms are listed first. The integration occurs by looking at each table to see which ones contain similar data. A good starting point is to look at the primary keys. If two tables have exactly the same primary keys, the tables should usually be combined. However, be careful. Sometimes the keys are wrong, and sometimes the keys might have slightly different names.

Notice that the Employee table shows up twice in the example. By carefully checking the data in each listing, you can form one new table that contains all of the columns. Hence the Phone and DateHired columns are moved to one table, and the two others are deleted. A similar process can be used for the Supplier, Animal, and Merchandise tables. The goal is to create a complete list of normalized tables that will hold the data for all the forms and reports. Be sure to double-check your work and to verify that the final list of tables is in 3NF or BCNF. Also, make sure that the tables can be joined through related columns of data.

The finalized tables can also be displayed on a detailed class diagram. The class diagram for the Pet Store is shown in Figure 3.52. A strength of the diagram is the ability to show how the classes (tables) are connected through relationships. Double-check the normalization to make sure that the basic forms can be re-created. For example, the sales form will start with the Customer, Employee, and Sale tables. The Animal table holds information about adoptions and donations. Sales of products requires the SaleItem and Merchandise tables. All of these tables can be connected by relationships on their attributes.

Sale(SaleID, Date, CustomerID, EmployeeID)
SaleAnimal(SaleID, AnimalID, SalePrice)
SaleItem(SaleID, ItemID, SalePrice, Quantity)
Customer(CustomerID, Name, Address, City, State, Zip)
Employee(EmployeeID, Name, Phone, DateHired)
Animal(AnimalID, Name, Category, Breed, DateOfBirth, Gender, Registration, Color, ListPrice)
Merchandise(ItemID, Description, Category, ListPrice, Cost)

AnimalOrder(OrderID, OrderDate, ReceiveDate, SupplierID, EmpID, ShipCost)
AnimalOrderItem(OrderID, AnimalID, Cost)
Supplier(SupplierID, Name, Contact, Phone, Address, City, State, Zip)
~~Employee(EmployeeID, Name, Phone, DateHired)~~
~~Animal(AnimalID, Name, Category, Breed, Gender, Registration, Cost)~~

MerchandiseOrder(PONumber, OrderDate, ReceiveDate, SID, EmpID, ShipCost)
MerchandiseOrderItem(PONumber, ItemID, Quantity, Cost)
~~Supplier(SupplierID, Name, Contact, Phone, Address, City, State, Zip)~~
~~Employee(EmployeeID, Name, Phone)~~
~~Merchandise(ItemID, Description, Category, QuantityOnHand)~~

Figure 3.51

Pet Store view integration. Data columns from similar tables can be combined into one table. For example, we need only one Employee table. Look for tables that have the same keys. The goal is to have one set of normalized tables that can hold the data for all the forms and reports.

Most of the relationships are one-to-many relationships, but pay attention to the direction. Access denotes the many side with an infinity (∞) sign. Of course, you first have to identify the proper relationships from the business rules. For instance, there can be many sales to each customer, but a given sale can list only one customer.

This final list shown in the class diagram in Figure 3.52 has three new tables: City, Breed, and Category. These validation tables have been added to simplify data entry and to ensure consistency of data. Without these tables employees would have to repeatedly enter text data for city name, breed, and category. There are two problems with asking people to type in these values: (1) it takes time, and (2) people might enter different data each time. By placing standardized values in these tables, employees can select the proper value from a list. Because the standard value is always copied to the new table, the data will always be entered exactly the same way each time it is used.

Asking the DBMS to enforce the specified relationships raises an interesting issue. The relationships require that data be entered in a specific sequence. The foreign key relationship specifies that a value for the customer must exist in the Customer table before it can be placed in the Sale table. From a business standpoint the rule makes sense; you must first meet customers before you can sell them something. However, this rule may cause problems for clerks who are entering sales data. You need some mechanism to help them enter new Customer data before attempting to enter the Sales data. Chapters 6 and 7 explain one way to resolve this issue.

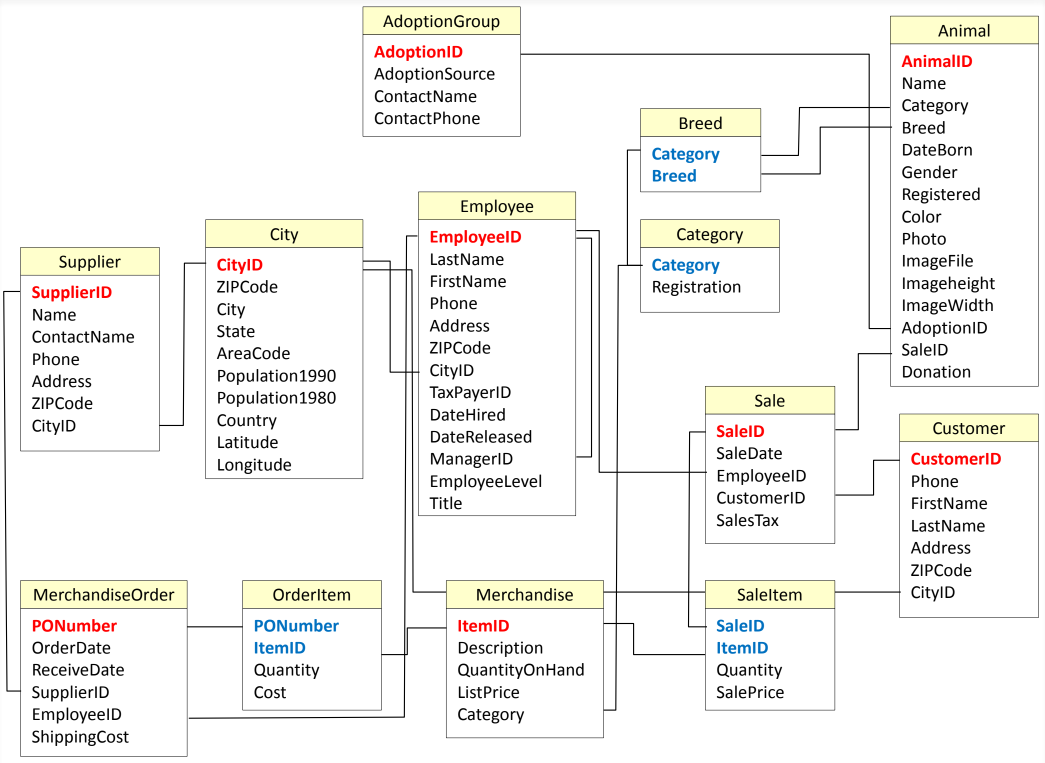


Figure 3.52

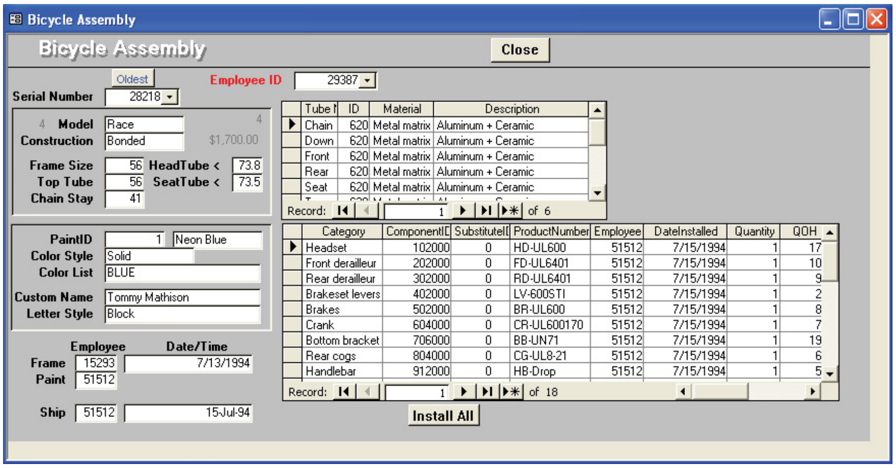
Pet Store class diagram. The tables become entities in the diagram. The relationships verify that the tables are interconnected through the data. Some new data has been added for the employees. Also, cities have been defined in a single table to simplify data entry. Likewise, the new Breed and Category tables ensure consistency of data.

Rolling Thunder Sample Integration Problem

The only way to learn database design and understand normalization is to work through more problems. To practice the concepts of data normalization and to illustrate the methods involved in combining sets of tables, consider a new problem involving a database for a small manufacturer: Rolling Thunder Bicycles. The company builds custom bicycles. Frames are built and painted in-house. Components are purchased from manufacturers and assembled on the bicycles according to the customer orders. Components (cranks, pedals, derailleurs, etc.) are typically organized into groups so that the customer orders an entire package of components without having to specify every single item. Additional details about bicycles and the company operations are available in the Rolling Thunder database.

To understand normalization and the process of integrating tables from various perspectives, consider four of the input forms: Bicycle Assembly, Manufacturer Transactions, Purchase Orders, and Components.

Builders use the Bicycle Assemble form shown in Figure 3.53 to determine the basic layout of the frame, the desired paint styles, and the components that need to be installed. As the frame is built and the components are installed, the workers check off the operations. The employee identification and the date/time are stored



Employee ID	29387
Serial Number	28218
Model	Race
Construction	Bonded
	$1,700.00
Frame Size	56
HeadTube <	73.8
Top Tube	56
SeatTube <	73.5
Chain Stay	41
PaintID	1 Neon Blue
Color Style	Solid
Color List	BLUE
Custom Name	Tommy Mathison
Letter Style	Block

	Employee	Date/Time
Frame	15293	7/13/1994
Paint	51512	
Ship	51512	15-Jul-94

Tube I	ID	Material	Description
Chain	620	Metal matrix	Aluminum + Ceramic
Down	620	Metal matrix	Aluminum + Ceramic
Front	620	Metal matrix	Aluminum + Ceramic
Rear	620	Metal matrix	Aluminum + Ceramic
Seat	620	Metal matrix	Aluminum + Ceramic

Record: 1 of 6

Category	ComponentID	SubstituteI	ProductNumber	Employee	DateInstalled	Quantity	QOH
Headset	102000	0	HD-UL600	51512	7/15/1994	1	17
Front derailleur	202000	0	FD-UL6401	51512	7/15/1994	1	10
Rear derailleur	302000	0	RD-UL6401	51512	7/15/1994	1	9
Brakeset levers	402000	0	LV-600STI	51512	7/15/1994	1	2
Brakes	502000	0	BR-UL600	51512	7/15/1994	1	8
Crank	604000	0	CR-UL600170	51512	7/15/1994	1	7
Bottom bracket	706000	0	BB-UN71	51512	7/15/1994	1	19
Rear cogs	804000	0	CG-UL8-21	51512	7/15/1994	1	6
Handlebar	912000	0	HB-Drop	51512	7/15/1994	1	5

Record: 1 of 18

Figure 3.53

Bicycle Assembly form. The main EmployeeID control is not stored directly, but the value is entered in the FrameAssembler column of the Bicycle table when the employee clicks the Frame box.

in the database. As the parts are installed, the inventory count is automatically decreased. When the bicycle is shipped, a trigger executes code that records the price owed by the customer so a bill can be printed and sent.

Collecting the data columns from the form results in the notation displayed in Figure 3.54. Notice that two repeating groups (tubes and components) occur, but they repeat independently of each other. They are not nested.

Components and other supplies are purchased from manufacturers. Orders are placed as supplies run low and are recorded on a Purchase Order form. Shown in Figure 3.55, the Purchase Order contains standard data on the manufacturer, along with a list of components (or other supplies) that are ordered.

The notation and the 4NF tables are derived in Figure 3.56. For practice you should work through the normalization on your own. Note that the computed columns do not need to be stored. However, be careful to store the shipping cost and discount, since those might be negotiated specifically on each order.

Payments to manufacturers are collected with a basic transaction form shown in Figure 3.57. Note that the initial balance and balance due are computed by code behind the form to display the effects of adding new transactions. Row entries for purchases are automatically generated by the Purchase Order form, so this form is generally used for payments or for corrections.

The 4NF tables resulting from the manufacturer transactions are shown in Figure 3.58. Again, work through the normalization yourself. Practice and experience are the best ways to learn normalization. Do not be misled: It is always tempting to read the "answers" in the book and say that normalization is easy. Normalization becomes much more complex when you face a blank page. Investigating and determining business rules is challenging when you begin.

The Component form in Figure 3.59 is used to add new components to the list and modify the descriptions of the components. It can also be used to make changes to the manufacturer data. Notice the use of two identification numbers: one is

BicycleAssembly(SerialNumber, Model, Construction, FrameSize, TopTube,
    ChainStay, HeadTube, SeatTube, PaintID, PaintColor, ColorStyle, ColorList,
    CustomName, LetterStyle, EmpFrame, EmpPaint, BuildDate, ShipDate,
  (Tube, TubeType, TubeMaterial, TubeDescription),
  (CompCategory, ComponentID, SubstID, ProdNumber, EmpInstall, DateInstall,
    Quantity, QOH)

Bicycle(<u>SerialNumber</u>, Model, Construction, FrameSize, TopTube, ChainStay,
    HeadTube, SeatTube, PaintID, ColorStyle, CustomName, LetterStyle,
    EmpFrame, EmpPaint, BuildDate, ShipDate)
Paint(<u>PaintID</u>, ColorList)
BikeTubes(<u>SerialNumber</u>, TubeID, Quantity)
TubeMaterial(<u>TubeID</u>, Type, Material, Description)
BikeParts(<u>SerialNumber</u>, <u>ComponentID</u>, SubstID, Quantity, DateInstalled, EmpInstalled)
Component(<u>ComponentID</u>, ProdNumber, Category, QOH)

Figure 3.54

Notation for the BicycleAssembly form. There are two repeating groups, but they are independent. The 4NF tables from this form are displayed, but you should try to derive the tables yourself.

assigned by Rolling Thunder, and the other is assigned by the manufacturer. Assigning our own number ensures consistency of the data format and guarantees a unique identifier. The manufacturer's product number is used to help place orders, since the manufacturer would have no use for our internal data.

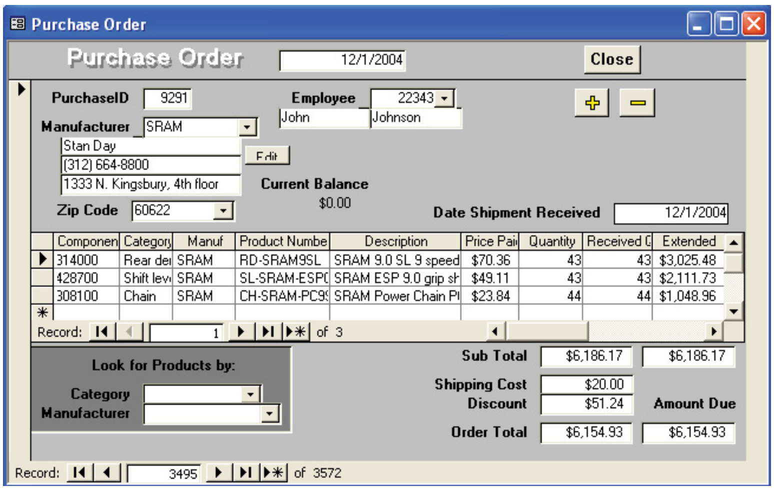
The 4NF tables derived from the Component form are shown in Figure 3.60. For the most part they are straightforward. One interesting difference in Rolling Thunder is the treatment of addresses and cities. Many business tables for customers, employees, suppliers, and so on, contain columns for city, state, and ZIP code. Technically, there is a hidden dependency in this basic data because the three are

Figure 3.55

Tables from the Purchase Order form. Note that the computed columns (extension is price * quantity) are not stored in the tables.

PurchaseOrder(<u>PurchaseID</u>, PODate, EmployeeID, FirstName, LastName,
    ManufacturerID, MfgName, Address, Phone, CityID, CurrentBalance,
    ShipReceiveDate,
  (ComponentID, Category,ManufacturerID, ProductNumber, Description, PricePaid,
    Quantity, ReceiveQuantity, ExtendedValue, QOH, ExtendedReceived),
  ShippingCost, Discount

PurchaseOrder(<u>PurchaseID</u>, PODate, EmpoyeeID, ManufacturerID, ShipReceiveDate,
    ShippingCost, Discount)
EmployeeID(<u>EmployeeID</u>, FirstName, LastName)
Manufacturer(<u>ManufacturerID</u>, Name, Address, Phone, Address, CityID, CurrentBalance)
City(<u>CityID</u>, Name, ZIPCode)
PurchaseItem(<u>PurchaseID</u>, <u>ComponentID</u>, Quantity, PricePaid ReceivedQuantity)
Component(<u>ComponentID</u>, Category, ManufacturerID, ProductNumber, Description,
    QOH)



Purchase Order 12/1/2004

PurchaseID 9291
Employee 22343 John Johnson
Manufacturer SRAM
Stan Day
(312) 664-8800
1333 N. Kingsbury, 4th floor
Zip Code 60622
Current Balance $0.00
Date Shipment Received 12/1/2004

Componen	Category	Manuf	Product Numbe	Description	Price Pai	Quantity	Received Q	Extended
314000	Rear der	SRAM	RD-SRAM9SL	SRAM 9.0 SL 9 speed	$70.36	43	43	$3,025.48
428700	Shift lev	SRAM	SL-SRAM-ESP(	SRAM ESP 9.0 grip sh	$49.11	43	43	$2,111.73
308100	Chain	SRAM	CH-SRAM-PC9	SRAM Power Chain Pl	$23.84	44	44	$1,048.96

Record: 1 of 3

Category
Manufacturer

Sub Total $6,186.17 $6,186.17
Shipping Cost $20.00
Discount $51.24
Amount Due
Order Total $6,154.93 $6,154.93

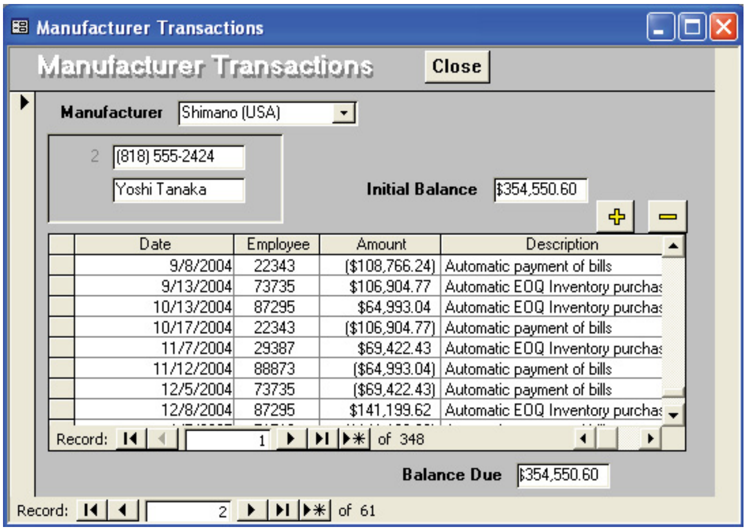
Record: 3495 of 3572

Figure 3.56

Purchase Order form. Only the items ordered is a repeating group. The Look for Products section is a convenience for users and does not store data. The Date Shipment Received box is initially blank and is filled in when the product arrives at the loading dock.

Figure 3.57

Manufacturer Transaction form. The balance due is stored in the database, but only one time. The Initial Balance and Balance Due boxes are computed by the form to display the effect of transactions added by the user.



Manufacturer Transactions

Manufacturer Shimano (USA)
2 (818) 555-2424
Yoshi Tanaka
Initial Balance $354,550.60

Date	Employee	Amount	Description
9/8/2004	22343	($108,766.24)	Automatic payment of bills
9/13/2004	73735	$106,904.77	Automatic EOQ Inventory purchas
10/13/2004	87295	$64,993.04	Automatic EOQ Inventory purchas
10/17/2004	22343	($106,904.77)	Automatic payment of bills
11/7/2004	29387	$69,422.43	Automatic EOQ Inventory purchas
11/12/2004	88873	($64,993.04)	Automatic payment of bills
12/5/2004	73735	($69,422.43)	Automatic payment of bills
12/8/2004	87295	$141,199.62	Automatic EOQ Inventory purchas

Record: 1 of 348

Balance Due $354,550.60

Record: 2 of 61

ManufacturerTransactions(ManufacturerID, Name, Phone, Contact, BalanceDue, (TransDate, Employee, Amount, Description)

Manufacturer(<u>ManufacturerID</u>, Name, Phone, Contact, BalanceDue)
ManufacturerTransaction(<u>ManufacturerID</u>, <u>TransactionDate</u>, EmployeeID, Amount, Description)

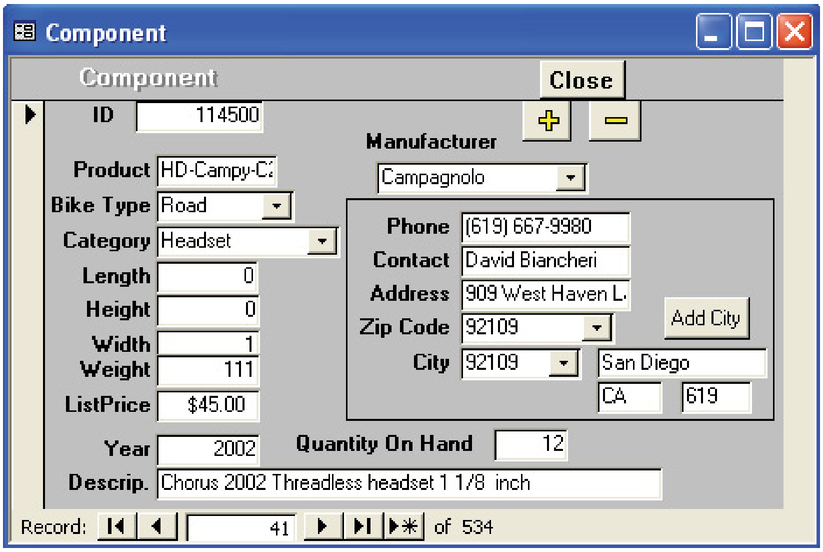
Figure 3.58

Tables for Manufacturer Transaction form. This normalization is straightforward. Note that the TransactionDate column also holds the time, so it is possible to have more than one transaction with a given manufacturer on the same day.

related. Hence a database can save space and data entry time by maintaining a separate City table. Of course, a City table for the entire United States, much less the world, could become large. A more challenging problem is that there is not a one-to-one relationship between cities and ZIP codes. Some cities have many ZIP codes, and some ZIP codes cover multiple cities. Rolling Thunder resolves these two issues by keeping a City table based on a unique CityID. If space is at a premium, the table can be reduced to contain only cities used in the database. As customers arrive from new cities, the basic city data is added. The ZIP code problem is handled by storing a base ZIP code for each city. The specific ZIP code related to each address is stored with the appropriate table (e.g., Manufacturer). This specific ZIP code could also be a nine-digit code that more closely identifies the location of the customer or manufacturer. Although it is possible to create a table

Figure 3.59

Component form. Note that components have an internal ID number that is assigned by Rolling Thunder employees. Products usually also have a Product number that is assigned by the manufacturer. It is difficult to rely on this number, since it might be duplicated across suppliers and the formats vary widely.


Component
Component
Close
ID 114500
Manufacturer
Campagnolo
Product HD-Campy-C:
Bike Type Road
Category Headset
Length 0
Height 0
Width 1
Weight 111
ListPrice $45.00
Phone (619) 667-9980
Contact David Biancheri
Address 909 West Haven L
Zip Code 92109
City 92109 San Diego CA 619
Add City
Year 2002
Quantity On Hand 12
Descrip. Chorus 2002 Threadless headset 1 1/8 inch
Record: 41 of 534

ComponentForm(ComponentID, Product, BikeType, Category, Length, Height, Width, Weight, ListPrice, Description, QOH, ManufacturerID, Name, Phone, Contact, Address, ZIPCode, CityID, City, State, AreaCode)

Component(ComponentID, ProductNumber, BikeType, Category, Length, Height, Width, Weight, ListPrice, Description, QOH, ManufacturerID)
Manufacturer(ManufacturerID, Name, Phone, Contact, Address, ZIPCode, CityID)
City(CityID, City, State, ZIPCode, AreaCode)

Figure 3.60

Tables derived from the Component form. The ZipCode in the Manufacturer table is specific to that company (probably a nine-digit code). The ZipCode in the City table is a base (five-digit) code that can be used for a reference point, but there are often many codes per city.

Figure 3.61

Integrated tables. Duplicate tables have been combined, and normalization (4NF) has been verified. Also draw a class diagram to be sure the tables link together. Note the addition of the Reference column as an audit trail to hold the corresponding PurchaseID. Observe that some tables (e.g., Employee) will need additional data.

Bicycle(SerialNumber, Model, Construction, FrameSize, TopTube, ChainStay, HeadTube, SeatTube, PaintID, ColorStyle, CustomName, LetterStyle, EmpFrame, EmpPaint, BuildDate, ShipDate)
Paint(PaintID, ColorList)
BikeTubes(SerialNumber, TubeID, Quantity)
TubeMaterial(TubeID, Type, Material, Description)
BikeParts(SerialNumber, ComponentID, SubstID, Quantity, DateInstalled, EmpInstalled)
Component(ComponentID, ProductNumber, BikeType, Category, Length, Height, Width, Weight, ListPrice, Description, QOH, ManufacturerID)
PurchaseOrder(PurchaseID, PODate, EmployeeID, ManufacturerID, ShipReceiveDate, ShippingCost, Discount)
PurchaseItem(PurchaseID, ComponentID, Quantity, PricePaid, ReceivedQuantity)
Employee(EmployeeID, FirstName, LastName)
Manufacturer(ManufacturerID, Name, Contact, Address, Phone, CityID, ZipCode, CurrentBalance)
ManufacturerTransaction(ManufacturerID, TransactionDate, EmployeeID, Amount, Description, Reference)
City(CityID, City, State, ZipCode, AreaCode)

PO	Manufacturer(ManufD, Name, Address, Phone, CityID, CurrentBalance)
Mfg	Manufacturer(ManufID, Name, Phone, Contact, BalanceDue)
Comp	Manufacturer(ManufID, Name, Phone, Contact, Address, ZIPCode, CityID)

Figure 3.62

Multiple versions of the Manufacturer table. Tables with the same key should be combined and reduced to one table. Moving Contact and ZipCode to the first table means the other two tables can be deleted. Do not be misled by the two names (CurrentBalance and BalanceDue) for the same column.

of complete nine-digit codes, the size is enormous, and the data tends to change. Companies that rely heavily on nine-digit mailings usually purchase verification software that contains authenticated databases to check their addresses and codes.

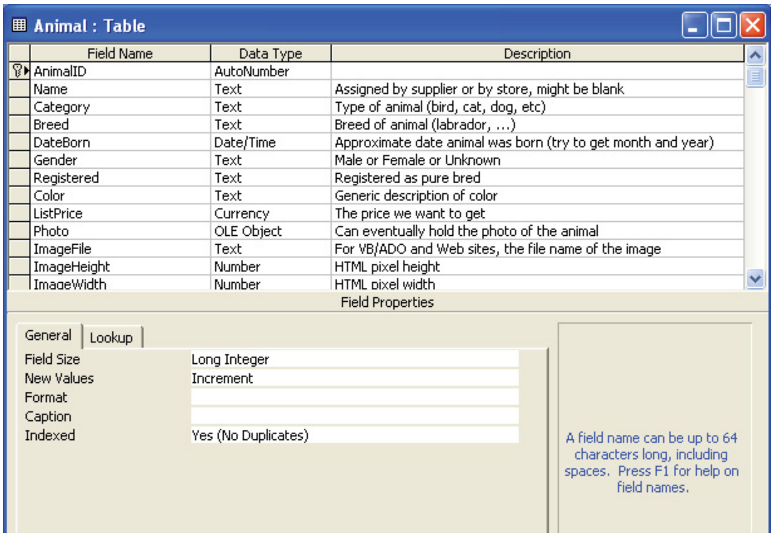
Look at the tables from Figures 3.54, 3.55, 3.59, and 3.60 again. Notice that similar tables are listed in each figure. In particular, look for the Manufacturer tables. Observe that the overlapping tables often contain different data from each form. In practice, particularly when there is a team of designers, similar columns might have different names, so be careful. The objective of this step is to combine the similar tables. The best way to start is to look for common keys. Tables that have the same key columns should be combined. For example, the Manufacturer variations are reproduced in Figure 3.61. The version from the PO table can be extended by adding the Contact and ZIPCode columns from the other variations.

After combining duplicate tables, you should have a single list of tables that contain all of the data from the forms. This list is shown in Figure 3.62. It is also a good idea at this point to double-check your work. In particular, verify that the keys are unique and that composite keys represent many-to-many relationships. Then verify the 3NF rules: Does each nonkey column depend on the whole key and nothing but the key? Also look for hidden dependencies that you might need to make explicit. Be sure that the tables can be linked back together through the data in the columns. You should be able to draw lines between all the tables. Now is a good time to draw a more complete class diagram. Each of the normalized tables becomes an entity. The relationships show how the tables are linked together. (See the Rolling Thunder database for the complete example.)

Finally, examine each table and decide whether you might want to collect additional data. For example, the Employee table would undoubtedly need more data, such as Address and DateHired. Similarly, you saw that the ManufacturerTransaction table could use a Reference column that will contain the PurchaseID when a transaction is automatically generated by the Purchase Order form. This column functions as an audit trail and makes it easier to trace accounting transactions back to the source. Some people might use date/time for the same purpose, but a round-off to seconds could cause problems.

Data Dictionary

How do you record the details for all of the columns and tables? In the process of collecting data and creating normalized tables, be sure to keep a data dictionary to record the data domains and various assumptions you make. A **data dictionary** or **data repository** consists of **metadata**, which is data that describes the data stored in the database. It typically lists all of the tables, columns, data domains,



Animal : Table

Field Name	Data Type	Description
AnimalID	AutoNumber	
Name	Text	Assigned by supplier or by store, might be blank
Category	Text	Type of animal (bird, cat, dog, etc)
Breed	Text	Breed of animal (labrador, ...)
DateBorn	Date/Time	Approximate date animal was born (try to get month and year)
Gender	Text	Male or Female or Unknown
Registered	Text	Registered as pure bred
Color	Text	Generic description of color
ListPrice	Currency	The price we want to get
Photo	OLE Object	Can eventually hold the photo of the animal
ImageFile	Text	For VB/ADO and Web sites, the file name of the image
ImageHeight	Number	HTML pixel height
ImageWidth	Number	HTML pixel width

Field Properties

General | Lookup

Field Size	Long Integer
New Values	Increment
Format	
Caption	
Indexed	Yes (No Duplicates)

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Figure 3.63

Table definition in Microsoft Access. Note the primary key indicator. Also note that text size limits and numeric subtypes are defined in the list at the bottom of the form.

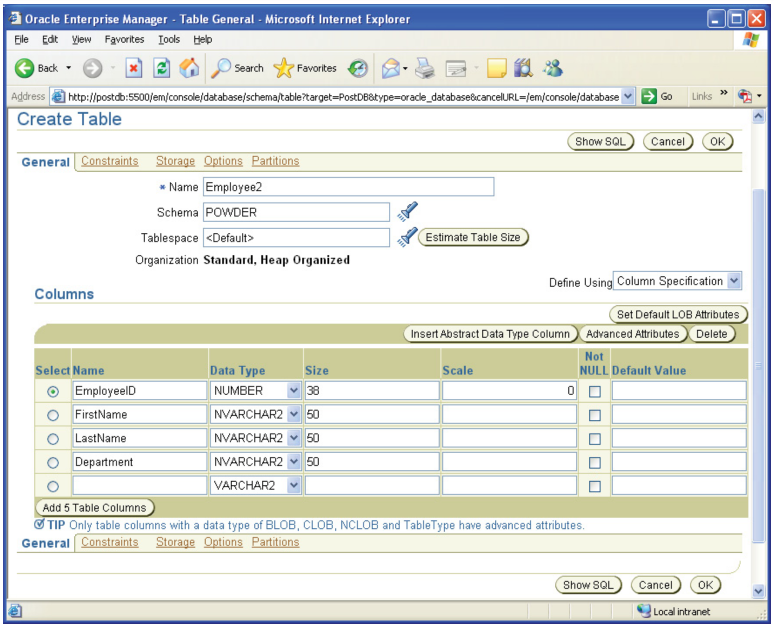
and assumptions. It is possible to store this data in a notebook, but it is easier to organize if it is stored on a computer. Some designers create a separate database to track the underlying project data. Specialized computer tools known as computer-aided software engineering (CASE) tools help with software design. One of their strengths is the ability to create, store, and search a comprehensive data dictionary.

DBMS Table Definition

When the logical tables are defined and you know the domains for all of the columns, you can enter the tables into a DBMS. Most systems have a graphical interface that makes it easier to enter the table definitions. In some cases, however, you might have to use the SQL data definition commands described in Chapter 4. In both cases, the process is similar. Define the table name, enter the column names, select the data type for the column, and then identify the keys. Sometimes keys are defined by creating a separate index. Some systems enable you to create a description for each column and table. This description might contain instructions to users or it might be an extension of your data dictionary to help designers make changes in the future.

At this time, you should determine which keys you want to generate with an autonumber function. Similarly, identify any computed columns and specify the calculations needed for them. Some databases enable you to store these calculations within the database definition; others require that you write them into queries.

You can also set **default values** for each column to speed up data entry. In the video store example, you might set a default value for the base rental rate. Default values can be particularly useful for dates. Most systems enable you to set a default value for dates that automatically enters the current date. At this point you should also set validation rules to enforce data integrity. As soon as the tables are defined, you can set relationships. In Microsoft Access, go to the relationships


Oracle Enterprise Manager - Table General - Microsoft Internet Explorer

Address http://postdb:5500/em/console/database/schema/table?target=PostDB&type=oracle_database&cancelURL=/em/console/database

Create Table

General | Constraints | Storage | Options | Partitions

* Name	Employee2
Schema	POWDER
Tablespace	<Default>
Organization	**Standard, Heap Organized**

Define Using: Column Specification

Columns

Select	Name	Data Type	Size	Scale	Not NULL	Default Value
●	EmployeeID	NUMBER	38	0	☐	
○	FirstName	NVARCHAR2	50		☐	
○	LastName	NVARCHAR2	50		☐	
○	Department	NVARCHAR2	50		☐	
○		VARCHAR2			☐	

TIP Only table columns with a data type of BLOB, CLOB, NCLOB and TableType have advanced attributes.

Figure 3.64

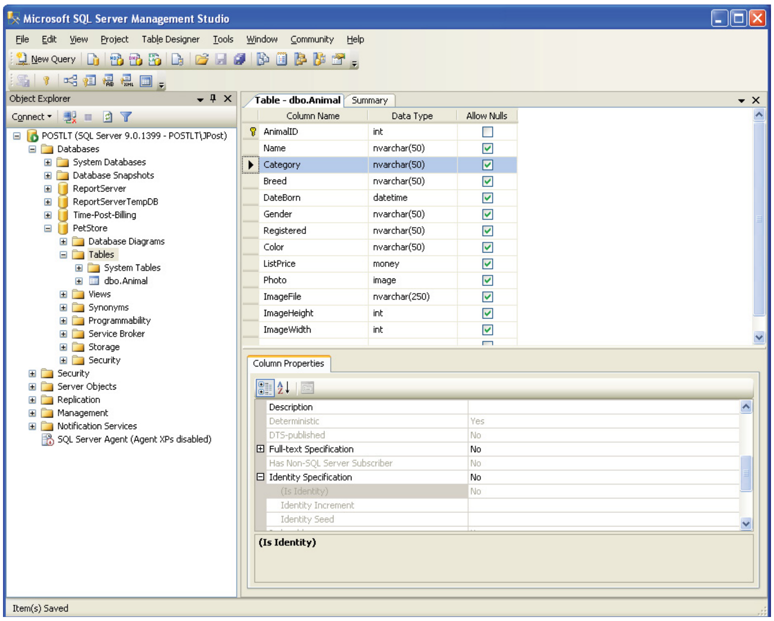
Oracle Schema Manager for creating tables. Primary keys and foreign keys are set in the Constraints tab. For primary key columns, be sure to also check that the values cannot be null.

screen, add all of the tables, then draw lines to show the connections (much like the class diagram). Be sure to check the boxes that specify "Enforce referential integrity," "Cascade on delete," and "Cascade on update." With SQL Server and Oracle, you specify referential integrity as constraints when you define the tables.

Figure 3.63 shows the form that Microsoft Access uses to define tables. Primary keys are set by selecting the appropriate rows and clicking an icon. Data type details such as character length and numeric subtype are set in the list at the bottom of the form. Changes to the table can be made at any time, but if data already exists in the table, you might lose some information if you select a smaller data type.

Figure 3.64 shows the form that can be used within Oracle to create tables—it is the Schema Manager. Primary and foreign keys can be set using the Constraints tab. Keep in mind that once you create a table in Oracle (and SQL Server), it can be difficult to change later. It is always possible to add new columns, but you might not be permitted to change the data type of an existing column or to delete columns.

If you are using Oracle version 8 or above, perform one additional step once the tables have been created by telling it to analyze the tables. For example, use the SQL Plus tool to issue commands for each table similar to: Analyze table Animal compute statistics. These commands tell Oracle to generate statistics for each table (notice the Statistics tab in the Schema Manager). Oracle uses these statistics to dramatically improve performance of queries.


Microsoft SQL Server Management Studio
File Edit View Project Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
Table - dbo.Animal
Summary
Column Properties
Item(s) Saved

POSTLT (SQL Server 9.0.1399 - POSTLT\JPost)
- Databases
  - System Databases
  - Database Snapshots
  - ReportServer
  - ReportServerTempDB
  - Time-Post-Billing
  - PetStore
    - Database Diagrams
    - Tables
      - System Tables
      - dbo.Animal
    - Views
    - Synonyms
    - Programmability
    - Service Broker
    - Storage
    - Security
- Security
- Server Objects
- Replication
- Management
- Notification Services
- SQL Server Agent (Agent XPs disabled)

Column Name	Data Type	Allow Nulls
AnimalID	int	☐
Name	nvarchar(50)	✓
Category	nvarchar(50)	✓
Breed	nvarchar(50)	✓
DateBorn	datetime	✓
Gender	nvarchar(50)	✓
Registered	nvarchar(50)	✓
Color	nvarchar(50)	✓
ListPrice	money	✓
Photo	image	✓
ImageFile	nvarchar(250)	✓
ImageHeight	int	✓
ImageWidth	int	✓

Description	
Deterministic	Yes
DTS-published	No
Full-text Specification	No
Has Non-SQL Server Subscriber	No
Identity Specification	No
(Is Identity)	No
Identity Increment	
Identity Seed	

(Is Identity)

Figure 3.65

Microsoft SQL Server form for creating tables. You can right-click and choose a menu item to create relationships.

Figure 3.65 shows the forms that SQL Server uses to create and edit tables. You can right-click the design to set relationships and constraints. Relationships are defined as constraints and all constraints are stored as separate rules.

It is easy to see the similarities of the database design tools for the various products. Yet more important differences between the systems lie in the data types used within each system. Although some of the data type names appear similar, be particularly careful with Oracle databases. Oracle's underlying data types are different—particularly in dealing with numbers. Over time, the DBMS vendors have loosely adopted the SQL standard data types. In fact, you can use the generic names in almost any DBMS. However, each vendor adds additional types or interesting twists. Designers often work with the vendor's native data types instead of the generic names.

For both Oracle and SQL Server, the graphical forms seem easy to use; however, experienced developers almost always rely on direct SQL statements stored in a text file to create tables. Figure 3.66 gives an example for the Animal table. You simply create a list of all of the table creation statements and store them in a text file. This file is read by the database SQL processor to create the tables. The list has several advantages over the graphical approach: (1) It is easier to change the text file. (2) It is easier to re-create the tables on a different database or different computer. (3) Changing a table definition usually requires creating a new table, copying in the existing data, deleting the old table, and renaming the new one. With the text file, you can quickly define the new table and run the statement to create the table. (4) It is easier to specify the primary key and foreign key relationships in the text file. Most of the graphical approaches are cumbersome and hard

```
CREATE TABLE Animal
(
          AnimalID  INTEGER,
          Name                VARCHAR2(50),
          Category  VARCHAR2(50),
          Breed               VARCHAR2(50),
          DateBorn DATE,
          Gender              VARCHAR2(50)
                    CHECK (Gender='Male' Or Gender='Female'
                              Or Gender='Unknown' Or Gender Is Null)
          Registered          VARCHAR2(50),
          Color               VARCHAR2(50),
          ListPrice  NUMBER(38,4)
                    DEFAULT 0,
          Photo               LONG RAW,
          ImageFile VARCHAR2(250),
          ImageHeight         INTEGER,
          ImageWidth          INTEGER,
                    CONSTRAINT pk_Animal PRIMARY KEY (AnimalID)
                    CONSTRAINT fk_BreedAnimal FOREIGN KEY (Category, Breed)
                    REFERENCES Breed(Category, Breed)
                    ON DELETE CASCADE
          CONSTRAINT fk_CategoryAnimal FOREIGN KEY (Category)
                    REFERENCES Category(Category)
                    ON DELETE CASCADE
);
```

Figure 3.66

Oracle SQL Statements to create the Animal table. The statements for SQL Server are similar—just change the data types.

to read. Also, some versions of Oracle had stricter limits using the graphical interface that could be avoided by creating the SQL statements directly. (5) Because of the foreign key constraints, the order in which the tables are created is critical. You cannot refer to a table in the foreign key constraint unless that table already has been created. For example, the Category and Breed tables must be created before the Animal table. Keeping the table definitions in a text file means you only have to set up the sequence one time. If you are uncertain about the SQL syntax for creating a table, you can examine existing structure files, or you can use the Schema Manager to enter the basic information, then click the Show SQL button to cut and paste the underlying SQL code.

Note that the DB Design tool can automatically generate the CREATE TABLE commands using the data types for each of the major DBMSs. It even analyzes the foreign key references and generates the tables in the proper order. If you prefer to stick with the individual vendor tools, note that almost all of them (except Access) have an option to display the CREATE TABLE command defined by the visual tool. You can copy-and-paste this command into a separate text file for future reference. However, you will have to sort the command into the correct order yourself.

Data Volume and Usage

One more step is required when designing a database: estimating the size of the resulting database. The process is straightforward, but you have to ask a lot of questions of the users. When you design a database, it is important to estimate the overall size and usage of the database. These values enable you to estimate the hardware requirements and cost of the system. The first step is to estimate the size of the tables. Generally, you should investigate three situations: How big is the

Customer(C#, Name, Address, City, State, Zip)
Row: 4 + 15 + 25 + 20 + 2 + 10 = 76

Order(O#, C#, Odate)
Row: 4 + 4 + 8 = 16

OrderItem(O#, P#, Quantity, SalePrice)
Row: 4 + 4 + 4 + 8 = 20

$$Orders\,in\,3\,yrs = 1000\,Customers * \frac{10\,Orders}{Customer} * 3\,yrs = 30{,}000$$

$$OrderLines = 30{,}000\,Orders * \frac{5\,Lines}{Order} = 150{,}000$$

Business rules
- Three year retention.
- 1000 customers.
- Average 10 orders per customer per year.
- Average 5 items per order.

• Customer	76 * 1000	76,000
• Order	16 * 30,000	480,000
• OrderItem	20 * 150,000	3,000,000
• **Total**		**3,556,000**

Figure 3.67

Estimating data volume. First estimate the size of each row, and then estimate the number of rows in the table. If there is a concatenated key, you will usually multiply an average value times the number of rows in a prior table, as in the calculation for OrderItem.

database now? How big will the database be in 2 or 3 years? and How big will the database be in 10 years?

Begin with the list of normalized tables. The process consists of estimating the average number of bytes in each row of the table and then estimating the number of rows in the table. Multiply the two numbers to get an estimate of the size of the table, and then add the table sizes to estimate the total database size. This number represents the minimum size of the database. Many databases will be three to five times larger than this base estimate. Some systems have more complex rules and estimation procedures. For example, Oracle provides a utility to help you estimate the storage required for the database. You still begin with the data types for each column and the approximate number of rows. The utility then uses internal rules about Oracle's procedures to help estimate the total storage space needed.

An example of estimating **data volume** is presented in Figure 3.67. Consider the Customer table. The database system sets aside a certain amount of storage space for each column of data. The amount used depends on the particular system, so consult the documentation for exact values. In the abbreviated Customer table, the identification number takes 4 bytes as a long integer, and you estimate that Names take an average of 15 characters. Other averages are displayed in the table. Better estimates could be obtained from statistical analysis of sample data. In any case the estimated size of one row of Customer data is 76 bytes. Evaluating the business provides an estimate of approximately 1,000 customers; hence, the Customer table would be approximately 76K bytes.

Estimating the size of the Order table follows a similar process, yielding an estimate of 16 bytes per row. Managers might know how many orders are placed in a given year. However, it might be easier to obtain the average number of orders placed by a given customer in 1 year. If that number is 10, then you could expect 10,000 orders in a given year. Similarly, to get the number of rows in the OrderItem table, you need to know the average number of products ordered on one order form. If that number is 5, then you can expect to see 150,000 rows in the OrderItem table in 1 year.

The next step is to estimate the length of time data will be stored. Some companies plan to keep their data online for many years, whereas others follow a strict retention and removal policy. For legal purposes data must be maintained for a certain number of years, depending on its nature. Keep in mind that agencies such as the IRS also require that retrieval software (e.g., the DBMS) be available to reproduce the data.

In addition to the basic data storage, your database will also reserve space for indexes, log files, forms, programs, and backup data. Experience with a particular database system will provide a more specific estimate, but the final total will probably be three to five times the size of the base estimate.

The final number will give you some idea of the hardware needed to support the database. Although performance and prices continue to change, only small databases can be run effectively on personal computers. Larger databases can be moved to a file server on a local area network (LAN). The LAN provides access to the data by multiple users, but performance depends heavily on the size of the database, the characteristics of the DBMS, and the speed of the network. As the database size increases (hundreds or thousands of megabytes), it becomes necessary to move to a dedicated computer to handle the data. Very large databases (terabytes) need multiple computers and specialized disk drives to minimize capacity and performance bottlenecks. The data estimates do not have to be perfect, but they provide basic information that you can give to the planning committee to help allocate funds for development, hardware, software, and personnel.

While you are talking with the users about each table, you should ask them to identify some basic security information. You will eventually need to assign security access rights to each table. Chapter 10 presents the details, but for now you should find out which people use the table, and which people should be denied some privileges. For example, clerks who order merchandise should not be allowed to acknowledge receipt of that merchandise. Otherwise, an unethical clerk could order merchandise, record it as being received, and then steal it. Four basic operations can be granted to data: read it, change it, delete it, or add new data. You should keep a list of who may or may not access each table.

Summary

Database design relies on normalization, or the process of splitting data into tables. Ultimately, each table refers to a single entity or concept. Each table must have a primary key that uniquely identifies each row of data. To create the tables, you begin with a collection of data—generally derived from a user form or report. You reach 1NF by finding the repeating groups of data and putting them in a separate table. Next, you go through each of the intermediate tables and identify primary keys. You reach 2NF by checking each nonkey column and asking whether it depends on the whole key. If not, put the column into a new table along with the portion of the key that it does depend on. To reach 3NF, you check to see whether the nonkey column depends on anything that is not in the key. If so, pull out the column and the dependent column and put them into a new table. BCNF states that you cannot have hidden dependencies—all dependencies must be part of the primary key. 4NF looks at problems with keys, and states that you cannot have two (or more) independent relationships within one table. In all cases, when you find incorrect dependencies or hidden dependencies, you solve the problem by splitting the tables and making the dependency explicit with a primary key.

Each form, report, or description that you collect from a user must be analyzed and a set of 4NF tables defined. For large projects several analysts may be given different forms, resulting in several lists of normalized tables. These tables must then be integrated into one standardized set of normalized data tables. Along the way you must specify the domain, or type of data, for each column. This final list of tables, with any comments, will be entered into the DMBS to start the database construction.

You should also collect estimates of data volume in terms of number of rows for each table. These numbers will enable you to estimate the average and maximum size of the database so that you can choose the proper hardware and software. You should also collect information on security conditions: Who owns the data? Who can have read access? Who can have write access? All of these conditions can be entered into the DBMS when you create the tables.

At this point, after you review your work, you can enter sample data to test your tables. When you are certain that the design is complete and accurate, you can begin building the application by constructing queries and creating forms and reports.

A Developer's View

Miranda learned that the class diagram is converted into a set of normalized tables. These tables are the foundation of the database application. Database design is crucial to developing your application. Engrave the basic normalization rule onto the back of your eyelids: Each nonkey column depends on the whole key and nothing but the key. Since the design depends on the business rules, make certain that you understand the rules. Listen carefully to the users. When in doubt, opt for flexibility. For your class projects, you should now be able to create the list of normalized tables. You should also be able to estimate the size of the database.

Key Terms

atomic
autonumber
Boyce-Codd normal form (BCNF)
cascading delete
composite keys
data dictionary
data integrity
data repository
data volume
default values
deletion anomaly
dependence
domain-key normal form (DKNF)
first normal form (1NF)
foreign key
fourth normal form (4NF)
globally-unique identifier (GUID)
hidden dependency
insertion anomaly
master-detail
metadata
pseudo column
referential integrity
repeating groups
second normal form (2NF)
surrogate keys
third normal form (3NF)

Review Questions

1. What is *dependency*?
2. What are the three main rules for normalization?
3. What problems do you encounter if data is not stored in normalized tables?
4. How are BCNF and 4NF different from 3NF?
5. What are the primary types of data that can be stored in a table?
6. Why is referential integrity important?
7. What complications are caused by setting referential integrity rules?
8. What problems do object-oriented designs cause in a relational database model and how do you compensate for them?
9. What elements do you look for when integrating views?
10. How do you estimate the potential size of a database?

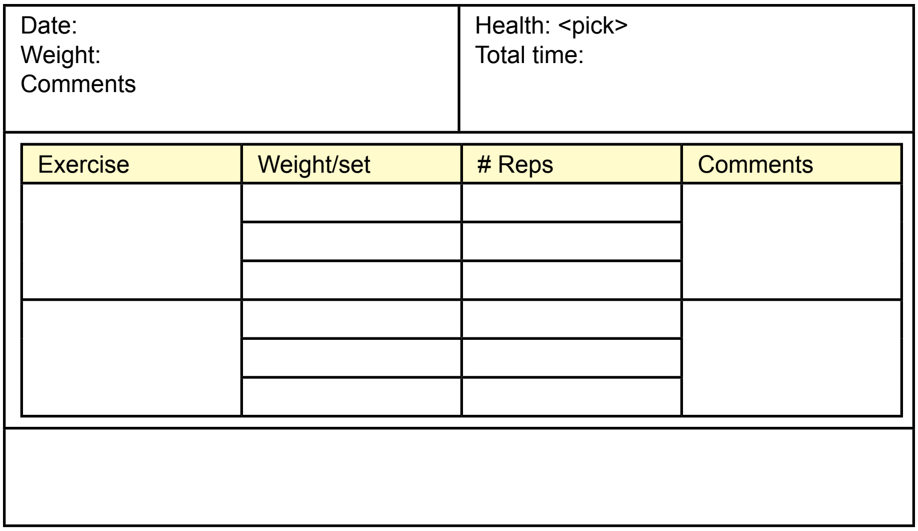
Exercises

1. A local family has a large garden and regularly sells produce at the local Farmer's Market. Up to now the group has just picked items and sold them each week—basically tracking just the amount of money received. Now the family wants to track sales by types of items (potatoes, lettuce, tomatoes, carrots, and so on); both in terms of quantity sold and the amount of money received. They want to use the data to determine planting amounts for the coming year. The crops require about the same level of fertilizer and watering so profits are mainly determined by the yield and the price received. No one wants to create individual item receipts for each sale—that would take too much time, but they will use a tally sheet to record the number of items sold and the prices. Then enter the sales into a computer (or tablet) at the end of the day. Some items are sold by the unit (such as melons or lettuce--bunch) while others, such as carrots, are sold by the pound. The family starts out the day with a set price, but if items are not selling well and have a limited shelf life, the price is reduced. So the amount sold needs to be recorded at each price point.

Market Location Date Family member in charge	Weather comments Crowd comments Competition

Item	Quantity at Start	Sale Price	Quantity Sold	Unit/ Pounds	Comments
Carrots	*20*	*$1.00*	*5*	*Pounds*	*Few buyers*
Carrots		*0.50*	*10*	*Pounds*	
Tomatoes	*40*	*$2.00*	*20*	*Pounds*	
Tomatoes		*$1.50*	*10*	*Pounds*	
Melons	*10*	*$3.00*	*10*	*Units*	*Cantaloupe*

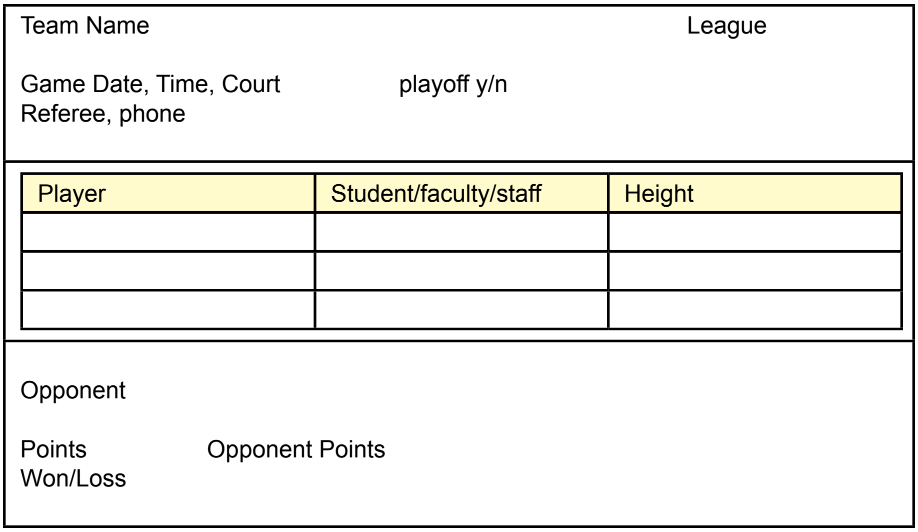
2. Your doctor told you that you need to get more exercise—particularly strength-based. So you decided to start lifting weights at the gym. To make it more interesting, you created an application to track your progress on each exercise. To make it easier to enter data, the application is based on the day and enables you to enter additional information such as health (*excellent, good, tired, weak, sick*), and general comments. For most weight-lifting exercises, you perform multiple lifts (repetitions) within a set, then change the weight for a new set and do another set of repetitions.



Date: Weight: Comments	Health: <pick> Total time:

Exercise	Weight/set	# Reps	Comments

3. You have been hired by the student sports director to build a database for the intramural basketball leagues. The leagues have several teams each year. For the most part the teams get to select the level of competition: A, B, or C. There is more prestige to winning the A (or B) league so the better teams select into that league. There is also a league for teams composed of players all under 6 feet, and there is also a co-ed league that requires at least 2 women on the floor for each game. For the most part, players are assigned to a single team, but they do have the ability to switch to a different team if they desire. So players sign up for each team game but no one tracks points scored by each person. Only the total team points and win/loss are tracked.



Team Name League

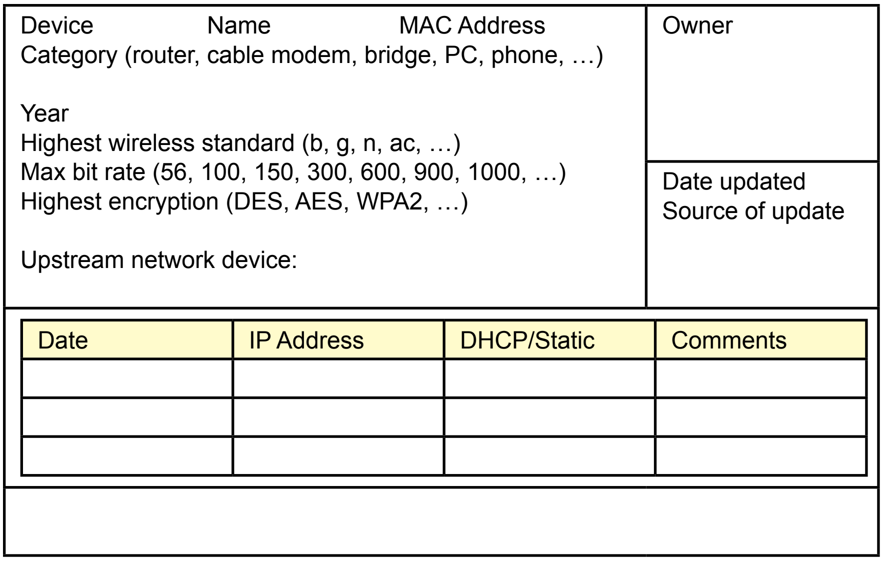
Game Date, Time, Court playoff y/n
Referee, phone

Player	Student/faculty/staff	Height

Opponent

Points Opponent Points
Won/Loss

4. You recently added the twentieth device to your wireless network. Well, it feels like 20, but it might be 10 or even 30; you no longer remember which devices have wireless and when they were last updated. Partly to gain faster speeds (to handle all of the devices), you are planning to replace your main wireless router. Which means you will have to update all of the wireless devices. And then you will have to deal with all of your friends' wireless devices when they come by. You considered buying a commercial network management tool but that seems too expensive. Instead, you want to build a small database application to record basic information about the wireless devices and the IP addresses they are assigned so that you can quickly identify each device when you need it. All devices have a unique MAC (media access control) address which the network uses to identify the item. Sometimes you have to change the MAC address, such as with routers so the upstream device can recognize it (common with cable modems). A big problem with wireless is that older devices do not support newer standards and you have to decide if you want to lower your overall standards to support older devices, or run two or three different wireless systems. You also need to track the owner of the device to decide if you want to remove support for it. Most devices use dynamic assignment (DHCP) to obtain a local IP address so it can change over time. But sometimes you need to set a static address to a device to make it accessible (such as older printers). You only need the IP address once in a while but when you need it you want to record it in case you might need it again. The values you need the most often can eventually be made static.



Device Name MAC Address
Category (router, cable modem, bridge, PC, phone, …)

Year
Highest wireless standard (b, g, n, ac, …)
Max bit rate (56, 100, 150, 300, 600, 900, 1000, …)
Highest encryption (DES, AES, WPA2, …)

Upstream network device:

Owner

Date updated
Source of update

Date	IP Address	DHCP/Static	Comments

5. A friend of yours is starting a business to build semi-customized cases for cell phones and laptops. The focus of the case is in the design—colors, logos, artwork and so on. She plans to buy relatively plain cases and then paint them with various designs. She can even take photographs from customers and incorporate those into the artwork. Existing stores and Web sites focus on finding a case to fit a particular device, but she thinks the process should work the other direction. Customers will pick the design and artwork and then specify the device. Cases for popular devices will be supported automatically and kept in stock, but other devices will take time to customize. The device aspect ratio is a particular problem because the artwork is relatively easy to scale up or down in size but not if the ratio of width to height changes too far. Then it has to be redrawn. And Apple devices tend to use the older 4:3 aspect ratio compared to other companies that use the newer HD 16:9 ratio.

Design Name   Basic colors   File
Description
Category
Aspect Ratio

Base price

Date created
Date modified
Date stopped

Artist
Phone
Commission rate

Standard Stock Devices

Device Name	Price	Quantity on Hand	Device type	Height	Width

Customer name
Phone
Address
City, State, ZIP

Sale Date
Shipment date

Item	Phone/Device	Sale Price	Quantity	Stock/Custom

Payment Method

Subtotal
Tax
Shipping
Total

6. A friend of yours is starting a "vintage" clothing shop. She will buy older clothes (lightly used) from people, clean and mend them if necessary, and then sell them in her store. Some older fashion items often become popular years later and "fashionistas" mix and match items to achieve their own style. But as the inventory grows, your friend needs a better way to track which items are popular to help figure out what price she should pay for clothing and when prices on existing items should be changed. Occasionally, a good customer will ask her to keep an eye out for a special item—particularly in terms of sizes. So she also needs to keep a list of search items. Currently, she tracks items by placing a tag on them when she buys them. The tag includes a basic description of the item, where and when she bought it, and eventually, she adds data to the tag that says when it was sold, the sale price and then the price/cost she paid for the item.

Item Description Category Men/Women/Child Base color Size	Designer/Manufacturer Mfg. Original Price History if known
Purchase Location Purchase Date Amount (coded on tag) Condition	Repairs made:
Sale Date Customer Name E-mail Phone	Sale Price

Search Item Description Designer Color Desired Size Category Estimated Price Substitutions (such as color)	Date Start Date Needed Date Found Location Price Paid Condition
Customer Name Phone E-mail Sale Date Sale Price	

7. You have volunteered to work for a local politician who is a friend of the family. She wants a system to track information about contacts with voters. Most voters in the district do not contact the office, but she wants to be sure to track information on those who do make contact. It is particularly important to track those who have strong opinions on various legislation. It is also important to track those who need rides to the polls each year or help with absentee ballots. And it is absolutely critical to track the requests for money sent to each person and the amount received each time. Note that houses are assigned to districts and regions; but those designations can change over time. Donors are often organized into groups where one person (a consolidator) collects money from other donors. Over time, voters and donors are assigned various tags (such as whale, whiner, or various issues) which are used to target future letters to each person.

Voter Last Name, First Name, Gender Phone Address, City, State, ZIP District #, Region	Party (repub/democ/ind) Probability of voting way we want:

Contact from voter

Date/Time	Method	Reason	Importance	Staff member Phone Vol/Paid

Fund-raising campaign
Campaign year

Event/Mailing/Request

Date	Event/Mailing	Primary topic	Location	Target

Donor Name E-mail Consolidator Labels/Tags	Amount	Date	Event	Comments

Sally's Pet Store

8. Define the tables needed to extend the Pet Store database to handle genealogy records for the animals.

9. Define the tables needed to extend the Pet Store database to handle health and veterinary records for the animals.

10. Sally wants to add payroll and monthly employee evaluation information to the database. Define the tables needed.

11. Sally wants to add pet grooming services. Define the tables necessary to schedule appointments, assuming two workers will be dedicated to this area.

Rolling Thunder Bicycles

12. Using the class diagram, identify five business rules that are described by the table definitions and table relationships (similar to the ListPrice rules described by the Sale example).

13. The company wishes to add more data for human resources, such as tax withholding, benefits selected, and benefit payments by the employees and by the company. Research common methods of handling this type of data and define the required tables.

Corner Med

14. Physicians and medical administrators are often interested in a hierarchical classification of illnesses and diagnoses. Some of the hierarchy is built into the ICD codes, but the managers and physicians want to be able to create reports that roll up the weekly diagnoses into specific categories. They also want the ability to define new categories. Essentially, the physician administrators will create a medical category and list the various conditions that apply to the category. For example, Broken Bones could be a general category, and specific fractures (e.g., S62.2 Fracture of first metacarpal bone using ICD10) would comprise the list of conditions. Define the table(s) needed to handle this summarization data. Optional: What if the physicians what to create multiple levels within the summary data? For example, Family Practice could be a parent category to Childhood Diseases, Accidents, Minor Illnesses, and Checkups. Each of these could have subcategories.

15. The physicians would like to add another step in the patient examination process. They want more complete records and the ability to handle cases where the diagnosis is not immediately available. Specifically, the physicians want to record the symptoms described by the patient at each visit. This record would also include the severity of the symptom and whether it was observed by the physician (or nurse). At each visit the patient's weight, blood pressure, and heart rate are also recorded (children are also measured for height). Along the same lines, they want to record any tests taken and the results of the tests. The tests can include simple physical tests such as reflexes as well as chemical tests. Define the tables and modify the class diagram to handle this additional data.

16. In 2006, Benjamin Brewer, M.D., a practicing physician, listed common statistics for a medical office ["A Doctor Faces Tough Decision to Stop Taking New Patients," The Wall Street Journal, February 7, 2006]. Read the article and use the numbers to estimate the size of the database after 1 year and 5 years of operation. Some basic data from the article: 3 physicians, 2,500 patients, 90 patients per week in office visits per physician. But, read the article to gain perspective on the situation. It is available in your library.

Web Site References

http://ibmdatamag.com/	IBM Data Magazine
http://www.for.gov.bc.ca/his/datadmin/	Canadian Ministry of Forests data administration site, with useful information on data administration and design. Start with the development standards.
http://support.microsoft.com/kb/100139/en-us	Introduction to normalization.
http://www.phpbuilder.com/columns/barry20000731.php3	Normalization examples.

Additional Reading

Date, C. J., *An Introduction to Database Systems*, 8th ed. Reading: Addison-Wesley, 2003. [A classic higher-level textbook that covers many details of normalization and databases.]

Fagin, R. "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems*, 2 no. 3 (September 1977), pp. 262-278. [A classic paper in the development of normal forms.]

Fagin, R. "A Normal Form for Relational Databases That Is Based on Domains and Keys," *ACM Transactions on Database Systems*, 6 no. 3 (September 1981), pp 387-415. [The paper that initially described domain-key normal form.]

Kent, W. "A simple guide to five normal forms in relational database theory," *Communications of the ACM*, 26 no. 2 (February 1983), 120-125. [A nice presentation of normalization with examples.]

Rivero, L., J. Doorn, and V. Ferraggine, "Elicitation and Conversion of Hidden Objects and Restrictions in a Database Schema, *Proceedings of the 2002 ACM Symposium on Applied Computing*, 2002, 463-469. [Good discussion of referential integrity issues and problems with weak designs heavily dependent on surrogate ID columns.]

Wu, M.S., "The Practical Need for Fourth Normal Form," *Proceedings of the Twenty-third SIGCSE Technical Symposium on Computer Science Education*, 1992, 19-23. [A small study showing that fourth normal form violations are common in business applications.]

Appendix: Formal Definitions of Normalization

One of the strengths of the relational database model is that it was developed from the mathematical foundations of set theory. Although it is not necessary to know the formal definitions, sometimes they make it easier to understand the process. For a more detailed description of the normal forms and the complications, you should read C. J. Date's advanced textbook. Keep in mind that the formal definitions use specific terms. Figure 3.1A lists the major terms and their common interpretation. Although the formal terms are more accurate, few people have a common understanding of the terms, so in most conversations, it is easier to use the informal terms.

Initial Definitions

A relation is a set of attributes with data that changes over time. Each attribute has a corresponding domain and refers to some real-world characteristic. The formal definitions refer to subsets of attributes, which are collections of the columns. The data value returned within tuples for a specified subset of attributes X is denoted t[X].

The essence of normalization is to recognize that a set of attributes has some real-world relationships. The goal is to accurately portray these relationship constraints. These semantic constraints are known as *functional dependencies (FD)*.

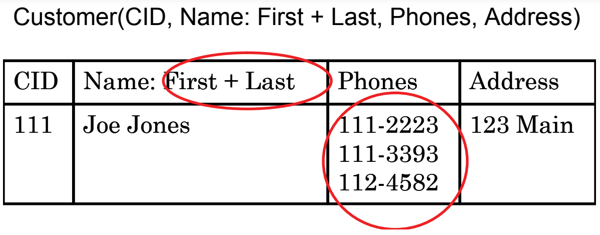
Definition: Functional Dependency and Determinant

Where X and Y are subsets of attributes, a functional dependency is denoted as $X \rightarrow Y$, read as (X implies Y or X determines Y) and holds when any rows of data that have identical values for the X attributes always have identical values for the Y attributes. That is, for tuples t1 and t2 of R, if t1[X] = t2[X], then t1[Y] = t2[Y]. In an FD, X is also known as a determinant, because given the dependency, once you are given the values for the X attributes, it determines the resulting values for the Y attributes.

Figure 3.1A

Terminology. The formal terms are more accurate and defined mathematically, but difficult for developers and users to understand.

Formal	Definition	Informal
Relation	A set of attributes with data that changes over time. Often denoted R.	Table
Attribute	Characteristic with a real-world domain. Subsets of attributes are multiple columns, often denoted X or Y.	Column
Tuple	The data values returned for specific attribute sets are often denoted as t[X].	Row of data
Schema	Collection of tables and constraints and relationships.	
Functional dependency	$X \rightarrow Y$	Business rule dependency



Customer(CID, Name: First + Last, Phones, Address)

CID	Name: First + Last	Phones	Address
111	Joe Jones	111-2223 111-3393 112-4582	123 Main

Figure 3.2A

Nonatomic attributes. This table is not in first normal form because the Name attribute is a composite of two elementary attributes, and the phone attribute is being used to handle multiple values.

Definition: Keys

A key is a set of attributes K such that, where U is the set of all attributes in the relation,

1. There is a functional dependency K → U.
2. If K' is a subset of K, then there is no FD K' → U.

That is, a set of key attributes K functionally determines all other attributes in the relation, and it is the smallest set of attributes that will do so (there is no smaller subset of K that determines the other attributes).

Primary keys are important in relational databases because they are used to identify rows of data. Sometimes multiple attribute sets could be used to form different keys, so they are sometimes referred to as *candidate keys*.

Normal Form Definitions

The definition of first normal form is closely tied to the definition of an atomic attribute, so both need to be defined at the same time.

Definition: First Normal Form (1NF)

A relation is in first normal form if and only if all of its attributes are atomic.

Definition: Atomic Attributes

Atomic attributes are single valued, which means they cannot be composite, multi-valued, or nested relations.

Essentially, a 1NF relation is a table with simple cells under each attribute column. You are not allowed to play tricks and try to squeeze extra data, other relationships, or multiple columns into one column. Figure 3.2A provides an example of a table that is not in first normal form because it has two attributes that are not atomic.

Second normal form is defined in terms of primary keys and functional dependency.

Definition: Second Normal Form (2NF)

A relation is in second normal form if it is in first normal form and each nonkey attribute is fully functionally dependent on the primary key. That is, $K \rightarrow A_i$ for each nonkey attribute A_i. Consequently, there is no subset K' such that $K' \rightarrow A_i$ for any attribute.

OrderProduct(<u>OrderID</u>, <u>ProductID</u>, Quantity, Description)

<u>OrderID</u>	<u>ProductID</u>	Quantity	Description
32	15	1	Blue Hose
32	16	2	Pliers
33	15	1	Blue Hose

Figure 3.3A

Not full dependency. The product description depends on just the ProductID and not the full key {OrderID, ProductID}, so this relation is not in second normal form.

This definition corresponds closely to the simpler version presented in the chapter that each nonkey column depends on the entire key, not just a portion of the key. Figure 3.3A shows an example of a relation that is not in second normal form.

The formal definition of third normal form is a little harder to comprehend because it relies on a new concept: transitive dependency.

<u>Definition: Transitive Dependency</u>

Given functional dependencies X → Y and Y → Z, the transitive dependency X → Z must also hold.

The concept of transitivity should be familiar from basic algebra. The fact that it holds true arises from the set-theory foundations. To understand the definition, remember that functional dependency represents business semantic relationships. Consider the relationship between OrderID, CustomerID, and customer Name attributes. The business rule that there is only one customer per order translates to a functional dependency OrderID → CustomerID. Once you know the OrderID value you always know the CustomerID value. Likewise, the key relationship between CustomerID and other attributes such as Name means there is a functional dependency CustomerID → Name. Applying transitivity, once you know the OrderID value, you can obtain the CustomerID value, and in turn learn the value of the customer Name.

Figure 3.4A

Transitive dependency. The customer Name and Phone attributes transitively depend on the CustomerID, so this relation is not in third normal form.

Order(<u>OrderID</u>, OrderDate, CustomerID, Name, Phone)

<u>OrderID</u>	OrderDate	CustomerID	Name	Phone
32	5/5/2004	1	Jones	222-3333
33	5/5/2004	2	Hong	444-8888
34	5/6/2004	1	Jones	222-3333

Employees can have many specialties, and many employees can be within a specialty. Employees can have many managers, but a manager can have only one specialty: Mgr → Specialty

EmpSpecMgr(EID, Specialty, ManagerID)

EID	Speciality	ManagerID
32	Drill	1
33	Weld	2
34	Drill	1

FD ManagerID → Specialty is not currently a key.

Figure 3.5A

Boyce-Codd normal form. Notice that there is a functional dependency from ManagerID to Specialty. Because this FD is not a candidate key in the relation, it is hidden, and this relation is not in BCNF.

Definition: Third Normal Form (3NF)

A relation is in third normal form if and only if it is in second normal form and no nonkey attributes are transitively dependent on the primary key. That is, given second normal form: $K \rightarrow A_i$ for each attribute A_i, there is no subset of attributes X such that $K \rightarrow X \rightarrow A_i$.

In simpler terms, each non-key attribute depends on the entire key (K), and not on some intermediate attribute (X). Figure 3.4A shows a common business example of a relation that is not in third normal form, because customer attributes depend transitively on the CustomerID.

As discussed in Chapter 3, Boyce-Codd normal form is a little harder to follow. It represents the same basic issue: removing a hidden dependency as seen by the formal definition.

Definition: Boyce-Codd Normal Form (BCNF)

A relation is in Boyce-Codd normal form if and only if it is in third normal form and every determinant is a candidate key. That is, if there is an FD $X \rightarrow Y$, then X must be the primary key (or equivalent to the primary key). In simpler terms: there cannot be a hidden dependency, where hidden means it is not part of the primary key.

As shown in the example in Figure 3.5A, consider the situation where employees can have many specialties, there are many employees for each specialty, and an employee can have many managers, but each manager is manager for only one specialty. This functional dependency (MangerID → Specialty) is not a key within the relation EmpSpecMgr(EID, Specialty, ManagerID), so the relation is not in BCNF. It has to be decomposed to create new relations ManagerSpecialty(ManagerID, Specialty), and EmployeeManager(EmployeeID, ManagerID) that explicitly have each functional dependency as keys.

Fourth normal form is slightly tricky but easy to apply once you understand it. The definition is closely tied to the definition of a multi-valued dependency.

Definition: Multi-Valued Dependency (MVD)

A multi-valued dependency (MVD) exists when there are at least three attributes in a relation (A, B, and C; which could be sets of attributes), and one attribute A determines the other two (B and C), but the two dependencies are independent of each other. That is, A → B and A → C, but B and C are not functionally dependent on each other.

For example, employees can have many specialties and be assigned many tools, but tools and specialties are not directly related to each other.

Definition: Fourth Normal Form (4NF)

A relation is in fourth normal form if and only if it is in Boyce-Codd normal form and there are no multi-valued dependencies. That is, all attributes of the relation are functionally dependent on A.

In the multi-valued dependency example for employee specialties and tools, the relation EmpSpecTools(EID, Specialty, ToolID) is not in fourth normal form, because of the two functional dependencies: EID → Specialty; and EID → ToolID. Solving the problem results in two simpler relations: EmployeeSpecialty(EID, Specialty) and EmployeeTools(EID, ToolID).

Part 2

Queries

An important step in building applications is creating queries to retrieve exactly the data that you want. Queries are used to answer business questions and serve as the foundation for forms and reports.

Chapter 4 shows you how to use two basic query systems: SQL and QBE. SQL has the advantage of being a standard that is supported by many database management systems. Once you learn it, you will be able to work with many different systems.

Chapter 5 shows some of the powerful aspects of SQL queries. In particular, it examines the use of subqueries to answer difficult business questions. It also shows that SQL is a complete database language that can be used to define new databases and tables. SQL is also a powerful tool to manipulate data.

Chapter 4

Data Queries

Chapter Outline

What You Will Learn in This Chapter

- Why do you need a query language?
- What are the main tasks of a query language?
- What business questions can be answered with the basic SQL SELECT command?
- What is the basic structure of a query?
- What tables and columns are used in the Pet Store?
- How do you write queries for a specific DBMS?
- How do you create a basic query?
- What types of computations can be performed in SQL?
- How do you compute subtotals?
- How do you use multiple tables in a query?
- How do you search XML and complex text strings?

A Developer's View

Miranda: Wow that was hard work! I sure hope normalization gets easier the next time.

Ariel: At least now you have a good database. What's next? Are you ready to start building the application?

Miranda: Not quite yet. I told my uncle that I had some sample data. He already started asking me business questions; for example, Which products were backordered most often? and Which employees sold the most items last month? I think I need to know how to answer some of those questions before I try to build an application.

Ariel: Can't you just look through the data and find the answer?

Miranda: Maybe, but that would take forever. Instead, I'll use a query system that will do most of the work for me. I just have to figure out how to phrase the business questions as a correct query.

Getting Started

Building basic SQL queries requires you to address four questions. (1) What output (columns and calculations) do you want to see? (2) What do you know or what constraints are given? (3) What tables are involved? (4) How are the tables joined? A powerful feature of SQL is the ease of computing subtotals with the GROUP BY statement. Learn the SELECT statement and use it as a fill-in-the-blanks model:

```
SELECT
FROM
INNER JOIN
WHERE
GROUP BY
HAVING
ORDER BY
```

Introduction

Why do you need a query language? Why not just ask your question in a natural language like English? Natural language processors have improved, and several companies have attempted to connect them to databases. Similarly, speech recognition is improving. Eventually, computers may be able to answer ad hoc questions using a natural language. However, even if an excellent natural language processor existed, it still would be better to use a specialized query language. The main reason for the problem is communication. If you ask a question of a database, a computer, or even another person, you can get an answer. The catch is, did the computer give you the answer to the question you asked? In other words, you have to know that the machine (or other person) interpreted the question in exactly the way you wanted. The problem with any natural language is that it can be ambiguous. If there is any doubt in the interpretation, you run the risk of receiving an answer that might appear reasonable, but is not the answer to the question you meant to ask.

A query system is more structured than a natural language so there is less room for misinterpretation. Query systems are also becoming more standardized, so that developers and users can learn one language and use it on a variety of different systems. **SQL** is the standard database query language. The standard is established through the ISO (International Organization of Standards) and it is updated every few years. Most database management systems implement most of the SQL 2003 standard. The draft SQL 2006 standard adopted as SQL 2008 provides definitions for several important programming concepts and for XML, but most DBMS vendors have continued to use their existing, proprietary definitions. Consequently, although these standards are accepted by most vendors, there is still room for variations in the SQL syntax, so queries written for one database system will not always work on another system. SQL 2011 added a few new elements to the features added in 2008. Plus it initiated new definitions for handling time elements.

Most database systems also provide a **query by example (QBE)** method or query builder to help beginners create SQL queries. These visually oriented tools generally let users select items from lists, and handle the syntax details to make it easier to create ad hoc queries. Although the QBE designs are easy to use and save time by minimizing typing, you must still learn to use the SQL commands. Many times, you will have to enter SQL into programming code or copy and edit SQL statements.

As you work on queries, you should also think about the overall database design. Chapter 3 shows how normalization is used to split data into tables that can be stored efficiently. Queries are the other side of that problem: They are used to put the tables back together to answer ad hoc questions and produce reports.

The first two sections of this chapter provide an overview of queries. The sections beginning with the Pet Store provide a more detailed explanation of how to build queries, starting with a single table and basic criteria issues. Overall, this chapter covers the basic features of the SELECT statement. The focus is on learning the key SELECT clauses and on translating business questions into SQL queries. Chapter 5 covers more complex business questions that utilize some of the more complex and more powerful features of SQL.

Two-Minute Chapter

Relational databases were initially created to store large amounts of data efficiently. Putting data into separate tables reduces duplication and simplifies adding new rows of data. For example, new sales can be added without interfering with existing sales or even added by multiple people at the same time. But, you might be wondering, how is it possible to retrieve this data? For example, the Customer data was split from the Sales data and stored in separate tables, linked by CustomerID. Sure, a person could retrieve a row from the Sales table, get the CustomerID value and then go look up the matching data in the Customer table, but that seems painful.

The SQL query language was created to answer these questions. The foundations of the SELECT command are covered in this chapter. One strength of SQL is that it is a declarative language instead of procedural. In a typically procedural programming language, you would have to write code to open a table, and loop through all of the rows to find the ones you want. With SQL, you simply tell (declare to) the DBMS what you want to see. You specify (1) the columns you want displayed, (2) the conditions you want to apply, (3) the tables involved, and

(4) how the tables are connected. Retrieving Sales and Customers in March is a simple as:

```
SELECT Customer.CID, Lastname, SaleDate
FROM Customer
INNER JOIN Sale ON Customer.CID = Sale.CID
WHERE (SaleDate BETWEEN '3/1/2013' AND '3/31/2013');
```

SQL can also handle basic calculations such as price * quantity. More importantly, it can compute totals with a simple function: Sum(price*quantity). Totals are computed across all of the specified rows. But SQL also computes subtotals for any level of grouping using the GROUP BY statement. A critical goal of this chapter is to be able to read business questions and write the matching SQL statement—particularly for computing subtotals. For instance, to find the best employee for the month of March, use:

```
SELECT Employee.EID, Lastname, Sum(Price*Quantity) As
TotalSales
FROM Employee
INNER JOIN Sale ON Employees.EID = Sales.EID
INNER JOIN SaleItem ON Sales.SaleID = SaleItems.SaleID
WHERE (SaleDate BETWEEN '3/1/2013' AND '3/31/2013')
GROUP BY Employee.EID, Lastname
ORDER BY Sum(Price*Quantity) DESC;
```

Query editors can be used to drag-and-drop tables and columns to create the JOINs and enter conditions. But you should learn the basic elements of the SELECT command so you can type them by hand when necessary.

Three Tasks of a Query Language

What are the main tasks of a query language? To create databases and build applications, you need to perform three basic sets of tasks: (1) define the database, (2) change the data, and (3) retrieve data. Some systems use formal terms to describe these categories. Commands grouped as **data definition language (DDL)** are used to define the data tables and other features of the database. The common DDL commands include: ALTER, CREATE, and DROP. Commands used to modify the data are classified as **data manipulation language (DML)**. Common DML commands are: DELETE, INSERT, and UPDATE. Some systems include data retrieval within the DML group, but the SELECT command is complex enough to require its own discussion. The appendix to this chapter lists the syntax of the various SQL commands. Virtually all tasks can be performed by issuing a DDL, DML, or query command. This chapter focuses on the SELECT command. The DML and DDL commands will be covered in more detail in Chapter 5.

The SELECT command is used to retrieve data: It is the most complex SQL command, with several different options. The main objective of the SELECT command is to retrieve specified columns of data for rows that meet some criteria. Database management systems are driven by query systems. Many query systems support a graphical interface which makes it easier to create queries by reducing typing and through visualizing the relationships among tables. But, ultimately you should learn the text versions of the SQL commands.

What output do you want to see?	SELECT columns
What tables are invovled?	FROM table
How are the tables joined?	INNER JOIN table
What do you already know (or what constraints are given)?	WHERE conditions

Figure 4.1

Four questions to create a query. Every query is built by asking these four questions. The SELECT… FROM … INNER JOIN … WHERE … syntax is the SQL form to creating a query.

SQL SELECT Overview

What business questions can be answered with the basic SQL SELECT command? For the most part, SQL is a declarative language, which is unlike traditional programming procedural languages. You simply have to tell the DBMS what you want and it determines how to get that data. You do not have to write loops or conditional statements. The SELECT command is the primary method of retrieving data from tables. This chapter focuses on its basic elements: (1) choosing columns and making basic calculations, (2) selecting rows of data based on given information, (3) joining related tables, (4) sorting the results, and (5) computing subtotals. Many business questions rely on these basic tools.

At the simplest level, business questions just need to retrieve data that matches some basic conditions. Questions such as: List customers who made a purchase in a specified month or bought a specific product; or Find employees who sold items to a specific customer. Building a query to answer these basic questions just involves identifying the tables that hold the desired data, selecting the desired columns to display, and entering the specified filter conditions, and perhaps sorting the results. It is critical that you learn to build these simple queries correctly.

As a small step up, the SELECT statement can also perform simple computations. Business problems often need to multiply values in two columns, such as Price*Quantity, or to subtract two values. The SELECT statement handles basic arithmetic as well as common mathematical and string functions. These calculations operate on data held in one row and follow standard rules for mathematical operations.

The most important capability of SQL in basic business questions is to compute subtotals. A surprising number of business questions involve subtotals. For instance: Which customer spent the most money last month? Which employee sold the most items last week? Which product category is the best seller? What are total sales by month? SQL easily handles subtotals with the GROUP BY clause. Simply list the column that contains the items to break (group) on, then include an aggregate function (usually Sum, Avg, Count) as a computation. The DBMS will find each unique value in the GROUP BY column (such as each employee), then compute the subtotal indicated in the Sum function for each item. Sorting the results makes it easy to find the highest or lowest value. To convert a business question into SQL, you often identify the items to be summed (or counted) and then determine which columns hold the grouping values. Examine the business question and look for words such as “by” or “for each.”

Four Questions to Retrieve Data

What is the basic structure of a query? Every attempt to retrieve data from a relational DBMS requires answering the four basic questions listed in Figure 4.1. The difference among query systems is how you fill in those answers. The figure also shows the matching SQL clauses for answering the questions.
Notice that in some easy situations you will not have to answer all four questions. Many easy questions involve only one table, so you will not have to worry about joining tables. As another example, you might want the total sales for the entire company, as opposed to the total sales for a particular employee, so there may not be any constraints.

The SELECT statements can be used as a fill-in-the-blanks type of form. Start the query by writing down or typing those key words on the left side of the page. Then fill in the items to the right of the keywords. It is often easiest to begin by writing down the output that you want to see on the SELECT statement, followed by the constraints on the WHERE clause. Then you can identify all of the tables needed and use the relationship diagram to see how the tables are joined.

What Output Do You Want to See?

As an initial step, you can think of a query as a way to filter data—both in terms of columns you want to see and limiting the rows based on various conditions. You could just retrieve every column from a table, but it gets hard to wade through columns that are not important to the business question. So you need to tell the DBMS which columns you want to see. More importantly, you first have to visualize your output before you can write the rest of the query. In general, a query system answers your query by displaying rows of data from various columns. You

Figure 4.2

Simple example of a column filter. Use the SELECT clause to choose only the columns or calculations you want to see.

EmployeeID	LastName	Phone
1	Reeves	402-146-7714
2	Gibson	919-245-0526
3	Reasoner	413-414-8275
4	Hopkins	412-524-0814
5	James	407-026-6653
6	Eaton	906-446-7957
7	Farris	615-891-5545
8	Carpenter	212-545-8897
9	O’Conner	203-180-0146
10	Shields	304-607-9081
11	Smith	80-333-9872

```
SELECT EmployeeID, LastName,Phone
FROM Employee
```

EmployeeID	LastName	Phone	EmploeeLevel
4	Hopins	412-524-9814	3
5	James	407-026-6653	3
7	Farris	615-891-5545	3

```
SELECT EmployeeID, LastName, phone, EmployeeLevel
FROM Employee
WHERE EmployeeLevel=3;
```

Figure 4.3

Simple example of a row filter. The WHERE clause limits the rows to be displayed based on multiple conditions. Connecting conditions with an "OR" statement returns more rows in the results because each row could meet on many conditions.

can also ask the DBMS to perform some basic computations, so you also need to identify any calculations and totals you need.

You generally answer this question by selecting columns of data from the various tables stored in the database. Of course, you need to know the names of all of the columns to answer this question. Generally, the hardest part in answering this question is to wade through the list of tables and identify the columns you really want to see. The problem is more difficult when the database has hundreds of tables and thousands of columns. Queries are easier to build if you have a copy of the class diagram that lists the tables, their columns, and the relationships that join the tables.

Figure 4.2 shows a simple example of a SELECT statement using the Employee table in the Pet Store database. The SELECT clause specifies the columns or calculations you want to see. You can think of it as a column filter—choosing a subset of the columns available in the tables. If you want to see all of the columns, you can simply use SELECT * FROM Employee to show all the columns.

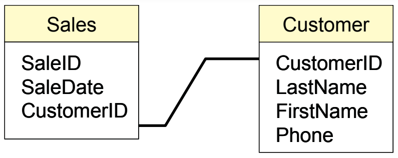
What Do You Already Know?

In most situations you want to restrict your search based on various criteria. For instance, you might be interested in sales on a particular date or sales from only one department. The search conditions must be converted into a standard Boolean notation (phrases connected with AND or OR). The most important part of this step is to write down all the conditions to help you understand the purpose of the query.

Figure 4.3 shows that you can think of the WHERE clause as a filtering statement. Rows are displayed in the results only if the data within the row matches the conditions in the WHERE clause. If multiple conditions are connected with an "OR" term, the query generally returns more rows because the data can match any one of the conditions. Connecting the conditions with an "AND" term reduces the number of rows because each row must match all of the conditions.

What Tables Are Involved?

With only a few tables, this question is easy. With hundreds of tables, it could take a while to determine exactly which ones you need. A good data dictionary with synonyms and comments will make it easier for you (and users) to determine



SaleID	SaleDate	CustomerID
1	5/1	1
2	5/1	2
3	5/2	4
4	5/2	1

CustomerID	LastName	FirstName	Phone
1	Jones	Mary	111-2222
2	Smith	Marta	222-3333
3	Jackson	Miguel	444-2222
4	Smith	Mark	555-5662

SaleID	SaleDate	CustomerID	CustomerID	LastName	FirstName	Phone
1	5/1	1	1	Jones	Mary	111-2222
2	5/1	2	2	Smith	Marta	222-3333
3	5/2	4	4	Smith	Mark	555-5662
4	5/2	1	1	Jones	Mary	111-2222

Figure 4.4

Joining tables. A join causes a lookup to match rows across the tables.

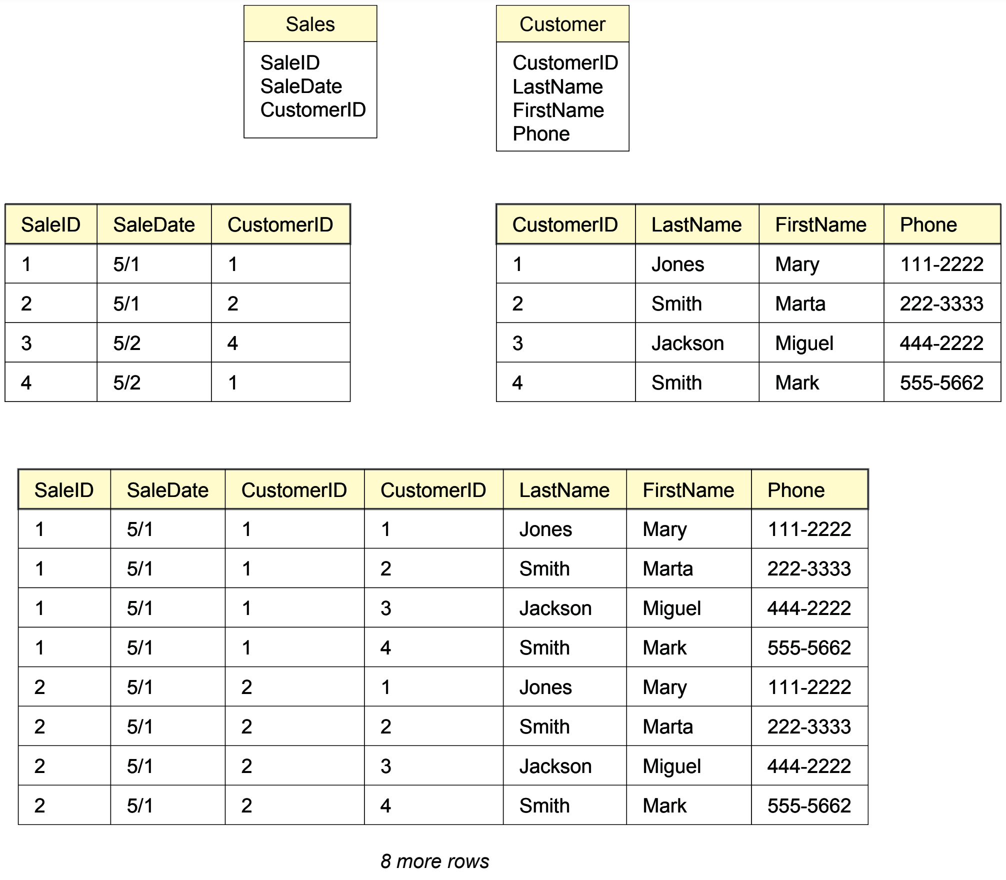
exactly which tables you need for the query. It is also critical that tables be given names that accurately reflect their content and purpose.

One hint in choosing tables is to start with the tables containing the columns listed in the first two questions (output and criteria). Next decide whether other tables might be needed to serve as intermediaries to connect these tables.

How Are the Tables Joined?

This question relates to the issues in data normalization and is the heart of a relational database. Tables are connected by data in similar columns. For instance, as shown in Figure 4.4, a Sales table has a CustomerID column. Corresponding data is stored in the Customer table, which also has a CustomerID column. In many cases matching columns in the tables will have the same name (e.g., CustomerID) and this question is easy to answer. The join performs a matching or lookup for the rows. You can think of the result as one giant table and use any of the columns from any of the joined tables. Note that columns are not required to have the same name, so you sometimes have to think a little more carefully. For example, an Order table might have a column for SalesPerson, which is designed to match the EmployeeID key in an Employee table.

Joining tables is usually straightforward as long as your database design is sound. In fact, most QBE systems will automatically use the design to join any tables you add. However, two problems can arise in practice: (1) You should verify that all tables are joined, and (2) Double-check any tables with multiple join conditions.


Sales
SaleID
SaleDate
CustomerID
Customer
CustomerID
LastName
FirstName
Phone

SaleID	SaleDate	CustomerID
1	5/1	1
2	5/1	2
3	5/2	4
4	5/2	1

CustomerID	LastName	FirstName	Phone
1	Jones	Mary	111-2222
2	Smith	Marta	222-3333
3	Jackson	Miguel	444-2222
4	Smith	Mark	555-5662

SaleID	SaleDate	CustomerID	CustomerID	LastName	FirstName	Phone
1	5/1	1	1	Jones	Mary	111-2222
1	5/1	1	2	Smith	Marta	222-3333
1	5/1	1	3	Jackson	Miguel	444-2222
1	5/1	1	4	Smith	Mark	555-5662
2	5/1	2	1	Jones	Mary	111-2222
2	5/1	2	2	Smith	Marta	222-3333
2	5/1	2	3	Jackson	Miguel	444-2222
2	5/1	2	4	Smith	Mark	555-5662

8 more rows

Figure 4.5

Cross join. With no join condition, the DBMS performs a cross join and matches every row in the first table to every row in the second table, often leading to millions of rows in the result. Be sure to specify a join condition and stay away from cross joins.

Technically, it is legal to use tables without adding a join condition. However, when no join condition is explicitly specified, the DBMS creates a **cross join** or Cartesian product between the tables. A cross join matches every row in the first table to every other row in the second table. For example, if both tables have 10 rows, the resulting cross join yields 10*10 = 100 rows of data. If the tables each have 1,000 rows, the resulting join has one million rows! A cross join will seriously degrade performance on any DBMS, so be sure to specify a join condition for every table. The one exception is that it is sometimes used to join a single-row result with every row in a second table. With only one row in a table, the cross join is reasonably fast. Figure 4.5 shows the results of a cross join using two small tables.

Sometimes table designs have multiple relationship connections between tables. For example, the Pet Store database joins Customer to City and City to Employee. A query system that automatically adds relationship joins will bring along every connection. But, you rarely want to use all of the joins at the same time. The key is to remember that a join represents a restrictive condition. In the Pet Store case, if you include the two joins from the Customer, City, and Employee tables, you would be saying that you only want to see customers who live in the same city as an employee.

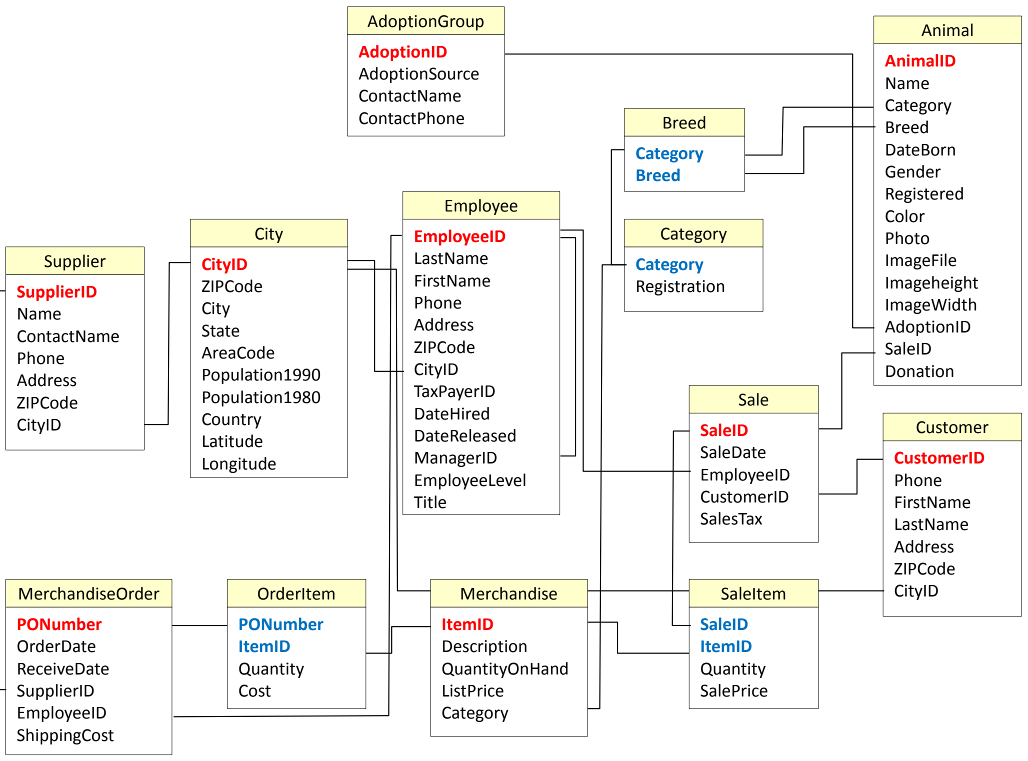
Sally's Pet Store

What tables and columns are used in the Pet Store? The initial Pet Store database has been built, and some basic historical data has been transferred from Sally's old files. When you show your work to Sally, she becomes very excited. She immediately starts asking questions about her business, and wants to see how the database can answer them.

The examples in this chapter are derived from the Pet Store database. The tables and relationships for this case are shown in Figure 4.6. After reading each section, you should work through the queries on your own. You should also solve the exercises at the end of the chapter. Queries always look easy when the answers are printed in the book. To learn to write queries, you must sit down and struggle through the process of answering the four basic questions.

Figure 4.6

Tables for the Pet Store database. Notice that animals and merchandise are similar, but they are treated separately.



Chapter 3 notes that data normalization results in a business model of the organization. The list of tables gives a picture of how the firm operates. Notice that the Pet Store treats merchandise differently than it treats animals. For example, each animal is listed separately on a sale, but customers can purchase multiple copies of merchandise items (e.g., bags of cat food). The reason for the split is that each animal is unique and can be adopted only once. Also, you need to keep additional information about the animals that does not apply to general merchandise.

When you begin to work with an existing database, the first thing you need to do is familiarize yourself with the tables and columns. You should look through some of the main tables to become familiar with the type and amount of data stored in each table. Make sure you understand the terminology and examine the underlying assumptions. For example, in the Pet Store case, an animal might be registered with a breeding agency, but it can be registered with only one agency. If it is not registered, the Registered column is **NULL** (or missing) for that animal. This first step is easier when you work for a specific company, since you should already be familiar with the firm's operations and the terms that it uses for various objects.

Vendor Differences

How do you write queries for a specific DBMS? The SQL standards present a classic example of software development trade-offs. New releases of the standards provide useful features, but vendors face the need to maintain compatibility with a large installed base of applications and users. Consequently, substantial differences exist across database products. These differences are even more pronounced when you look at the graphical interfaces.

Whenever possible, you should use the newer standards because the queries are easier to read. However, it is likely that you will encounter queries written in the older syntax, so you should also learn how to read these older versions. The one catch is that each DBMS vendor had its own proprietary syntax. It is impossible to cover all of the variations in this book. The details of the syntax and the basic steps for writing and testing queries within a DBMS are explained in the accompanying workbooks. Each workbook explores the same issues using a single DBMS. At a minimum, you should read through and work the examples in one workbook. If you have time, it is instructive to compare the techniques of several vendors.

Query Basics

How do you create a basic query? The basic goal is to convert a business question into a database query. It is best to begin with relatively easy queries. This chapter first presents queries that involve a single table to show the basics of creating a query. Then it covers details on constraints, followed by a discussion on computations and aggregations. Groups and subtotals are then explained. Finally, the chapter discusses how to select data from several tables at the same time.

Figure 4.7 presents several business questions that might arise at the Pet Store. Most of the questions are relatively easy to answer. In fact, if there are not too many rows in the Animal table, you could probably find the answers by hand-searching the table. Actually, you might want to work some of the initial questions by hand to help you understand what the query system is doing.

- Which animals were born after August 1?
- List the animals by category and breed.
- List the categories of animals that are in the Animal list.
- Which dogs have a donation value greater than $250?
- Which cats have black in their color?
- List cats excluding those that are registered or have red in their color.
- List all dogs who are male and registered or who were born before 01-June-2013 and have white in their color.
- What is the extended value (price * quantity) for sale items on sale 24?
- What is the average donation value for animals?
- What is the total value of order number 22?
- How many animals were adopted in each category?
- How many animals were adopted in each category with total adoptions of more than 10?
- How many animals born after June 1 were adopted in each category with total adoptions more than 10?
- List the CustomerID of everyone who bought or adopted something between April 1, 2013 and May 31, 2013.
- List the names of everyone who bought or adopted something between April 1, 2013 and May 31, 2013.
- List the name and phone number of anyone who adopted a registered white cat between two given dates.

Figure 4.7

Sample questions for the Pet Store. Most of these are easier since they involve only one table. They represent typical questions that a manager or customer might ask.

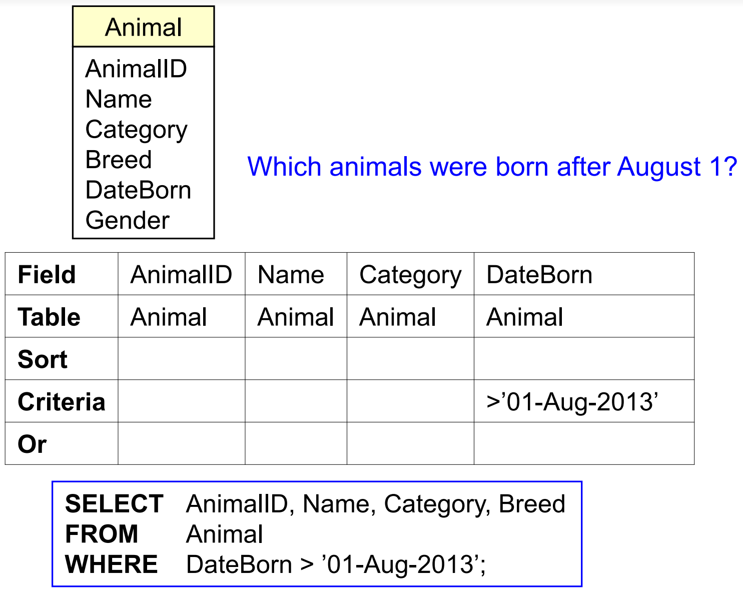
The foundation of queries is that you want to see only some of the columns from a table and that you want to restrict the output to a set of rows that match some criteria. For example, in the first query (animals with yellow color), you might want to see the AnimalID, Category, Breed, and their Color. Instead of listing every animal in the table, you want to restrict the list to just those with a yellow color.

Single Tables

The first query to consider is: *Which animals were born after August 1?* Figure 4.8 shows a QBE approach and the SQL. The two methods utilize the same underlying structure. The QBE approach saves some typing, but eventually you need to be able to write the SQL statements. If you write down the SQL keywords, you can fill in the blanks—similar to the way you fill in the QBE grid.

First consider answering this question with a QBE system. The QBE system will ask you to choose the tables involved. This question involves only one table: Animal. You know that because all of the data you want to see and the constraint are based on columns in the Animal table. With the table displayed, you can now choose which columns you want to see in the output. The business question is a little vague, so select AnimalID, Name, Category, and DateBorn.

The next step is to enter the criteria that you already know. In this example, you are looking for animals born after a specific date. On the QBE grid, enter the condition >'01-Aug-2013' on the Criteria row under the DateBorn column. There is one catch: Different DBMSs use different syntax for the way you enter the date. Most of them will accept the date format and the single quotes shown here. For



Field	AnimalID	Name	Category	DateBorn
Table	Animal	Animal	Animal	Animal
Sort				
Criteria				>’01-Aug-2013’
Or				

```
SELECT   AnimalID, Name, Category, Breed
FROM     Animal
WHERE    DateBorn > ’01-Aug-2013’;
```

Figure 4.8

Sample query shown in QBE and SQL. Since there is only one table, only three questions need to be answered: What tables? What conditions? What do you want to see?

Microsoft Access, do not include any quotation marks. The Access QBE interface will automatically add # marks instead. It is a good idea to run the query now. Check the DateBorn result to ensure that the query was entered correctly.

The four basic questions are answered by filling out blanks on the QBE grid. (1) The output to be displayed is placed as a field on the grid. (2) The constraints are entered as criteria or conditions under the appropriate fields. (3) The tables involved are displayed at the top (and often under each field name). (4) The table joins are shown as connecting lines among the tables. The one drawback to QBE systems is that you have to answer the most difficult question first: Identifying the tables involved. The QBE system uses the table list to provide a list of the columns you can choose. Keep in mind that you can always add more tables as you work on the problem.

Introduction to SQL

SQL is a powerful query language. However, unlike QBE, you generally have to type in the entire statement. Most systems enable you to switch back and forth between QBE and SQL, which saves some typing. Perhaps the greatest strength of SQL is that it is a standard that most vendors of DBMS software support. Hence, once you learn the base language, you will be able to create queries on all of the major systems in use today. Note that some people pronounce SQL as “sequel,” arguing that it descended from a vendor’s early DBMS called quel. Also, “Sequel” is easier to say than “ess-cue-el.” But with the introduction of CQL for Cassandra (see Chapter 13) it will be safer to just say SQL.

The most commonly used command in SQL is the SELECT statement, which is used to retrieve data from tables. A simple version of the command is shown

SELECT	columns	What do you want to see?
FROM	tables	What tables are involved?
JOIN	conditions	How are the tables joined?
WHERE	criteria	What are the constraints?

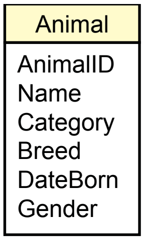
Figure 4.9

The basic SQL SELECT command matches the four questions you need to create a query. The uppercase letters are used in this text to highlight the SQL keywords. They can also be typed in lowercase.

in Figure 4.9, which contains the four basic parts: **SELECT**, **FROM**, **JOIN**, and **WHERE**. These parts match the basic questions needed by every query. In the example in Figure 4.8, notice the similarity between the QBE and SQL approaches. The four basic questions are answered by entering items after each of the four main keywords. When you write SQL statements, it is best to write down the keywords and then fill in the blanks. You can start by listing the columns you want to see as output, then write the constraints in the WHERE clause. By looking at the columns you used, it is straightforward to identify the tables involved. You can use the class diagram to understand how the tables are joined.

Figure 4.10

The ORDER BY clause sorts the output rows. The default is to sort in ascending order, adding the keyword DESC after a column name results in a descending sort. When columns like Category contain duplicate data, use a second column (e.g., Breed) to sort the rows within each category.



Field	Name	Category	Breed
Table	Animal	Animal	Animal
Sort		Ascending	Ascending
Criteria			
Or			

Name	Category	Breed
Cathy	Bird	African Grey
	Bird	Canary
Debbie	Bird	Cockatiel
	Bird	Cockatiel
Terry	Bird	Lovebird
	Bird	Other
Charles	Bird	Parakeet
Curtis	Bird	Parakeet
Ruby	Bird	Parakeet
Sandy	Bird	Parrot
Hoyt	Bird	Parrot
	Bird	Parrot

```
SELECT Name, Category, Breed
FROM Animal
ORDER BY Category, Breed;
```

```
SELECT Category
 FROM Animal;
```

Category
Fish
Dog
Fish
Cat
Cat
Dog
Fish
Dog
Dog
Dog
Fish
Cat
Dog
. . .

```
SELECT DISTINCT Category
 FROM Animal;
```

Category
Bird
Cat
Dog
Fish
Mammal
Reptile
Spider

Figure 4.11

The DISTINCT keyword eliminates duplicate rows of the output. Without it the animal category is listed for every animal in the database.

Sorting the Output

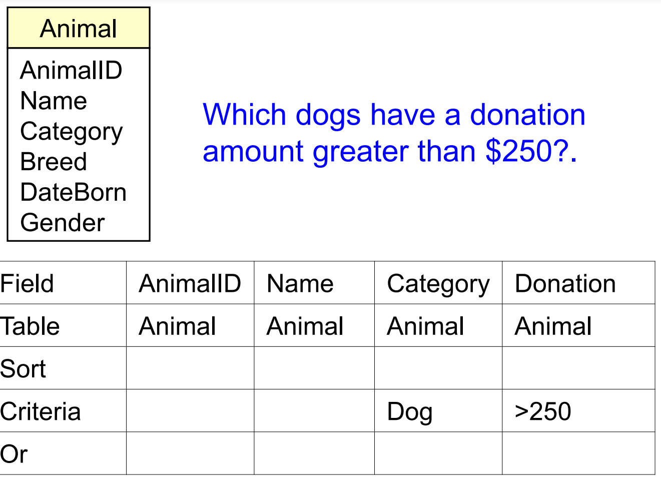
Database systems treat tables as collections of data. For efficiency the DBMS is free to store the table data in any manner or any order that it chooses. Yet in most cases you will want to display the results of a query in a particular order. The SQL **ORDER BY** clause is an easy and fast means to display the output in any order you choose. As shown in Figure 4.10, simply list the columns you want to sort. The default is ascending (A to Z or low to high with numbers). Add the phrase **DESC** (for descending) after a column to sort from high to low. In QBE you select the sort order on the QBE grid.

In some cases you will want to sort columns that do not contain unique data. For example, the rows in Figure 4.10 are sorted by Category. In these situations you would want to add a second sort column. In the example, rows for each category (e.g., Bird) are sorted on the Breed column. The column listed first is sorted first. In the example, all birds are listed first, and birds are then sorted by Breed. To change this sort sequence in QBE, you have to move the entire column on the QBE grid so that Category is to the left of Breed.

Distinct

The SELECT statement has an option that is useful in some queries. The **DISTINCT** keyword tells the DBMS to display only rows that are unique. For example, the query in Figure 4.11 (*SELECT Category FROM Animal*) would return a long list of animal types (Bird, Cat, Dog, etc.). In fact, it would return the category for every animal in the table—obviously; there are many cats and dogs. To prevent the duplicates from being displayed, use the SELECT DISTINCT phrase.

Note that the DISTINCT keyword applies to the entire row. If there are any differences in a row, it will be displayed. For example, the query *SELECT DISTINCT Category, Breed FROM Animal* will return more than the seven rows shown in



Field	AnimalID	Name	Category	Donation
Table	Animal	Animal	Animal	Animal
Sort				
Criteria			Dog	>250
Or				

```
SELECT   AnimalID, Category, DateBorn
FROM     Animal
WHERE    Category=N'Dog' AND Donation>250;
```

Figure 4.12

Boolean algebra. An example of two conditions connected by AND. QBE uses an AND connector for all conditions listed on the same row. Note the use of the N'…' notation in SQL Server and Oracle to specify Unicode (National) data for strings. In Access, simply use double-quote marks without the N.

Figure 4.11 because each category can have many breeds. That is, each category/breed combination will be listed only once, such as Dog/Retriever. Microsoft Access supports the DISTINCT keyword, but you have to enter it in the SQL statement.

Criteria

In most questions, identifying the output columns and the tables is straightforward. If there are hundreds of tables, it might take a while to decide exactly which tables and columns you want, but it is just an issue of perseverance. On the other hand, identifying constraints and specifying them correctly can be more challenging. More importantly if you make a mistake on a constraint, you will still get a result. The problem is that it will not be the answer to the question you asked—and it is often difficult to see that you made a mistake.

The primary concept of constraints is based on **Boolean algebra**, which you learned in mathematics. In practice, the term simply means that various conditions are connected with AND and OR clauses. Sometimes you will also use a **NOT** statement, which negates or reverses the truth of the statement that follows it. For example, NOT (Category = N'Dog') means you are interested in all animals except dogs.

Consider the example in Figure 4.12. The first step is to note that two conditions define the business question: dog and donation. The second step is to recognize that both of these conditions need to be true at the same time, so they are connected by AND. As the database system examines each row, it evaluates both clauses. If any one clause is false, the row is skipped. Notice the use of N'Dog' statement which converts the entered text to Unicode (national) format. Because the data table formats were defined as Unicode, it is best to enter this specification whenever you write a query. If the text in the query processor uses English, the conversion is usually automatic and you could skip the N prefix. However, it is always safer to specify the conversion explicitly.

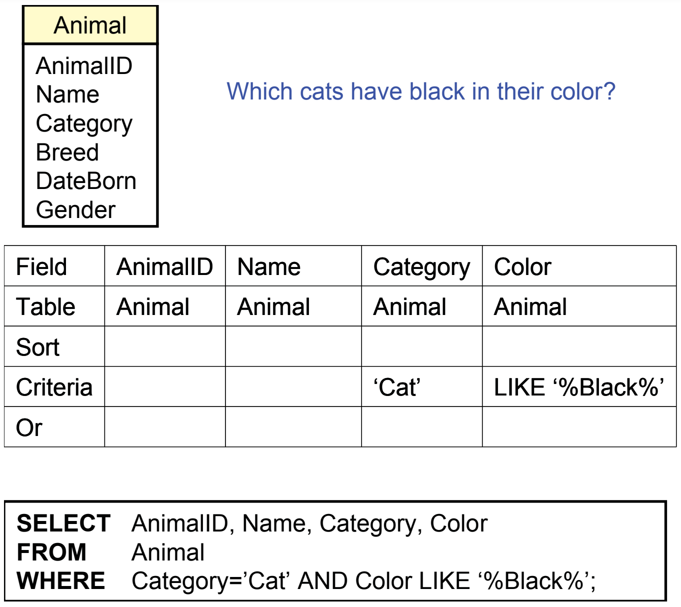
Notice that the SQL statement is straightforward—just write the two conditions and connect them with an AND clause. The QBE is a little trickier. With QBE, every condition listed on the same criteria row is connected with an AND clause. Conditions on different criteria rows are joined with an OR clause. You have to be careful creating (and reading) QBE statements, particularly when there are many different criteria rows.

Pattern Matching

Databases are designed to handle many different types of data, including numbers, dates, and text. The standard comparison operators (<, >, =, and so on) work well for numbers, dates, and simple text values. However, larger text fields often require more powerful comparisons. The SQL standard provides the **LIKE** command to handle simple pattern matching tasks. The LIKE command uses two special characters to create a pattern that is compared to each selected row of text. In standard SQL, the percent sign (%) in a pattern matches any character or characters (including none). The underscore (_) matches exactly one character. Before exploring patterns, note that Microsoft Access uses an asterisk (*) and question mark (?) instead. Access does provide the option to use the standard percent sign and underscore characters, but almost no one activates that option.

You construct a pattern by using the percent or underscore characters. Generally, you want to search for a specific word or phrase. Consider the request from a customer who wants a black cat. If you look at the Color column of the Animal table, you will see that can contain multiple colors for any animal. If you think about animals for a minute, it is clear that an animal can have multiple colors. Technically, this choice means that the Color column is probably not atomic; and you could have specified a completely new table that lists each color on a separate line. However, color definitions are somewhat subjective, and it is more complicated to enter data and write queries when multiple tables are involved. Consequently, the database is a little more usable by listing the colors in a single column. But, now you have to search it. If you search using the equals sign (say, WHERE Color=N'Black'), you will see only animals that are completely black. Perhaps the customer is willing to settle for a cat that has a few white spots, which might have been entered as Black/White; and will not show up in the simple equality search.

The answer is to construct a pattern search that will list a cat that has the word Black anywhere in the Color column. Figure 4.13 shows the syntax of the query. The key is the phrase: Color LIKE N'%Black%'. Placing a percent sign at the start of the pattern means that any characters can appear before the word Black. Placing a percent sign at the end of the pattern means that any characters can appear after the word Black. Consequently, if the word Black appears anywhere in the color list, the LIKE condition will be true. Note that the simple color "Black" will also be matched because the percent sign matches no characters. If you leave off the first percent sign (Color LIKE N'Black%'), the condition would be true only if the Color column begins with the word Black (followed by anything else).



Which cats have black in their color?

Field	AnimalID	Name	Category	Color
Table	Animal	Animal	Animal	Animal
Sort				
Criteria			'Cat'	LIKE '%Black%'
Or				

```
SELECT  AnimalID, Name, Category, Color
FROM    Animal
WHERE   Category='Cat' AND Color LIKE '%Black%';
```

Figure 4.13

Pattern matching. The percent sign matches any characters, so if the word Black appears anywhere in the Color column the LIKE condition is true.

You can construct more complex conditions using pattern matching, but you should test these patterns carefully. For instance, you could search a Comment column for two words using: Comment LIKE N'%friendly%children%'. This pattern will match any row that has a comment containing both of the words (friendly and children). There can be other words in front of, behind, or between the two words, but they must appear in the order listed.

You can also use the single character matching tool (underscore) to create a pattern. This tool is useful in certain situations. It is most useful when you have a text column that is created with a particular format. For instance, most automobile license plates follow a given pattern (such as AAA-999). If a policeman gets a partial license plate number, he could search for matches. For instance, License LIKE N'XQ_-12_', would search for plates where the third character and third number are not known. Keep in mind that the single-character pattern will only match a character that exists. In the example, if a license number has three letters but only two numbers, the pattern will never match it because the pattern requires a third number. In business, the single-character pattern is useful for searching product codes that contain a fixed format. For instance, a department store might identify products by a three-character department code, a two-character color code, a two-digit size code, and a five-digit item code: DDDCC11-12345. If you wanted to find all blue (BL) items of size 9 (09), you could use: ItemCode LIKE N'_ _ _BL09-_ _ _ _ _'. Note that spaces were added in the text to show the number of underscores, but you would need to enter the underscores into the query without any intervening spaces.

a	b	a AND b	a OR b
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Figure 4.14

A truth table shows the difference between AND and OR. Both clauses must be true when connected by AND. Only one clause needs to be true when clauses are connected by OR.

Boolean Algebra

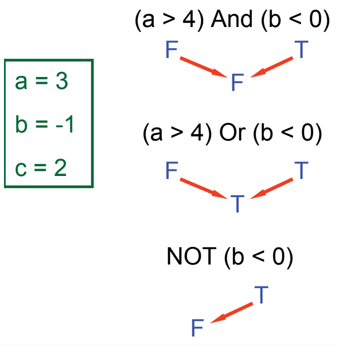
One of the most important aspects of a query is the choice of rows that you want to see. Most tables contain a huge number of rows, and you want to see only the few that meet a business condition. Some conditions are straightforward. For example, you might want to examine only dogs. Other criteria are complex and involve several conditions. For instance, a customer might want a list of all yellow dogs born after June 1, 2013, or registered black labs. Conditions are evaluated according to Boolean algebra, which is a standard set of rules for evaluating conditions. You are probably already familiar with the rules from basic algebra courses; however, it pays to be careful.

The DBMS uses Boolean algebra to evaluate conditions that consist of multiple clauses. The clauses are connected by these operators: AND, OR, NOT. Each individual clause is evaluated as true or false, and then the operators are applied to evaluate the truth value of the overall criterion. Figure 4.14 shows how the primary operators (AND, OR) work. The DBMS examines each row of data and evaluates the Boolean condition. The row is displayed only if the condition is true.

A condition consisting of two clauses connected by AND can be true only if both of the clauses (a And b) are true. A statement that consists of two clauses connected by OR is true as long as at least one of the two conditions is true. Consider the examples shown in Figure 4.15. The first condition is false because it

Figure 4.15

Boolean algebra examples. Evaluate each clause separately. Then evaluate the connector. The NOT operator reverses the truth value.



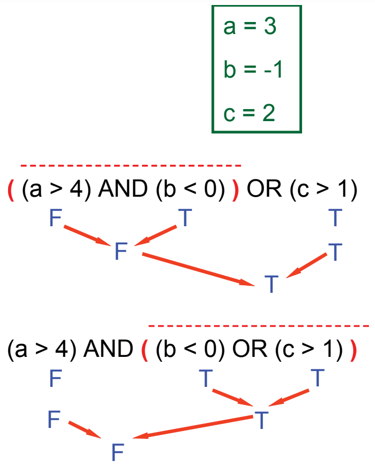

a = 3
b = -1
c = 2
((a > 4) AND (b < 0)) OR (c > 1)
F T T
F T
T
(a > 4) AND ((b < 0) OR (c > 1))
F T T
F T
F

Figure 4.16

Boolean algebra mixing AND and OR operators. The result changes depending on which operator is applied first. You must set the order of evaluation with parentheses. Innermost clauses are evaluated first.

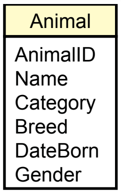
asks for both clauses to be true, and the first one is false (a < 4). The second example is true because it requires only that one of the two clauses be true. Consider an example from the Pet Store. If a customer asks to see a list of yellow dogs, he or she wants a list of animals where the category is Dog AND the color is yellow.

As shown in Figure 4.16, conditions that are more complex can be created by adding additional clauses. A complication arises when the overall condition contains both AND connectors and OR connectors. In this situation the resulting truth value depends on the order in which the clauses are evaluated. You should always use parentheses to specify the desired order. Innermost parentheses are evaluated first. In the example at the top of Figure 4.16, the AND operation is performed before the OR operation, giving a result of true. In the bottom example, the OR connector is evaluated first, leading to an evaluation of false.

If you do not use parentheses, the operators are evaluated from left to right. This result may not be what you intended, yet the DBMS will still provide a response. To be safe, you should build complex conditions one clause at a time. Check the resulting selection each time to be sure you get what you wanted. To find the data matching the conditions in Figure 4.16, you would first enter the (a > 4) clause and display all of the values. Then you would add the (b < 0) clause and display the results. Finally, you would add the parentheses and then the (c > 1) clause.

No matter how careful you are with Boolean algebra there is always room for error. The problem is that natural languages such as English are ambiguous. For example, consider the request by a customer who wants to see a list of "All dogs that are yellow or white and born after June 1." This staement can be intrepreted two ways:

1. (dogs AND yellow) OR (white AND born after June 1).
2. (dogs) AND (yellow OR white) AND (born after June 1).



Customer: "I want to look at a cat, but I don't want any cats that are registered or that have red in their color."

Field	AnimalID	Category	Registered	Color
Table	Animal	Animal	Animal	Animal
Sort				
Criteria		'Cat'	Is Null	Not Like '%Red%'
Or				

```
SELECT AnimalID, Category, Registered, Color
FROM Animal
WHERE (Category='Cat') AND
        NOT ((Registered is NOT NULL)
                    OR (Color LIKE '%Red%')).
```

Figure 4.17

Sample problem with negation. Customer knows what he or she does not want. SQL can use NOT, but you should use DeMorgan's law to negate the Registered and Color statements.

These two requests are significantly different. The first interpretation returns all yellow dogs, even if they are older. The second interpretation requests only young dogs, and they must be yellow or white. Most people do not use parentheses when they speak—although pauses help indicate the desired interpretation. A good designer (or salesperson) will ask the customer for clarification.

DeMorgan's Law

Designing queries is an exercise in logic. A useful technique for simplifying complex queries was created by a logician named Augustus DeMorgan. Consider the Pet Store example displayed in Figure 4.17. A customer might come in and say, "I want to look at a cat, but I don't want any cats that are registered or that have red in their color." Even in SQL, the condition for this query is a little confusing: (Category = N'Cat') AND NOT ((Registered is NOT NULL) OR (Color LIKE N'%Red%')). The negation (NOT) operator makes it harder to understand the condition. It is even more difficult to create the QBE version of the statement.

The solution lies with **DeMorgan's law**, which explains how to negate conditions when two clauses are connected with an AND or an OR. DeMorgan's law states that to negate a condition with an AND or an OR connector, you negate each of the two clauses and switch the connector. An AND becomes an OR, and vice versa. Figure 4.17 shows how to handle the negative condition for the Pet Store customer. Each condition is negated (NOT NULL becomes NULL, and red becomes NOT red). Then the connector is changed from OR to AND. Figure 4.18 shows that the final truth value stays the same when the statement is evaluated both ways.

The advantage of the new version of the condition is that it is a little easier to understand and much easier to use in QBE. In QBE you enter the individual clauses for Registration and Color. Placing them on the same line connects them

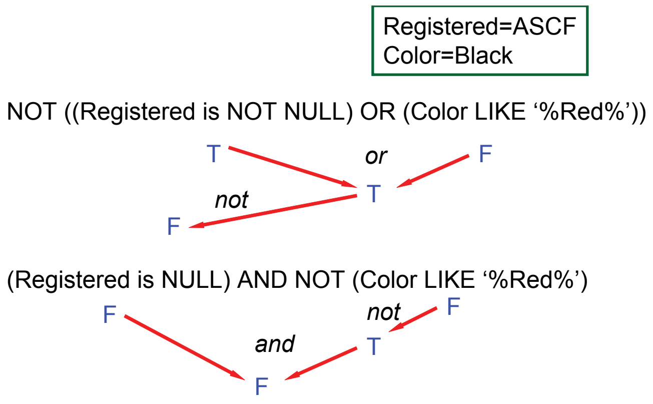
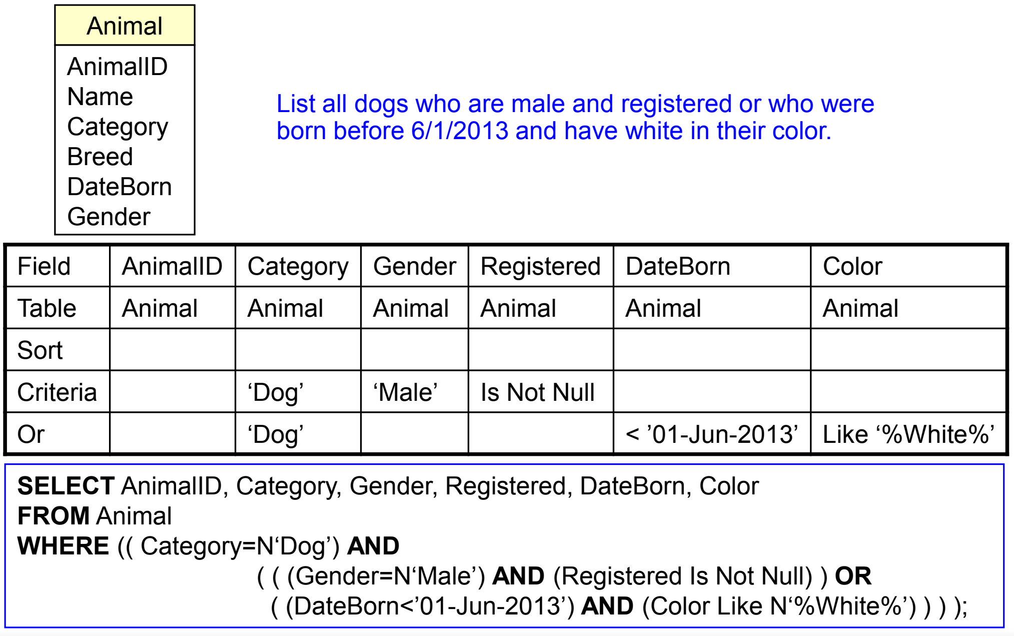

Registered=ASCF
Color=Black
NOT ((Registered is NOT NULL) OR (Color LIKE '%Red%'))
T
or
F
T
not
F
(Registered is NULL) AND NOT (Color LIKE '%Red%')
F
not
F
and
T
F

Figure 4.18

DeMorgan's law. Compound statements are negated by reversing each item and swapping the connector (AND for OR). Use truth tables to evaluate the examples.

Figure 4.19

Boolean criteria—mixing AND and OR. Notice the use of parentheses in SQL to ensure the clauses are interpreted in the right order. Also note that QBE required duplicating the condition for "Dog" in both rows.


Animal
AnimalID
Name
Category
Breed
DateBorn
Gender
List all dogs who are male and registered or who were born before 6/1/2013 and have white in their color.

Field	AnimalID	Category	Gender	Registered	DateBorn	Color
Table	Animal	Animal	Animal	Animal	Animal	Animal
Sort						
Criteria		'Dog'	'Male'	Is Not Null		
Or		'Dog'			< '01-Jun-2013'	Like '%White%'

```
SELECT AnimalID, Category, Gender, Registered, DateBorn, Color
FROM Animal
WHERE (( Category=N'Dog') AND
                           ( ( (Gender=N'Male') AND (Registered Is Not Null) ) OR
                            ( (DateBorn<'01-Jun-2013') AND (Color Like N'%White%') ) ) );
```

with AND. In natural language the new version is expressed as follows: A cat that is not registered and is not red. In practice DeMorgan's law is useful to simplify complex statements. However, you should always test your work by using sample data to evaluate the truth tables.

Criteria can become more complex when you mix clauses with AND and OR in the same query. Consider the question in Figure 4.19 to list all dogs who are male and registered or who were born before June 1 and have white in their color.

First, note that there is some ambiguity in the English statement about how to group the two clauses. Figure 4.20 shows the two possibilities. The use of the second who helps to clarify the split, but the only way to be absolutely certain is to use either parentheses or more words.

The SQL version of the query is straightforward—just be sure to use parentheses to indicate the priority for evaluating each phrase. Innermost clauses are always evaluated first. A useful trick in proofreading queries is to use a sample row and mark T or F above each condition. Next, combine the marks based on the parentheses and connectors (AND, OR). Then read the statement in English and see whether you arrive at the same result.

With QBE you list clauses joined by AND on the same row, which is equivalent to putting them inside one set of parentheses. Separate clauses connected by OR are placed on a new row. To interpret the query, look at each criteria row separately. If all of the conditions on one line are true, then the row is determined to be a match. A data row needs to match only one of the separate criteria lines (not all of them).

A second hint for building complex queries is to test just part of the criteria at one time—particularly with QBE. In this example, you would first write and test a query for male and registered. Then add the other conditions and check the results at each step. Although this process takes longer than just leaping to the final query, it helps to ensure that you get the correct answer. For complex queries it is always wise to examine the SQL WHERE clause to make sure the parentheses are correct.

Useful WHERE Clauses

Most database systems provide the comparison operators displayed in Figure 4.21. Standard numeric data can be compared with equality and inequality operators. Text comparisons are usually made with the LIKE operator for pattern matching. For all text criteria, you need to know if the system uses case-sensitive comparisons. By default, Microsoft Access and SQL Server are not case-sensitive, so you can type the pattern or condition using any case. On the other hand, Oracle

Figure 4.20

Ambiguity in natural languages means the sentence could be interpreted either way. However, version (1) is the most common interpretation.

List all dogs who are male and registered or who were born before 6/1/2007 and have white in their color.

1: (male and registered) or (born before June 1 and white)
2: (male) and (registered or born before June 1) and (white)

Comparisons	Examples
Operators	<, =, >, <>, >=, BETWEEN, LIKE, IN
Numbers	AccountBalance > 200
Text Simple Pattern match one Pattern match any	 Name > 'Jones' License LIKE 'A_ _82_' Name LIKE 'J%'
Dates	SaleDate BETWEEN '15-Aug-2013' AND '31-Aug-2013'
Missing Data	City IS NULL
Negation	Name IS NOT NULL
Sets	Category IN ('Cat', 'Dog', 'Hamster')

Figure 4.21

Common comparisons used in the WHERE clause. The BETWEEN clause is useful for dates but can be used for any type of data.

is case-sensitive by default so you have to be careful to type the case correctly. If you do not know which case was used, you can use the UPPER function to convert to upper case and then write the pattern using capital letters.

The **BETWEEN** clause is a useful way to handle common date conditions. The clause (SaleDate BETWEEN '15-Aug-2013' AND '31-Aug-2013' is equivalent to (SaleDate >= '15-Aug-2013' AND SaleDate <= '31-Aug-2013'). The date syntax shown here can be used on most database systems. Some systems allow you to use shorter formats, but on others, you will have to specify a conversion format. These conversion functions are not standard. For example, Access can read almost any common date format if you surround the date by pound signs (#) instead of quotes. Oracle often requires the TO_DATE conversion function, such as SaleDate >= TO_DATE('8/15/13', 'mm/dd/yy'). Be sure that you test all date conversions carefully, especially when you first start working with a new DBMS.

Another useful condition is to test for missing data with the NULL comparison. Two common forms are IS NULL and IS NOT NULL. Be careful—the statement (City = NULL) will not work with most systems, because NULL is not really a value. You must use (City IS NULL) instead. Unfortunately, conditions with the equality sign are not flagged as errors. The query will run—it just will never match anything.

Computations

What types of computations can be performed in SQL? For the most part you would use a spreadsheet or write separate programs for serious computations. However, queries can be used for two types of computations: aggregations and simple arithmetic on a row-by-row basis. Sometimes the two types of calculations are combined. Consider the row-by-row computations first.

Basic Arithmetic Operators

SQL and QBE can both be used to perform basic computations on each row of data. This technique can be used to automate basic tasks and to reduce the amount

SaleItem(<u>SaleID</u>, <u>ItemID</u>, SalePrice, Quantity)

```
Select SaleID, ItemID, SalePrice, Quantity,
SalePrice*Quantity As Extended
From SaleItem;
```

SaleID	ItemID	Price	Quantity	Extended
24	25	2.70	3	8.10
24	26	5.40	2	10.80
24	27	31.50	1	31.50

Figure 4.22

Computations. Basic computations (+ - * /) can be performed on numeric data in a query. The new display column should be given a meaningful name.

of data storage. Consider a common order or sales form. As Figure 4.22 shows, the basic tables would include a list of items purchased: SaleItem(SaleID, ItemID, SalePrice, Quantity). In most situations you would need to multiply SalePrice by Quantity to get the total value for each item ordered. Because this computation is well defined (without any unusual conditions), there is no point in storing the result—it can be recomputed whenever it is needed. Simply build a query and add one more column. The new column uses elementary algebra and lists a name: SalePrice*Quantity AS Extended. Remember that the computations are performed for each row in the query.

Most systems provide additional mathematical functions. For example, basic mathematical functions such as absolute value, logarithms, and trigonometric functions are usually available. Although these functions provide extended capabilities, always remember that they can operate only on data stored in one row of a table or query at a time.

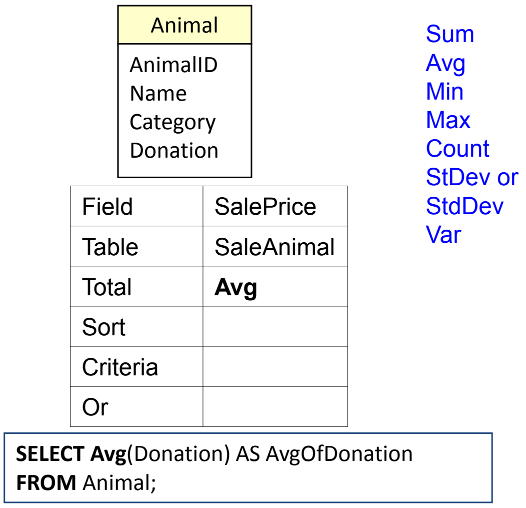
Aggregation

Databases for business often require the computation of totals and subtotals. Hence, query systems provide functions for **aggregation** of data. The common functions listed in Figure 4.23 can operate across several rows of data and return one value. The most commonly used functions are Sum and Avg, which are similar to those available in spreadsheets.

With SQL, the functions are simply added as part of the SELECT statement. With QBE, the functions are generally listed on a separate Total line. With Microsoft Access, you first have to click the summation (∑) button on the toolbar to add the Total line to the QBE grid. In both SQL and QBE, you should provide a meaningful name for the new column.

The Count function is useful in many situations, but make sure you understand the difference between Sum and Count. Sum totals the values in a numeric column. Count simply counts the number of rows. If you supply a column name to the Count function, you should use a primary key column or an asterisk (*).

The difficulty with the Count function lies in knowing when to use it. You must first understand the English question. For example, the question *How many employees does the Pet Store have?* would use the Count function: SELECT Count(*) From Employee. The question *How many units of Item 9764 have been sold?* requires the Sum function: SELECT Sum(Quantity) FROM OrderItem. The



Field	SalePrice
Table	SaleAnimal
Total	**Avg**
Sort	
Criteria	
Or	

```
SELECT Avg(Donation) AS AvgOfDonation
FROM Animal;
```

Figure 4.23

Aggregation functions. Sample query in QBE and SQL to answer: What is the average sale price for all animals? Note that with Microsoft Access you have to click the summation button on the toolbar (∑) to display the Total line on the QBE grid.

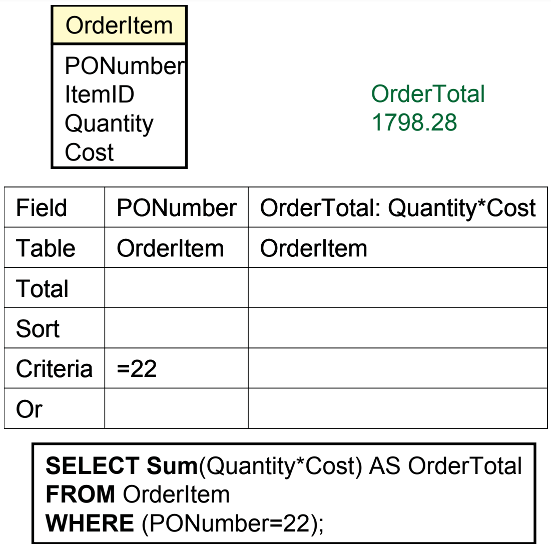
difference is that there can be only one employee per row in the Employee table, whereas a customer can buy multiple quantities of an item at one time. Also keep in mind that Sum can be used only on a column of numeric data (e.g., Quantity).

In many cases you will want to combine the **row-by-row calculations** with an aggregate function. The example in Figure 4.24 asks for the total value of a particular order. To get total value, the database must first calculate Quantity * Cost for each row and then get the total of that column. The example also shows that it is common to specify a condition (WHERE) to limit the rows used for the total. In this example, you want the total for just one order.

There is one important restriction to remember with aggregation. You cannot display detail lines (row by row) at the same time you display totals. In the order example you can see either the detail computations (Figure 4.22) or the total value (Figure 4.24). In most cases it is simple enough to run two queries. However, if you want to see the detail and the totals at the same time, you need to create a report. Some of the most recent SQL standard extensions include provisions for displaying totals and details, but it is almost always easier to create a report.

Note that you can compute several aggregate functions at the same time. For example, you can display the Sum, Average, and Count at the same time: SELECT Sum(Quantity), Avg(Quantity), Count(Quantity) From OrderItem. In fact, if you need all three values, you should compute them at one time. Consider what happens if you have a table with a million rows of data. If you write three separate queries, the DBMS has to make three passes through the data. By combining the computations in one query, you cut the total query time to one-third. With huge tables or complex systems, these minor changes in a query can make the difference between a successful application and one that takes days to run.

Sometimes when using the Count function, you will also want to include the DISTINCT operator. For example, *SELECT COUNT (DISTINCT Category)*



```
OrderTotal
1798.28
```

Field	PONumber	OrderTotal: Quantity*Cost
Table	OrderItem	OrderItem
Total		
Sort		
Criteria	=22	
Or		

```
SELECT Sum(Quantity*Cost) AS OrderTotal
FROM OrderItem
WHERE (PONumber=22);
```

Figure 4.24

Computations. Row-by-row computations (Quantity*Cost) can be performed within an aggregation function (Sum), but only the final total will be displayed in the result.

FROM Animal will count the number of different categories and ignore duplicates. Although the command is part of the SQL standard, some systems (notably Access) do not support the use of the DISTINCT clause within the Count statement. To obtain the same results in Access, you would first build the query with the DISTINCT keyword. Save the query and then create a new query that computes the Count on the saved query.

Functions

The SELECT command also supports functions that perform calculations on the data. These calculations include numeric forms such as the trigonometric functions, string function such as concatenating two strings, date arithmetic functions, and formatting functions to control the display of the data. Unfortunately, these functions are not standardized, so each DBMS vendor has different function names and different capabilities. Nonetheless, you should learn how to perform certain standard tasks in whichever DBMS you are using. Figure 4.25 lists some of the common functions you might need. Even if you are learning only one DBMS right now, you should keep this table handy in case you need to convert a query from one system to another.

String operations are relatively useful. Concatenation is one of the more powerful functions, because it enables you to combine data from multiple columns into a single display field. It is particularly useful when you want to combine a person's last and first names. Other common string functions convert the data to all lowercase or all uppercase characters. The length function counts the number of characters in the string column. A substring function is used to return a selected portion of a string. For example, you might choose to display only the first 20 characters of a long title.

Task	Access	SQL Server	Oracle
Strings			
Concatenation	FName & “ “ & LName	FName + ‘ ‘ + LName	Fname || ‘ ‘ || LName
Length	Len(LName)	Length(LName)	LENGTH(LName)
Upper case	UCase(LName)	Upper(LName)	UPPER(LName)
Lower case	LCause(LName)	Lower(LName)	LOWER(LName)
Partial string	MID(LName,2,3)	Substring(LName,2,3)	SUBSTR(LName,2,3)
Dates			
Today	Date( ), Time( ), Now( )	GetDate( )	SYSDATE
Month	Month(myDate)	DateName(month, myDate)	TRUNC(myDate, ‘mm’)
Day	Day(myDate)	DatePart(day, myDate)	TRUNC(myDate, ‘dd’)
Year	Year(myDate)	DatePart(year, myDate)	TRUNC(myDate, ‘yyyy’)
Date arithmetic	DateAdd	DateAdd	ADD_MONTHS
	DateDiff	DateDif	MONTHS_BETWEEN
			LAST_DAY
Formatting	Format(item, format)	Str(item, length, decimal)	TO_CHAR(item, format)
		Cast, Convert	TO_DATE(item, format)
Numbers			
Math functions	Cos, Sin, Tan, Sqrt	Cos, Sin, Tan, Sqrt	COS, SIN, TAN, SQRT
Exponentiation	2 ^ 3	Power(2, 3)	POWER(2, 3)
Aggregation	Min, Max, Sum, Count,	Min, Max, Sum, Count,	MIN, MAX, SUM, COUNT,
Statistics	Avg, StDev, Var	Avg, StDev, Var,	REGR, CORR
		LinRegSlope, Correlation	

Figure 4.25

Differences in SQL functions. This table shows some of the differences that are commonly encountered when working with these database systems. Queries are often used to perform basic computations, but the syntax for handling these computations depends on the specific DBMS.

The powerful date functions are often used in business applications. Date columns can be subtracted to obtain the number of days between two dates. Additional functions exist to get the current date and time or to extract the month, day, or year parts of a date column. Date arithmetic functions can be used to add (or subtract) months, weeks, or years to a date. One issue you have to be careful with is entering date values into a query. Most systems are sensitive to the fact that world regions have different standards for entering and displaying dates. For example, 5/1/2013 is the first day in May in the United States, but it is the fifth day in January in Europe. To make sure that the DBMS understands exactly how you want a date interpreted, you might have to use a conversion function and specify the date format. Additional formatting functions can be used for other types of data, such as setting a fixed number of decimal points or displaying a currency sign.

A DBMS might have dozens of numeric functions, but you will rarely use more than a handful. Most systems have the common trigonometric functions (e.g., sine and cosine), as well as the ability to raise a number to a power. Most also provide some limited statistical calculations such as the average and standard deviation, and occasionally correlation or regression computations. You will have to consult the DBMS documentation for availability and details on additional functions. However, keep in mind that you can always write your own functions and use them in queries just as easily as the built-in functions.

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

Field	Category	AnimalID
Table	Animal	Animal
Total	**Group By**	Count
Sort		Descending
Criteria		
Or		

Category	CountOfAnimalID
Dog	100
Cat	47
Bird	15
Fish	14
Reptile	6
Mammal	6
Spider	3

```
SELECT      Category, Count(AnimalID) AS CountOfAnimalID
FROM        Animal
GROUP BY    Category
ORDER BY    Count(AnimalID) DESC;
```

Figure 4.26

GROUP BY computes subtotals and counts for each type of animal. This approach is much more efficient than trying to create a WHERE clause for each type of animal. To convert business questions to SQL, watch for phrases such as by or for each which usually signify the use of the GROUP BY clause.

Subtotals and GROUP BY

How do you compute subtotals? To look at totals for only a few categories, you can use the Sum function with a WHERE clause. For example you might ask *How many cats are in the animal list?* The query is straightforward: SELECT Count (AnimalID) FROM Animal Where (Category = N'Cat'). This technique will work, and you will get the correct answer. You could then go back and edit the query to get the count for dogs or any other category of animal. However, eventually you will get tired of changing the query. Also, what if you do not know all the categories?

Consider the more general query: Count the number of animals in each category. As shown in Figure 4.26, this type of query is best solved with the GROUP BY clause. This technique is available in both QBE and SQL. The SQL syntax is straightforward: just add the clause GROUP BY Category. The **GROUP BY** statement can be used only with one of the aggregate functions (Sum, Avg, Count, and so on). With the GROUP BY statement, the DBMS looks at all the data, finds the unique items in the group, and then performs the aggregate function for each item in the group.

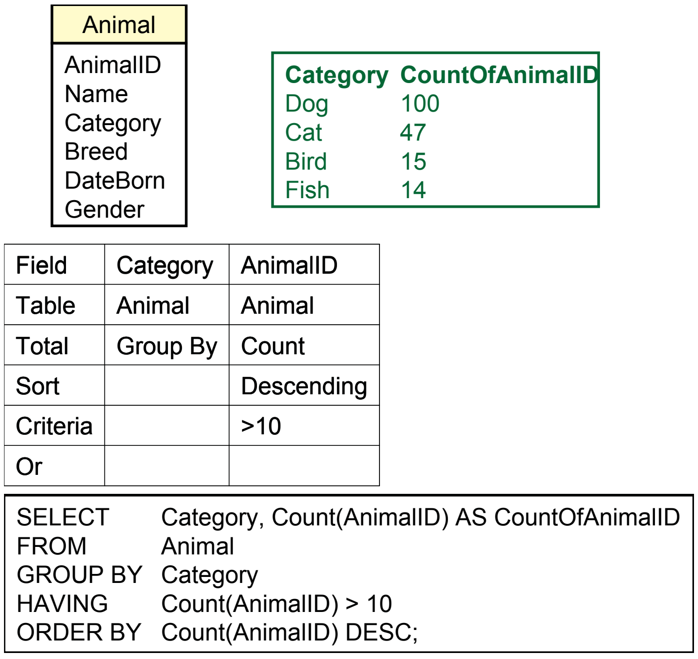
By default, the output will generally be sorted by the group items. However, for business questions, it is common to sort (ORDER BY) based on the computation. The Pet Store example is sorted by the Count, listing the animals with the highest count first. Be careful about adding multiple columns to the GROUP BY clause. The subtotals will be computed for each distinct item in the entire GROUP BY clause. If you include additional columns (e.g., Category and Breed), you might end up with a more detailed breakdown than you wanted.

Microsoft added a useful feature that can be used in conjunction with the ORDER BY statement. Sometimes a query will return thousands of lines of output. Although the rows are sorted, you might want to examine only the first few rows. For example, you might want to list your 10 best salespeople or the top 10 percent of your customers. When you have sorted the results, you can easily limit the output displayed by including the **TOP** statement; for example, SELECT TOP 10 SalesPerson, SUM(Sales) FROM Sales GROUP BY SalesPerson ORDER BY SUM(Sales) DESC. This query will compute total sales for each salesperson and display a list sorted in descending order. However, only the first 10 rows of the output will be displayed. Of course, you could choose any value instead of 10. You can also enter a percentage value (e.g., TOP 5 PERCENT), which will cut the list off after 5 percent of the rows have been displayed. These commands are useful when a manager wants to see the "best" of something and skip the rest of the rows. Oracle does not support the TOP condition, but you can use the internal row numbers to accomplish the same task. The command syntax relies on subqueries covered in the next chapter, but you might want to reduce your output rows, so an example is given here:

```
SELECT * FROM (SELECT … FROM …) WHERE ROWNUM <= 10;
```

Figure 4.27

Limiting the output with a HAVING clause. The GROUP BY clause with the Count function provides a count of the number of animals in each category. The HAVING clause restricts the output to only those categories having more than 10 animals.



Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

Category	CountOfAnimalID
Dog	100
Cat	47
Bird	15
Fish	14

Field	Category	AnimalID
Table	Animal	Animal
Total	Group By	Count
Sort		Descending
Criteria		>10
Or		

```
SELECT      Category, Count(AnimalID) AS CountOfAnimalID
FROM        Animal
GROUP BY    Category
HAVING      Count(AnimalID) > 10
ORDER BY    Count(AnimalID) DESC;
```

The 2011 SQL standard expanded a clause to do the same thing but the syntax is slightly different and it might take a while to be fully supported. The new clause is the FETCH statement added to the end of the SELECT clause. For example, the salesperson query would be written as:

```
SELECT SalesPerson, SUM(Sales) FROM Sales GROUP BY
SalesPerson ORDER BY SUM(Sales) DESC
FETCH FIRST 10 ROWS WITH TIES
```

Conditions on Totals (HAVING)

The GROUP BY clause is powerful and provides useful information for making decisions. In cases involving many groups, you might want to restrict the output list, particularly when some of the groups are relatively minor. The Pet Store has categories for reptiles and spiders, but they are usually special-order items. In analyzing sales the managers might prefer to focus on the top-selling categories.

One way to reduce the amount of data displayed is to add the **HAVING** clause. The HAVING clause is a condition that applies to the GROUP BY output. In the example presented in Figure 4.27, the managers want to skip any animal category that has fewer than 10 animals. Notice that the SQL statement simply adds one line. The same condition can be added to the criteria grid in the QBE query. The HAVING clause is powerful and works much like a WHERE statement. Just be sure that the conditions you impose apply to the computations indicated by the GROUP BY clause. The HAVING clause is a possible substitute in Oracle which lacks the TOP statement. You can sort a set of subtotals and cut off the list to display only values above a certain limit.

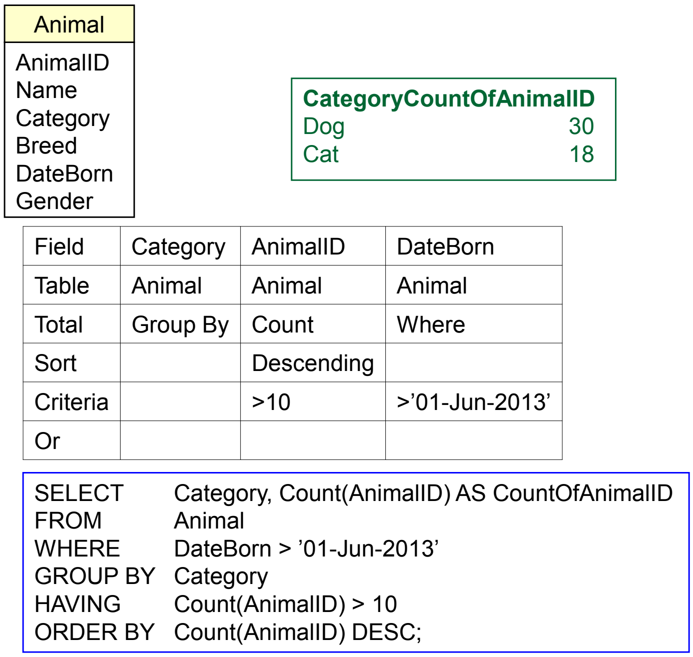
WHERE versus HAVING

When you first learn QBE and SQL, WHERE and HAVING look very similar, and choosing the proper clause can be confusing. Yet it is crucial that you understand the difference. If you make a mistake, the DBMS will give you an answer, but it will not be the answer to the question you want.

The key is that the WHERE statement applies to every single detail row in the original table. The HAVING statement applies only to the subtotal output from a GROUP BY query. To add to the confusion, you can even combine WHERE and HAVING clauses in a single query—because you might want to look at only some rows of data and then limit the display on the subtotals.

Consider the question in Figure 4.28 that counts the animals born after June 1, 2013, in each Category, but lists only the Category if there are more than 10 of these animals. The structure of the query is similar to the example in Figure 4.25. The difference in the SQL statement is the addition of the WHERE clause (DateBorn > #6/1/2013#). This clause is applied to every row of the original data to decide whether it should be included in the computation. Compare the count for dogs in Figure 4.26 (30) with the count in Figure 4.25 (100). Only 30 dogs were born after June 1, 2013. The HAVING clause then limits the display to only those categories with more than 10 animals.

The query is processed by first examining each row to decide whether it meets the WHERE condition. If so, the Category is examined and the Count is increased for that category. After processing each row in the table, the totals are examined to see whether they meet the HAVING condition. Only the acceptable rows are displayed. The same query in QBE is a bit more confusing. Both of the conditions are listed in the criteria grid. However, look closely at the Total row, and you will



Category	CountOfAnimalID
Dog	30
Cat	18

Field	Category	AnimalID	DateBorn
Table	Animal	Animal	Animal
Total	Group By	Count	Where
Sort		Descending	
Criteria		>10	>’01-Jun-2013’
Or			

```
SELECT     Category, Count(AnimalID) AS CountOfAnimalID
FROM       Animal
WHERE      DateBorn > ’01-Jun-2013’
GROUP BY   Category
HAVING     Count(AnimalID) > 10
ORDER BY   Count(AnimalID) DESC;
```

Figure 4.28

WHERE versus HAVING. Count the animals born after June 1, 2007, in each category, but list the category only if it has more than 10 of these animals. The WHERE clause first determines whether each row will be used in the computation. The GROUP BY clause produces the total count for each category. The HAVING clause restricts the output to only those categories with more than 10 animals.

see a Where entry for the DateBorn column. This entry is required to differentiate between a HAVING and a WHERE condition. To be safe, you should always look at the SQL statement to make sure your query was interpreted correctly.

The Best and the Worst

Think about the business question, *Which product is our best seller?* How would you build a SQL statement to answer that question? To begin, you have to decide if “best” is measured in quantity or revenue (price times quantity). For now, simply use quantity. A common temptation is to write a query similar to: SELECT Max(Quantity) FROM SaleItem. This query will run. It will return the individual sale that had the highest sale quantity, but it will not sum the quantities. A step closer might be: SELECT ItemID, Max(Sum(Quantity)) FROM SaleItem GROUP BY ItemID. But this query will not run because the database cannot compute the maximum until after it has computed the sum. So, the best answer is to use: *SELECT ItemID, Sum(Quantity) FROM SaleItem GROUP BY ItemID ORDER BY Sum(Quantity) DESC*. This query will compute the total quantities purchased for each item and display the result in descending order—the best-sellers will be at the top of the list.

Note that this query displays more than the simple “best” answer. It displays all of the totals. The advantage to this approach is that it shows other rows that might be close to the "best" entry, which is information that might be valuable to the decision maker. The one drawback to this approach is that it returns the complete list of items sold. Generally, most businesspeople will want to see more than just the top or bottom item, so it is not a serious drawback—unless the list is too long. In that case, you can use the TOP or HAVING command to reduce the length of the list.

Multiple Tables

How do you use multiple tables in a query? All the examples so far have used a single table—to keep the discussion centered on the specific topics. In practice, however, you often need to combine data from several tables. In fact, the strength of a DBMS is its ability to combine data from multiple tables.

Chapter 3 shows how business forms and reports are dissected into related tables. Although the normalization process makes data storage more efficient and avoids common problems, ultimately, to answer the business question, you need to recombine the data from the tables. For example, the Sale table contains just the CustomerID to identify the specific customer. Most people would prefer to see the customer name and other attributes. This additional data is stored in the Customer table—along with the CustomerID. The objective is to take the CustomerID from the Sale table and look up the matching data in the Customer table.

Figure 4.29

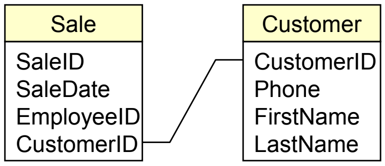
List the CustomerID of everyone who bought something between April 1, 2013 and May 31, 2013. Most people would prefer to see the name and address of the customer—those attributes are in the Customer table.

Sale
SaleID
SaleDate
EmployeeID
CustomerID
SalesTax

Field	CustomerID	SaleDate
Table	Sale	Sale
Sort	Ascending	
Criteria		Between '01-Apr-2013' And '31-May-2013'
Or		

```
SELECT DISTINCT CustomerID
FROM Sale
WHERE (SaleDate Between '01-Apr-2013'
            And '31-May-2013')
ORDER BY CustomerID;
```

```
CustomerID
6
8
14
19
22
24
28
36
37
38
39
42
50
57
58
63
74
80
90
```



Field	CustomerID	LastName	SaleDate
Table	Sale	Customer	Sale
Sort		Ascending	
Criteria			Between '01-Apr-2013' And '31-May-2013'
Or			

```
SELECT DISTINCT Sale.CustomerID, Customer.LastName
FROM Customer
INNER JOIN Sale ON Customer.CustomerID = Sale.CustomerID
WHERE (SaleDate Between '01-Apr-2013' And '31-May-2013')
ORDER BY Customer.LastName;
```

CustomerID	LastName
22	Adkins
57	Carter
38	Franklin
42	Froedge
63	Grimes
74	Hinton
36	Holland
6	Hopkins
50	Lee
58	McCain
…	

Figure 4.30

Joining tables causes the rows to be matched based on the columns in the JOIN statement. You can then use data from either table. The business question is, List the last name of customers who bought something between April 1, 2013, and May 31, 2013.

Joining Tables

With modern query languages, combining data from multiple tables is straightforward. You simply specify which tables are involved and how the tables are connected. QBE is particularly easy to use for this process. To understand the process, first consider the business question posed in Figure 4.29: list the CustomerID of everyone who bought something between 4/1/2013 and 5/31/2013. Because some customers might have made purchases on several days, the DISTINCT clause can be used to delete the duplicate listings.

Most managers would prefer to see the customer name instead of CustomerID. However, the name is stored in the Customer table because it would be a waste of space to copy all of the attributes to every table that referred to the customer. If you had these tables only as printed reports, you would have to take the CustomerID from the sale reports and find the matching row in the Customer table to get the customer name. Of course, it would be time-consuming to do the matching by hand. The query system can do it easily.

As illustrated in Figure 4.30, the QBE approach is somewhat easier than the SQL syntax. However, the concept is the same. First, identify the two tables involved (Sale and Customer). In QBE, you select the tables from a list, and they are displayed at the top of the form. In SQL, you enter the table names on the FROM line. Second, you tell the DBMS which columns are matched in each table. In this case you match CustomerID in the Sale table to the CustomerID in the Customer table. Most of the time the column names will be the same, but they could be different.

```
SQL 92/Current Syntax
FROM Table1
INNER JOIN Table2
  ON Table1.Column = Table2.Column

SQL 89/Old Sytnax
FROM Table1, Table2
WHERE Table1.Column = Table2.Column

Informal Syntax for Notes
FROM Table1, Table2
JOIN Column
```

Figure 4.31

SQL 92 and SQL 89 syntax to join tables. The informal syntax cannot be used with a DBMS, but it is easier to read when you need to combine many tables.

In SQL tables are connected with the JOIN statement. This statement was changed with the introduction of SQL 92—however, you will encounter many older queries that still use the older SQL 89 syntax. With SQL 89 the JOIN condition is part of the WHERE clause. Most vendors have converted to the SQL 92 syntax, so this text will rely on that format. As Chapter 5 shows, the SQL 92 syntax is much easier to understand when you need to change the join configuration.

The syntax for a JOIN is displayed in Figure 4.31. An informal syntax similar to SQL 89 is also shown. The DBMS will not accept statements using the informal syntax, but when the query uses many tables, it is easier to jot down the informal syntax first and then add the details needed for the proper syntax. Note that with both QBE and SQL, you must specify the tables involved and which columns contain matching data.

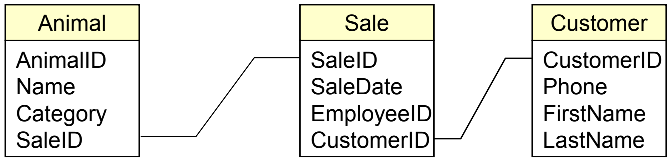
Identifying Columns in Different Tables

Examine how the columns are specified in the SQL JOIN statement. Because the column CustomerID is used in both tables, it would not make sense to write CustomerID = CustomerID. The DBMS would not know what you meant. To keep track of which column you want, you must also specify the name of the table: Sale.CustomerID. Actually, you can use this syntax anytime you refer to a column. You are required to use the full table.column name only when the same column name is used in more than one table.

Joining Many Tables

A query can use data from several different tables. The process is similar regardless of the number of tables. Each table you want to add must be joined to one other table through a data column. If you cannot find a common column, either the normalization is wrong or you need to find a third table that contains links to both tables.

Consider the example in Figure 4.32: List the name and phone number of anyone who adopted a registered white cat between two given dates. An important step is to identify the tables needed. For large problems involving several tables, it is best to first list the columns you want to see as output and the ones involved in the constraints. In the example, the name and phone number you want to see are



Field	LastName	Phone	Category	Registered	Color	SaleDate
Table	Customer	Customer	Animal	Animal	Animal	Sale
Sort	Ascending					
Criteria			'Cat'	Is Not Null	Like '%White%'	Between '01-Jun-2013' And '31-Dec-2013'
Or						

```
SELECT Customer.LastName, Customer.Phone
FROM Customer
INNER JOIN Sale ON Customer.CustomerID=Sale.CustomerID
INNER JOIN Animal ON Sale.SaleID=Animal.SaleID
WHERE ((Animal.Category=N'Cat') AND (Animal.Registered Is Not Null)
 AND (Color Like N'%White%')  AND (SaleDate Between '01-Jun-2013' And '31-Dec-2013'));
```

Figure 4.32

Joining multiple tables. QBE makes joining multiple tables relatively easy—just connect the tables with a line. With SQL, just start with two tables and expand outward; for example, start with (Animal INNER JOIN SaleAnimal ON Animal. malID = SaleAnimal.AnimalID), and then add a third table (Sale) with its JOIN.

in the Customer table. The Registration status, Color, and Category (Cat) are all in the Animal table. The SaleDate is in the Sale table. The Animal table connects to the Sale table through the SaleID column in the Animal table.

When the database contains a large number of tables, complex queries can be challenging to build. You need to be familiar with the tables to determine which tables contain the columns you want to see. For large databases, an entity-relationship diagram (ERD) or a class diagram can show how the tables are connected. Chapter 3 explains how Access sets referential integrity for foreign key relationships. Access uses the relationships to automatically add the JOINs to QBE when you choose a table. You can also use the class diagram to help users build queries.

When you first see it, the SQL 92 syntax for joining more than two tables can look confusing. In practice, it is best not to memorize the syntax. When you are first learning SQL, understanding the concept of the JOIN is far more important than worrying about syntax. Figure 4.33 shows the syntax needed to join three tables. To read it or to create a similar statement, start with the first table and JOIN it to a second table with the corresponding ON condition. Then JOIN the next table with a matching ON statement. Just be sure that the new table can be joined to one of the existing tables. Unfortunately, this syntax will not work in Microsoft Access, which requires the addition of parentheses. Figure 4.33 also shows an easier syntax that is faster to write when you are first developing a query or when you are in a hurry—perhaps on an exam. It is similar to the older SQL 89 syntax (but not exactly correct) where you list all the tables in the FROM clause and then join them in the WHERE statement.

```
SQL 92 Syntax for Three Tables
FROM Table1
  INNER JOIN Table2 ON Table1.ColA = Table2.ColA
  INNER JOIN Table3 ON Table2.ColB = Table3.ColB

Easier notation, But Not Correct Syntax
FROM Table1, Table2, Table3
JOIN          ColA      ColB
```

Figure 4.33

Joining multiple tables. With SQL 92 syntax, first join two tables within parentheses and then add a table and its JOIN condition. When you want to focus on the tables being joined, use the easier notation—just remember that it must be converted to SQL 92 syntax for the computer to understand it.

Hints on Joining Tables

Joining tables is closely related to data normalization. Normalization splits data into tables that can be stored and searched more efficiently. Queries and SQL are the reverse operation: JOINs are used to recombine the data from the tables. If the normalization is incorrect, it might not be possible to join the tables. As you build queries, double-check your normalization to make sure it is correct. Students often have trouble with JOINs, so this section provides some hints to help you understand the potential problems.

Remember that any time you use multiple tables, you must join them together. Most database query systems will accept a query even if the tables are not joined. They will even give you a result. Unfortunately, the result is usually meaningless. The joined tables also create a huge query result. Without any constraints most query systems will produce a cross join, where every row in one table is paired with every row in the other table.

Where possible, you should double-check the answer to a complex query. Use sample data and individual test cases in which you can compute the answer by hand. You should also build a complex query in stages. Start with one or two tables and check the intermediate results to see if they make sense. Then add new tables and additional constraints. Add the summary calculations last (e.g., Sum, Avg). It's hard to look at one number (total) and decide whether it is correct. Instead, look at an intermediate listing and make sure it includes all of the rows you want; then add the computations.

Columns used in a JOIN are often key columns, but you can join tables on any column. Similarly, joined columns may have different names. For example, you might join an Employee.EmployeeID column to a Sale.SalesPerson column. The only technical constraint is that the columns must contain the same type of data (domain). In some cases, you can minimize this limitation by using a function to convert the data. For example, you might use Left(ZipCode,5) = ZipCode5 to reduce a nine-digit ZipCode string to five digits. Just make sure that it makes sense to match the data in the two columns. For instance, joining tables on Animal.AnimalID = Employee.EmployeeID would be meaningless. The DBMS would actually accept the JOIN (if both ID values are integers), but the JOIN does not make any sense because an Employee can never be an Animal (except in science-fiction movies).

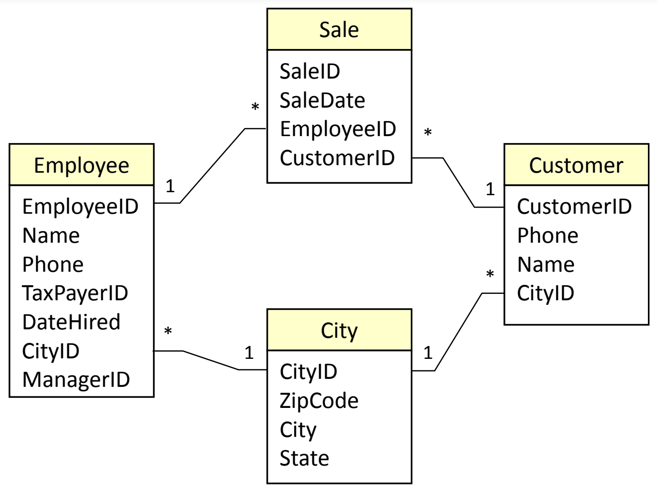


Figure 4.34

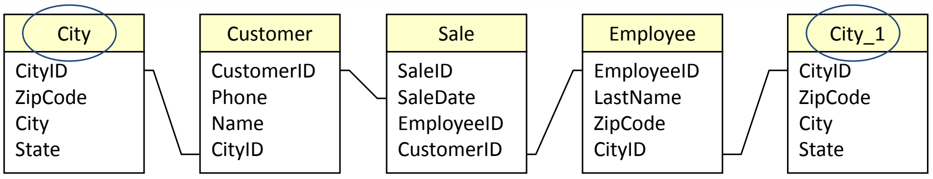
A query with these four tables with four JOINS would return only rows where the Employee had the same CityID as the Customer. If you need only the Customer city, just delete the JOIN between Employee and CityID. If you want both cities, add a second copy of the City table as a fifth table.

Avoid multiple ties between tables. This problem often arises in Access when you have predefined relationships between tables. Access QBE automatically uses those relationships to join tables in a query. If you select the four tables shown in Figure 4.34 and leave all four JOINs, you will not get the answer you want. The four JOINs will return Sales only where the Employee placing the order has the same CityID as the Customer! If you only need the City for the Customer, the solution is to delete the JOIN between Employee and City. In general, if your query uses four tables, you should have three JOINs (one less than the number of tables).

Sometimes it is helpful to remember that a JOIN condition also works as a row filter. The standard join will only return rows from a table that match those in the first table. For example, Sale.CustomerID = Customer.CustomerID will return customer data but only if those customers have already participated in a sale.

Table Alias

Consider the preceding Employee/Customer/City example in more detail. What if you really want to display the City for the Customer and the City for the Employee? Of course, you want to allow the cities to be different. The answer involves a little-known trick in SQL: just add the City table twice. The second "copy" will have a different name (e.g., City_1). You give a table a new name (**alias**) within the FROM clause: FROM City AS City_1. As shown in Figure 4.35, the City table is joined to the Customer. The City_1 table is joined to the Employee table. Now the query will perform two separate JOINs to the same table—simply because it has a different name.



```
SELECT Customer.CustomerID, Customer.CityID, City.City, Sale.EmployeeID,
Employee.LastName, Employee.CityID, City_1.City
FROM (City INNER JOIN (Customer INNER JOIN (Employee INNER JOIN Sale ON
Employee.EmployeeID = Sale.EmployeeID) ON Customer.CustomerID =
Sale.CustomerID) ON City.CityID = Customer.CityID) INNER JOIN City AS City_1 ON
Employee.CityID = City_1.CityID;
```

CID	Customer.CityID	City.City	EID	LastName	Employee.CityID	City_1.City
15	11013	Galveston	1	Reeves	11060	Lackland AFB
53	11559	Beaver Dam	2	Gibson	9146	Roanoke Rapids
38	11701	Laramie	3	Reasoner	8313	Springfield
66	7935	Danville	8	Carpenter	10592	Philadelphia
5	9175	Fargo	3	Reasoner	8313	Springfield

Figure 4.35

Table alias. The City table is used twice. The second time, it is given the alias City_1 and treated as a separate table. Hence, different cities can be retrieved for Customer and for Employee.

Create View

Any query that you build can be saved as a **view**. Microsoft simply refers to them as saved queries, but SQL and Oracle call them views. In either case the DBMS analyzes and stores the SQL statement so that it can be run later. If a query needs to be run many times, you should save it as a view so that the DBMS has to analyze it only once. Figure 4.36 shows the basic SQL syntax for creating a view. You start with any SELECT statement and add the line (CREATE VIEW …).

The most powerful feature of a view is that it can be used within another query. Views are useful for queries that you have to run many times. You can also create views to handle complex questions. Users can then create new, simpler queries based on the views. In the example in Figure 4.36, you would create a view (Kit-

Figure 4.36

Views. Views are saved queries that can be run at any time. They improve performance because they have to be entered only once, and the DBMS has to analyze them only once.

```
CREATE VIEW Kittens AS
SELECT *
FROM Animal
WHERE (Category = ‘Cat’ AND (Today-DateBorn < 180);
SELECT Avg(ListPrice)
FROM Kittens
WHERE (Color LIKE '%Black%');
```

Figure 4.37

Queries based on views. Views can be used within other queries.

tens) that displays data for Cats born within the last 180 days. As shown in Figure 4.37, users could search the Kittens view based on other criteria such as color.

As long as you want to use a view only to display data, the technique is straightforward. However, if you want a view that will be used to change data, you must be careful. Depending on how you create the view, you might not be able to update some of the data columns in the view. The example shown in Figure 4.38 is an updatable view. The purpose is to add new data for ordering items. The user enters the OrderID and the ItemID. The corresponding description of that Item is automatically retrieved from the Item table.

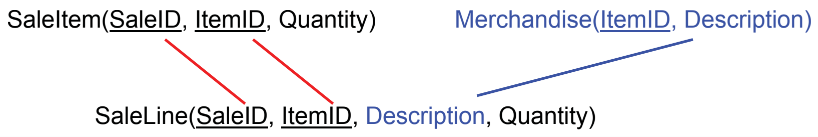
Figure 4.39 illustrates the problem that can arise if you are hasty in choosing the columns in a view. Here the OrderLine view uses the ItemID value from the Item table (instead of from the OrderItem table). Now you will not be able to add new data to the OrderLine view. To understand why, consider what happens when you try to change the ItemID from 57 to 32. If it works at all, the new value is stored in the Item table, which simply changes the ItemID of cat food from 57 to 32.

To ensure that a view can be updated, the view should be designed to change data in only one table. The rest of the data is included simply for display—such as verifying that the user entered the correct ItemID. You should never include primary key columns from more than one table. Also, to remain updatable, a view cannot use the DISTINCT keyword or contain a GROUP BY or HAVING clause.

Views have many uses in a database. They are particularly useful in helping business managers work with the database. A database administrator (DBA) or MIS worker can create views for the business managers, who see the section of the database expressed only in the views. Hence, you can hide the view's complexity and size. Most important, you can hide the JOINs needed to build the view, so managers can work with simple constraints. By keeping the view updatable, managers never need to use the underlying raw tables.

Figure 4.38

Updatable view. The OrderLine view is designed to change data in only one table (OrderItem). The Description from the Item table is used for display to help the user verify that the ItemID was entered correctly.


SaleItem(SaleID, ItemID, Quantity)
Merchandise(ItemID, Description)
SaleLine(SaleID, ItemID, Description, Quantity)

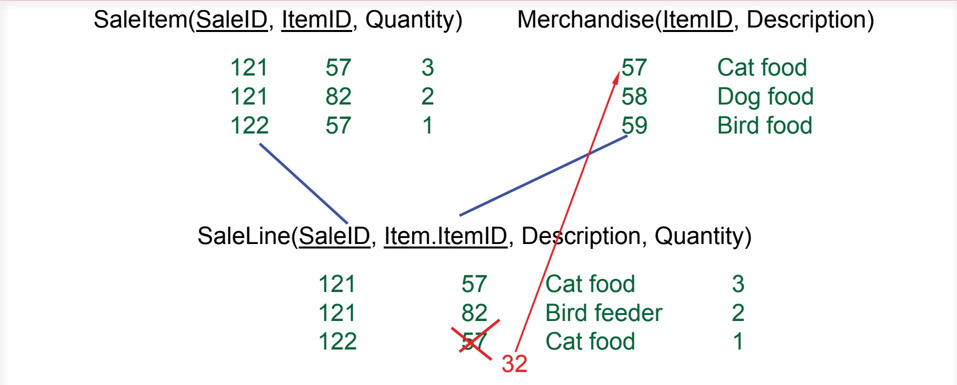

SaleItem(SaleID, ItemID, Quantity)
121 57 3
121 82 2
122 57 1
Merchandise(ItemID, Description)
57 Cat food
58 Dog food
59 Bird food
SaleLine(SaleID, Item.ItemID, Description, Quantity)
121 57 Cat food 3
121 82 Bird feeder 2
122 57 Cat food 1
32

Figure 4.39

Nonupdatable view. Do not mix primary keys from different tables. If this view works at all, it will not do what you want. If you try to change the ItemID from 57 to 32, you will only change the ItemID of cat food. You will not be able to enter new data into the OrderItem table.

Note that some database systems place restrictions on commands allowed within a view. For example, older Oracle and newer SQL Server systems do not allow you to use the ORDER BY clause in a saved view. The reason for this restriction was to enable the system to provide better performance by optimizing the query. To sort a result, you had to add the ORDER BY statement to a new query that called the saved view. Finally, no matter how careful you are at constructing a view with a JOIN statement, the DBMS might still refuse to consider it updateable. When the DMBS accepts it, updateable views can save some time later when building forms. But, at other times you have to give up and go with simpler forms.

Newer Searches and Patterns

Note: This section covers XQuery and RegEx searches for XML data and text strings. It could be skipped or covered later. Be sure you understand basic SQL commands before dealing with this material.

How do you search XML and complex text strings? Over time, companies have found the need to store complex data in databases. Although most DBMSs can store new and different types of data, it also becomes important to retrieve that data. The standard WHERE conditions apply only to the basic data types (simple numbers and text). A few types of complex data have become important and common enough that vendors have adopted standard methods to search these new data types. The two most common types of data are: XML hierarchies and long text.

XML is stored as tagged data, and an entry is commonly organized as a hierarchy. A parent node can have multiple child nodes. For instance, an <Order> tag can have multiple <Item> tags to indicate which items are being ordered. Developers and users need a common method to search an XML tag, including the ability to drill down and list elements within the hierarchy. XQuery was developed as a standard to perform these searches. Today, the SQL 2006 standard and most of the big DBMSs support the XML data type and the use of XQuery to search XML data.

```
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2010</ShipDate>
<Items>
 <Item>
  <ItemID>15</ItemID>
  <Description>Leash</Description>
  <Quantity>20</Quantity>
  <Price>8.95</Price>
 </Item>
 <Item>
  <ItemID>32</ItemID>
  <Description>Collar</Description>
  <Quantity>25</Quantity>
  <Price>14.50</Price>
 </Item>
 </Items>
</shipment>
```

Figure 4.40

Sample XML data. Assume vendors send a shipping invoice file when they send products. The sample data shows a single shipment that contains a ShipID and ShipDate. The repeating section contains a list of items and the quantity that were shipped.

The basic pattern matching provided by the SQL standard is somewhat simplistic. With only two search symbols (all text or one character), it is relatively easy to use. But it is not very powerful. Programmers have long had a powerful string search tool called regular expressions. Technically, regular expressions were added to the SQL 1999 standard, but only recently have vendors added it as a feature.

XQuery

XQuery is a standardized method for retrieving values from an XML string. XML uses tags to mark each item and the designer can create almost any terms for the tags. But the XML string has to be well-formed and can be validated against a schema that specifies the data model. An XML data model is essentially a hierarchical definition of the data and repeating elements that can be stored in the XML string. Figure 4.40 shows a simple XML file with sample data. When vendors ship products to Sally's Pet Store, they are asked to send this XML file that contains the shipping invoice data. The vendor provides a ShipID and the ShipDate to reference their data in case questions arise later. The repeating Items section contains a list of the items that were shipped along with the quantity and price paid. This example is intentionally kept simple. A real-world invoice could have many levels and options. Note that all XML tags are case-sensitive.

For illustration, Figure 4.41 shows a simple version of the **XML schema** for the sample data. The schema is the data definition and it can be used to create and to validate XML data files. Note how the hierarchical form is defined through the nested elements. In this case, the reference to Items is listed within the main shipment. Also, note that each data point is defined by an element and the ele-

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
 <xsd:element name="shipment">
  <xsd:complexType>
   <xsd:element name="ShipID" minOccurs="0" type="xsd:int" />
   <xsd:element name="ShipDate" minOccurs="0" type="xsd:date" />
   <xsd:sequence>
    <xsd:element ref="Items" minOccurs="0" maxOccurs="unbounded" />
   </xsd:sequence>
  </xsd:complexType
 </xsd:element>
 <xsd:element name="Items">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="ItemID" minOccurs="0"  type="xsd:int" />
    <xsd:element name="Description" minOccurs="0" />
    <xsd:element name="Quantity" minOccurs="0" type="xsd:int" />
    <xsd:element name="Price" minOccurs="0" type="xsd:double" />
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 4.41

Simple XML schema for sample data. Note the hierarchical definition through the nested elements.

ment specifies the type of data. The default data type is text. XML and schemas support considerably more complex data specifications, but it is best to start with the simple formats. One way to create an XML data file and a schema definition is to export a table from Microsoft Access as an XML file. This approach adds some overhead to define a few Office/Access features; but it is a relatively painless method to create an XML schema.

Storing XML Data

Unfortunately, Microsoft Access tables do not directly support the XML data type and XQuery. You will need to use SQL Server or Oracle to work with the remaining examples. A few standalone tools found on the Web can also be used to learn and experiment with XQuery.

SQL Server (and Oracle) can handle XML both as data stored within a table and as a data variable within the programming language. The underlying concepts are the same, but since this chapter focuses on tables and queries, the examples here use XML data placed into a table. Figure 4.42 shows the CREATE TABLE and INSERT commands to create a new ShippingInvoice table and insert the sample data. Notice that the entire XML string is inserted into a single cell (row/column) of the ShippingInvoice table. That is, the table contains one row to represent the single invoice and all of the XML data goes into the XML Contents column.

Retrieving XML Data with XQuery

XML data is basically a large string. You can retrieve the entire XML string with a relatively standard SELECT statement, simply by specifying the desired row.

```
CREATE TABLE ShippingInvoice (
            ShippingID int IDENTITY(1,1) NOT NULL,
            InvoiceDate date NULL,
            OrderID int NULL,
            Contents xml NULL,
 CONSTRAINT pk_ShippingInvoice PRIMARY KEY (ShippingID)
)
GO
INSERT INTO ShippingInvoice (InvoiceDate, OrderID, Contents)
VALUES ('19-May-2013', 12, '
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2013</ShipDate>
<Items>
 <Item><ItemID>15</ItemID><Description>Leash</Description>
   <Quantity>20</Quantity><Price>8.95</Price></Item>
 <Item><ItemID>32</ItemID><Description>Collar</Description>
   <Quantity>25</Quantity><Price>14.50</Price></Item>
  </Items>
</shipment>
');
```

Figure 4.42

SQL Server table with XML data and INSERT command.

Figure 4.43

Simple XQuery example. The SQL SELECT statement specifies the desired row and column that contains the XML data. The .query('shipment/Items') term is a new element that builds the XML query.

```
SELECT Contents.query('shipment/Items') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;
<Items>
  <Item>
    <ItemID>15</ItemID>
    <Description>Leash</Description>
    <Quantity>20</Quantity>
    <Price>8.95</Price>
  </Item>
  <Item>
    <ItemID>32</ItemID>
    <Description>Collar</Description>
    <Quantity>25</Quantity>
    <Price>14.50</Price>
  </Item>
</Items>
```

However, you will often need to use XQuery to extract particular elements from the XML string. Figure 4.43 shows one of the simplest XQuery examples. The SQL SELECT statement is used to specify the desired row (ShippingID=1) and the column (Contents) of the XML document. The .query('shipment/Items') is the new term that activates XQuery. XQuery supports several relatively complex capabilities for searching an XML document. Only a few commonly-used examples are given here. Once you understand the basic structure of XML and XQuery, you can study the reference documents to create more complex queries. However, keep in mind that instead of building a hugely complex XQuery, it is often better to extract all of the XML data and store it directly in relational tables. Then you can use the power and speed of SQL to retrieve the data.

The simplest form of an XQuery is to retrieve a segment of the XML document. The clearest way to do that is to simply specify the desired segment from the top down. In the sample data, "shipment" is the root node and "Items" is the repeating section. Hence, the query 'shipment/Items' returns the entire entry under the "Items" node.

What if you want to retrieve data based on a specific value stored within a node? For instance, you want to retrieve all of the information for ItemID 15 in the sample data. XQuery has a couple of methods for retrieving this data. Figure 4.44 shows the simplest approach: /shipment/Items/Item[ItemID=15]. Specify the hierarchical structure and then enter the desired condition in square brackets [ ].

Another useful trick is to return just a single element within a node. In the prior example, instead of returning the entire entry for ItemID 15, it might be useful to retrieve only the Quantity shipped. Figure 4.45 shows the syntax for specifying a single node. Simply add the node name after the brackets: /shipment/Items/Item[ItemID=15]/Quantity.

Complex search conditions are available, including a "contains" function to search text elements for specific values. Elements can also include attributes, such as <Price currency="USD">8.95</Price>, which uses a currency attribute to specify the monetary units. (USD is the standard symbol for U.S. Dollar). To search for this particular attribute, you could use a query of the form: /shipment/

Figure 4.44

XQuery to retrieve entry based on value stored within an element. Find entry for ItemID 15.

```
SELECT Contents.query('
  /shipment/Items/Item[ItemID=15]
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

<Item>
  <ItemID>15</ItemID>
  <Description>Leash</Description>
  <Quantity>20</Quantity>
  <Price>8.95</Price>
</Item>
```

```
SELECT Contents.query('
  /shipment/Items/Item[ItemID=15]/Quantity
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

<Quantity>20</Quantity>
```

Figure 4.45

Extract a single element from the found node by adding the element name (/Quantity) after the brackets.

Items/Item/Price[@currency="USD]; which will return only those items where the price currency is directly specified as USD.

Several other search options are available, but they use a somewhat cryptic annotation mechanism. XQuery also supports a relatively flexible search language that is often easier to read. It is abbreviated as FLWOR: for, let, where, order by, return. Figure 4.46 shows an example of the "for" loop that examines all of the items, searching for an entry with ItemID equal to 15. Note the use of an internal variable ($item) that is used to temporarily hold the values of each node being examined. The query also uses the data function to extract the value and return the simple number (20) without its XML tags. This data function could have been used in the prior examples as well. The point of the "for" loop is that it returns all of the nodes that meet the specified criteria, so it can return multiple "rows" of data—although all of the "rows" are stored within the single XML document. The return statement also supports an if/then/else construct so you can return modified results based on a conditional test.

XQuery in Oracle

Oracle support for XML and XQuery is similar to the examples in this section, but the syntax is slightly different. First, the data type needed is XMLType. Also, note that "contents" is a reserved word in Oracle, so the column name in the ShippingInvoice table should be changed to xContents. If you create a sequence and an insert trigger for the table, the existing INSERT command will work. However, it

Figure 4.46

Using FLWOR commands to search nodes and return a single value. Also uses the data function to return the value without the XML tags.

```
SELECT Contents.query('
  for $item in /shipment/Items/Item
    where $item/ItemID=15
  return data($item/Quantity)
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

20
```

is probably easier to just modify the INSERT command to include the ShippingID and specify the ShippingID value as 1. So, the setup commands are:

```
CREATE TABLE ShippingInvoice (
    ShippingID number NOT NULL,
    InvoiceDate date,
    OrderID number,
    xContents XMLType,
 CONSTRAINT pk_ShippingInvoice PRIMARY KEY (ShippingID)
);
INSERT INTO ShippingInvoice (ShippingID, InvoiceDate,
OrderID, xContents)
VALUES (1, '19-May-2013', 12, '
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2013</ShipDate>
<Items>
  <Item><ItemID>15</ItemID><Description>Leash</
Description>
    <Quantity>20</Quantity><Price>8.95</Price></Item>
  <Item><ItemID>32</ItemID><Description>Collar</
Description>
    <Quantity>25</Quantity><Price>14.50</Price></Item>
  </Items>
</shipment>
');
```

The syntax for calling XQuery is also somewhat different. However, the XQuery functions are mostly standardized. The basic command matching Figure 4.43 shows the difference using the extract function:

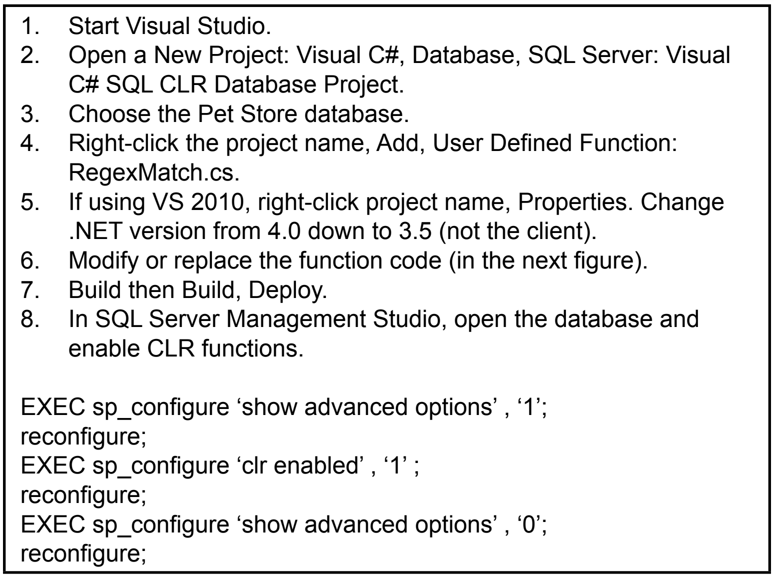
```
SELECT extract(xContents, '
   /shipment/Items
') As ListResult
FROM ShippingInvoice
WHERE ShippingID=1;
```

Once you understand the syntax, the method of using XQuery is the same as the other examples. But always be sure to test everything.

Summary

XQuery is a powerful tool for searching XML data trees. However, keep in mind that all searches through XML are based on string values and they are rarely (if ever) indexed. Consequently, XQuery searches can be quite slow. Additionally, you have to be cautious and test all of your queries to ensure they are retrieving exactly the requested data and not missing anything. If XML data needs to be searched often, it is better to extract the individual elements and put them into standard relational tables. Then use SQL to perform the searches. XQuery could be used to perform the data extraction. Remember that XQuery works on data in one row and one cell at a time.

Ultimately, a designer must make the decision of whether to extract XML data into relational tables or to leave it stored as a single XML document. If the data is rarely used except for occasional searches, it can probably be left as an XML document. However, it will still be necessary to have someone around who can



1. Start Visual Studio.
2. Open a New Project: Visual C#, Database, SQL Server: Visual C# SQL CLR Database Project.
3. Choose the Pet Store database.
4. Right-click the project name, Add, User Defined Function: RegexMatch.cs.
5. If using VS 2010, right-click project name, Properties. Change .NET version from 4.0 down to 3.5 (not the client).
6. Modify or replace the function code (in the next figure).
7. Build then Build, Deploy.
8. In SQL Server Management Studio, open the database and enable CLR functions.

```
EXEC sp_configure 'show advanced options' , '1';
reconfigure;
EXEC sp_configure 'clr enabled' , '1' ;
reconfigure;
EXEC sp_configure 'show advanced options' , '0';
reconfigure;
```

Figure 4.47

Steps to create a CLR project in SQL Server and enable CLR functions in the database.

write and test XQuery code for those times when a manager does need to search the data. As shown in the examples, it is possible to prebuild SQL views that contain XQuery searches. These views can be saved and run later. The tools include the ability to reference SQL parameter variables within the XQuery so these views can be controlled through other code. However, because of the complexity and tricky nature of XML queries, avoid giving users the ability to create their own XQuery searches.

Regular Expressions (RegEx) Patterns

Increasingly, applications are being built that contain unstructured text data. For instance, a database might hold open-ended comments entered by workers or customers; or a database might be built to hold boilerplate paragraphs for use in contracts or negotiations. Think in terms of the open content on the Web, but it is data stored in internal databases. Now think about how people will want to search this data. Many times, they will want to enter keywords or phrases or even more complex conditions to find matches to sophisticated patterns. If the data consists of HTML or PDF pages stored on an internal Web server, it is possible to purchase commercial search engines to help index and find pages. But how are you going to create searches for text stored in a relational database? Basic SQL pattern matching was discussed in an earlier section of this chapter. It consists of two wildcard characters (% and _ in the standard) that match any characters or any single character. This basic approach is not going to be enough. To address these issues, SQL 1999 added support for **regular expressions**, usually abbreviated RegEx. It has taken vendors a few years, but the big systems now support regular expression searches.

Regular expressions were created many years ago and were heavily used by programmers—particularly on UNIX-based systems. Today, most programming languages support them, and the big DBMSs also support their usage for matching text values. Microsoft Access does not support them for table searches; however, any code written in one of the Microsoft languages (Visual Basic, C#, C++ and so on) has regular expression processing which can be applied to the rows retrieved from queries.

Some systems, such as MySQL, support regular expressions directly as part of the query. For example, Oracle defines functions for Regexp_Count, Regexp_Instr, Regexp_Like, and Regexp_Replace. The Regexp_Like function is similar to an extended Like command used in the WHERE clause. SQL Server is more complicated to set up, but SQL server also has a simpler option that extends the standard LIKE command.

Regular expressions have powerful options supporting many complex types of searches. On the simple side, a pattern can search for a single word. More complex patterns can be created to see if an entered string is an e-mail address. Patterns can be written to search for repeating characters or phrases.

SQL Server Setup

Beginning with SQL Server 2005, Microsoft added the ability to create user-defined **common-language runtime (CLR)** functions in SQL Server. CLR functions are written in a Visual Studio language, such as C#, compiled and installed into the database so that they can be used as functions within SQL. The process takes a few steps to set up the function, but once the function is installed, it is used

Figure 4.48

Microsoft C# function to create the RegexMatch function for use in SQL Server.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
   public static readonly RegexOptions Options =
   RegexOptions.IgnorePatternWhitespace |
   RegexOptions.Singleline;

   [Microsoft.SqlServer.Server.SqlFunction]
   public static SqlBoolean RegexMatch(
      SqlChars input, SqlString pattern)
   {
      Regex regex = new Regex( pattern.Value, Options );
      return regex.IsMatch( new string( input.Value ) );
   }
};
```

much like the functions in the other database systems. Figure 4.47 outlines the steps needed to create a CLR function and enable it to work within the database.

Figure 4.48 shows the entire C# code needed to create the RegexMatch function. Visual Studio generates much of the code automatically, but it needs to be edited. It might be easier to use copy-and-paste to replace the entire function.

Use the Build option to compile the function and correct any errors. Use the Build | Publish menu option and Visual Studio will compile the function and install it in the database. From that point, you can use the new function (RegexMatch) to examine any string or table column for patterns.

RegEx Patterns

Regular expressions are powerful search tools, but they can be cryptic and hard to follow. This section presents only the basic concepts to get started. Once you are comfortable with these tools, you can use tutorials and reference documents on the Web to learn more detailed techniques if you need them. For example, RegEx also supports search and replace for patterns, but this option is not covered here.

The regex function uses two string parameters: (1) The text to be searched, and (2) A regex pattern. In a database context, the text to be searched can be a column from a table and the function can be written into the SELECT clause or the WHERE clause. For instance, consider searching the Merchandise table in the Pet Store database. Figure 4.49 illustrates using the RegEx function to search for any description containing the word “Small.” A plain string is the simplest search pattern. Note that patterns are case-sensitive by default. The simple pattern will match a row if the pattern exists exactly as written anywhere within the column data.

One useful tool of RegEx is the ability to specify a range of characters using square brackets. For example, as shown in Figure 4.50, the pattern [a-z] would match a single letter between “a” and “z” and only in lower-case form. The hyphen is a range indicator but it is also possible to specify individual characters. For example, the pattern [AEIOU] would match any one of the vowels in the list, or [123] would match one of the three digits.

The Microsoft SQL Server LIKE command also supports the square brackets—even without implementing the regular expression function. Hence, if all you need is a simple pattern to check for individual characters or ranges of characters, you

Figure 4.49

A simple text pattern search using RegEx in the LIKE clause. By default, RegEx comparisons are case-sensitive. Entering a simple string will try to match that pattern anywhere within the Description column.

```
SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N'Small') <> 0;

1     Dog Kennel-Small    11    45.00    Dog
5     Cat Bed-Small       36    25.00    Cat
32    Collar-Dog-Small    47    12.00    Dog
```

[a-z] [AEIOU] [123] [^0-9]	Match one lower-case letter. Match one of the vowels. Match one of the numbers. Match a character not a digit.

```
SELECT *
FROM Customer
WHERE dbo.RegexMatch(LastName, N'H[ai]ll') <> 0;
```

```
78   (505) 646-2748   Elaine   Hall …
```

Figure 4.50

Groups of characters using square brackets. Will match if a single character matches one of the pattern characters. The example would find customers with a last name of Hill or Hall. The caret (^) negates the pattern.

can use the square brackets directly within the LIKE clause. For example, consider the clause:

```
      WHERE LastName LIKE N'Sm[iy]th'
```

This clause will match either Smith or Smyth because the brackets accept either the "i" or the "y" character.

RegEx has several special characters—many of which handle common ranges that are useful for various comparisons. Figure 4.51 shows the most commonly-used characters. Note the case-sensitive characters. Upper-case letters generally mean the negation or reverse of the lower-case symbol, such as "d" for digits and "D" for anything except digits. The caret (^) and dollar sign ($) are useful because they anchor the string comparison to the start or end of the input text. Without these, the pattern is always tested at any point within the text string. For instance,

Figure 4.51

Special characters. Many of them match commonly-used ranges such as digits or alphanumeric characters.

. (dot)	Match any single character.
\n	Match newline character.
\t	Match the tab character.
\d	Match a digit [0-9].
\D	Match a non-digit [^0-9].
\w	Match an alphanumeric character.
\W	Match a non-alphanumeric character.
\s	Match a whitespace character.
\S	Match a non-whitespace character.
\	Escape special characters, such as \.
^	Match the beginning of the input.
$	Match the end of the input.

a simple pattern “One” would be tested and could appear at any point in the search text. However, the pattern “^One” will only match if the word “One” appears exactly at the start of the input.

So far, the patterns apply to a single character or word at a time. In many cases, it is useful to allow a digit or character to repeat. RegEx has several ways to quantify the number of characters. Three special characters are useful: * ? +. The asterisk (*) matches any number of what falls before it—from zero to infinity. The question mark (?) matches zero or one of the pattern preceding it. The plus sign (+) matches one or more of the pattern before it. Figure 4.52 summarizes these differences and provides an example of the difference between the asterisk and the plus sign. You should run the two queries and check the results. Almost all of the merchandise rows will appear when using the asterisk because the “n” is optional and most rows contain the letter “e”.

These special characters cover cases of zero, one, and infinite repetition. In some cases, you want the ability to specify exactly how many times a pattern should repeat. The RegEx pattern for that is to put the number in curly braces, such as: \d{3}. The letter \d specifics a numeric digit and the {3} repetition annotation says exactly three digits must exist. Figure 4.53 shows an example of using the fixed repetition to test a U.S. Social Security number. This example comes from Microsoft’s MSDN article. U.S. Social Security numbers consist of nine digits, commonly written in three groups separated by hyphens, such as 123-45-6789. The RegEx pattern is straightforward. For example, the term \d{3} tests for the presence of exactly three digits. In the full pattern, note the use of the start (^) and end ($) markers to prevent the introduction of extraneous characters.

RegEx patterns can be much more complex. One useful feature is the ability to group characters together using parentheses ( ). Figure 4.54 shows some straightforward examples of creating groups. Anything placed in parentheses is treated as a group, so the pattern (ab)+ searches for at least one occurrence of the two

Figure 4.52

Repetition Characters. The characters apply to the pattern immediately preceding the character. Notice that the asterisk is less useful than it appears because the zero means the pattern does not have to exist.

*	Match zero or infinite of the pattern before the asterisk.
?	Match zero or one of the pattern before.
+	Match one or more of the pattern before.

```
SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N’n*e’)<>0;
>>>> Match any description that contains the letter “e”.
(Because the * means the n is optional.

SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N’n+e’)<>0;
>>>> Match any description that contains the letter “n” followed by any
characters and then the letter “e”.
(Because the + means the n is required.
```

{#} such as {3}   Exact number of repetitions.

```
SELECT dbo.RegexMatch(N'123-45-6789', N'^\d{3}-\d{2}-\d{4}$' )
```

Pattern to test a U.S. Social Security Number. The string must begin with 3 digits, then a hyphen, then 2 digits, another hyphen, and end with exactly 4 digits.

Figure 4.53

Exact repetition. Enter the exact number of repetitions in curly braces. A useful method when a data element must have an exact number of digits, spaces, or characters.

characters “ab” together. The figure also shows that grouping is more powerful when the “Or” connector (|) is added. The pattern (aa|bb)+ searches for at least one occurrence of either “aa” or “bb” (or both). Be careful to note the difference between square brackets and parentheses. Brackets represent a single character to be matched while parentheses require the entire term to be matched. In the example, [ab] would match either the letter a or b, so the string “acb” would be matched as true because it contains an “a” (and a “b”). However the pattern (ab) does not match the string “acb” because the parentheses require an exact match of the entire term and the pattern “ab” does not exist in the string “acb”.

Summary

This section is merely an introduction to regular expressions. Several other features exist, and you can find many tutorials and reference works on the Web. However, before trying to learn, and memorize, the many features of expressions you need to practice the simpler versions so that you completely understand these features. Regular expressions are often combined into long, complex pattern strings. These patterns can be difficult to read and debug. They are even harder when someone else has written the pattern. Whenever you create regular expression patterns, be sure you document them and explain the objectives. Better yet, you should always create sample test cases before you try to write a RegEx pattern. Create or find real-world sample data that includes several cases that you do and do not want to match. As you build the pattern, you can test it in sections against the sample data. More importantly, if anyone modifies the patterns later, they can be re-tested against the data to ensure they are still correct.

Be cautious when creating regular expressions—particularly if they are intended for use at restricting data entry. For example, it is tempting to create a pattern to force people to enter telephone numbers in a specific manner. But, what hap-

Figure 4.54

Grouping patterns with parentheses and using the Or connector: |.

(ab)+ matches ab, abab, tab, but not acb.
(aa|bb)+ matches aa, bbcc, bbddaa, but not abab.

pens when someone needs to enter a phone number that does not match the pattern? For example, international phone numbers require more digits (international code), and generally do not follow the same pattern as U.S. phone numbers. Similarly, be cautious writing patterns for e-mail addresses. Data formats and usages change over time.

Regular expressions are powerful tools, but they carry a price. Because of their complexity, they are difficult to optimize and rarely used with indexes. In most cases, the query processor needs to retrieve every single row, apply the pattern, and then decide whether the row is included in the results. This process can be time consuming if the query retrieves millions, billions, or trillions of rows of data. When users truly need to search everything, the delay is probably acceptable. However, it is best to try and write a query without using regular expressions—particularly for queries with multiple tables and JOINs.

Summary

The key to creating a query is to answer four questions: (1) What output do you want to see? (2) What constraints do you know? (3) What tables are involved? (4) How are the tables joined? The essence of creating a query is to use these four questions to get the logic correct. The WHERE clause is a common source of errors. Be sure that you understand the objectives of the query. Be careful when combining OR and AND statements and use DeMorgan's law to simplify the conditions.

Always test your queries. The best method to build complex queries is to start with a simpler query and add tables. Then add conditions one at a time and check the output to see whether it is correct. Finally, enter the computations and GROUP BY clauses. When performing computations, be sure that you understand the difference between Sum and Count. Remember that Count simply counts the number of rows. Sum produces the total of the values in the specified column.

Joining tables is straightforward. Generally the best approach is to use QBE to specify the columns that link the tables and then check the syntax of the SQL command. Remember that JOIN columns can have different names. Also remember that you need to add a third (or fourth) table to link two tables with no columns in common. Keep the class diagram handy to help you determine which tables to use and how they are linked to each other.

A Developer's View

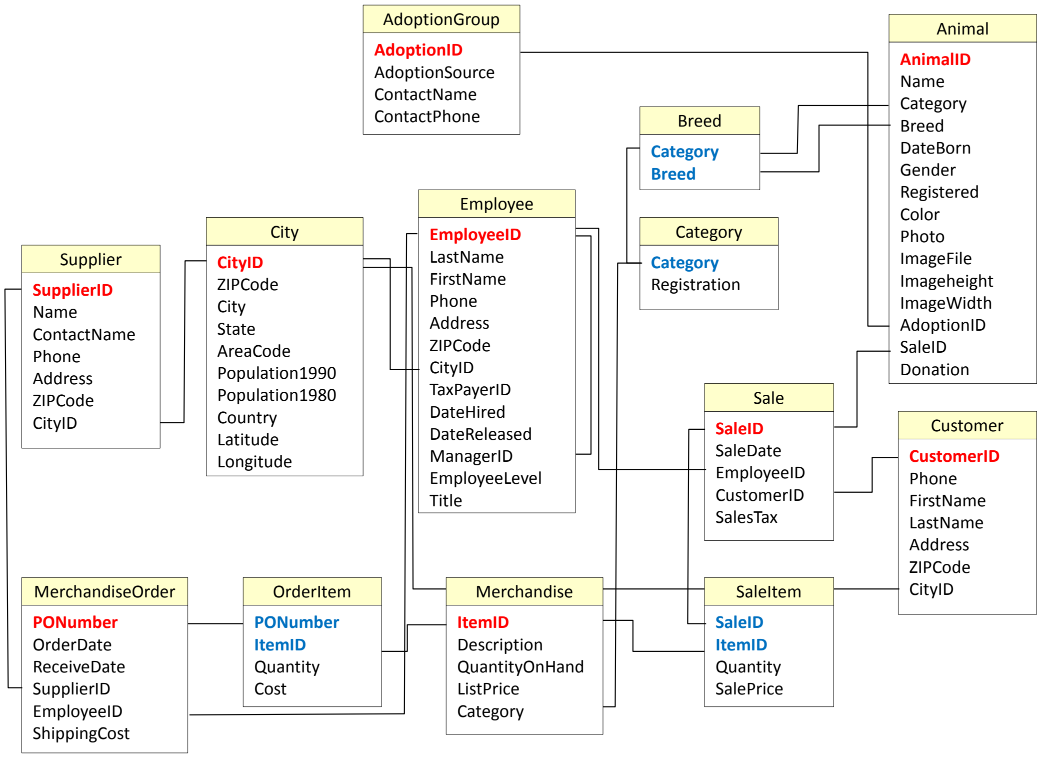
As Miranda noted, SQL and QBE are much easier than writing programs to retrieve data. However, you must still be careful. The most dangerous aspect of queries is that you may get a result that is not really an answer to the business question. To minimize this risk, build queries in pieces and check the results at each step. Be particularly careful to add aggregation and GROUP BY clauses last, so that you can see whether the WHERE clause was entered correctly. If you name your columns carefully, it is easier to see how tables should be joined. However, columns do not need the same names to be joined. For your class project, you should identify some common business questions and write queries for them.

Key Terms

aggregation
alias
BETWEEN
Boolean algebra
common-language runtime (CLR)
cross join
data definition language (DDL)
data manipulation language (DML)
DeMorgan’s law
DESC
DISTINCT
FETCH
FROM
GROUP BY
HAVING
JOIN
LIKE
NOT
NULL
ORDER BY
query by example (QBE)
regular expression (RegEx)
row-by-row calculations
SELECT
SQL
TOP
view
WHERE
XML schema
XQuery

Review Questions

1. What are the three basic tasks of a query language?
2. What are the four questions used to create a query?
3. What is the basic structure of the SQL SELECT command?
4. What is the purpose of the DISTINCT operator?
5. Why is it important to use parentheses in complex (Boolean) WHERE clauses?
6. How is pattern matching used to select rows of data?
7. What is DeMorgan’s law, and how does it simplify conditions?
8. How do you compute subtotals using SQL?
9. How do the basic SQL arithmetic operators (+, -, etc.) differ from the aggregation (SUM, etc.) commands?
10. What basic aggregation functions are available in the SELECT command?
11. What is the difference between Count and Sum? Give an example of how each would be used.
12. What is the difference between the WHERE and HAVING clauses? Give an example of how each would be used.
13. What is the SQL syntax for joining two tables?
14. How do you identify a column when the same name appears in more than one table?
15. What is XQuery and when would you use it?
16. What are regular expressions? What are their strengths and weaknesses?

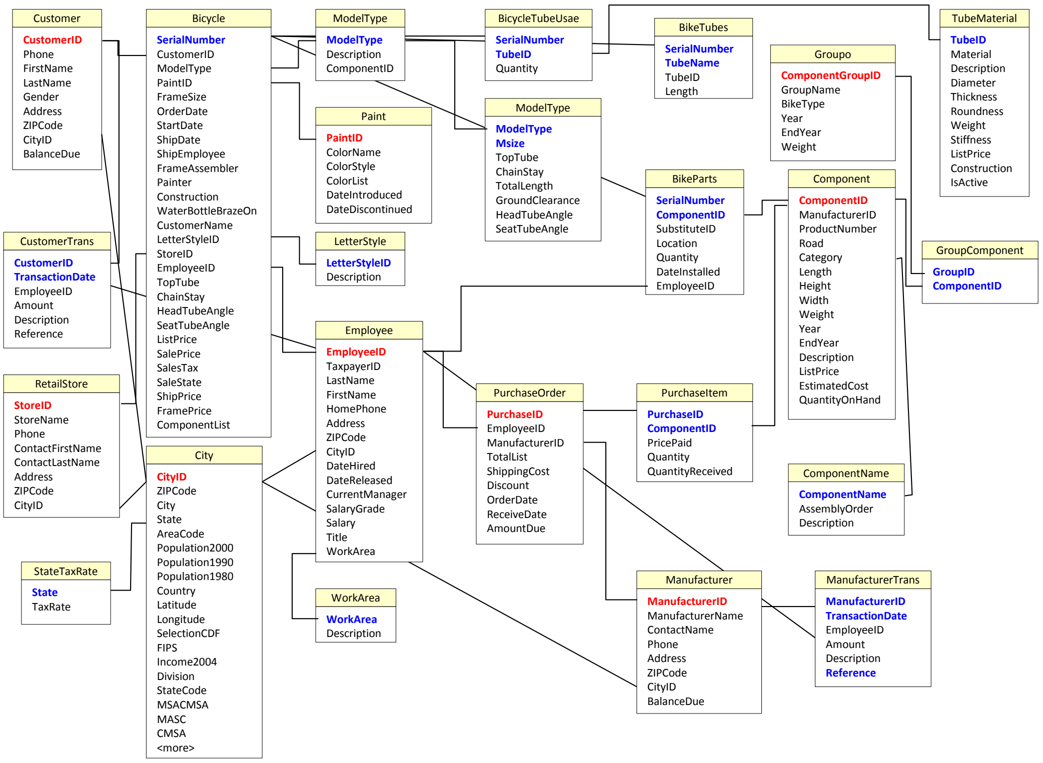


Exercises

Sally's Pet Store

1. Which employee still working for the store was hired the most recently?
2. What is the largest quantity of items ever ordered/purchased by the store at one time?
3. List all cats with no black in their coloring.
4. List customers from Tennessee (TN) who bought cat merchandise.
5. List employees who participated in adoptions of female dogs in March.
6. List customers who bought a dog kennel in March.
7. List the name and contact information for suppliers in Nebraska (NE).
8. List the items sold in May with no duplicates.
9. List the name and phone number of each customer who adopted an animal in February.
10. List the adoption groups with phone number who placed cats in October.
11. List all of the employees who are managed by Katy Reasoner.
12. What is the largest value of sale ever made?

13. Which adoption group has placed the most animals?
14. Which day of the week (Sun/Mon/…) has the highest total sales value? Hint: In Access use the function Format(date, “ddd”) to obtain the day of the week.
15. Are male dogs more likely to be registered than the females?
16. What was the most popular item sold in May (by quantity)?
17. By total value, which supplier has the highest sales for the year?
18. Does the store have more money tied up in inventory (quantity on hand) for dog items or cat items?
19. On average by supplier, how long does it take to receive an order from suppliers?
20. By value, which category of items were sold the most in the second quarter of the year?
21. Which employee sold the most quantity of items in March?
22. By count of state, where are most of the customers located?
23. Which category of items had the highest sales value in May?
24. From which supplier did the store purchase the most cat merchandise by value?
25. Which sale had the highest total discount (ListPrice – SalePrice)*Quantity?

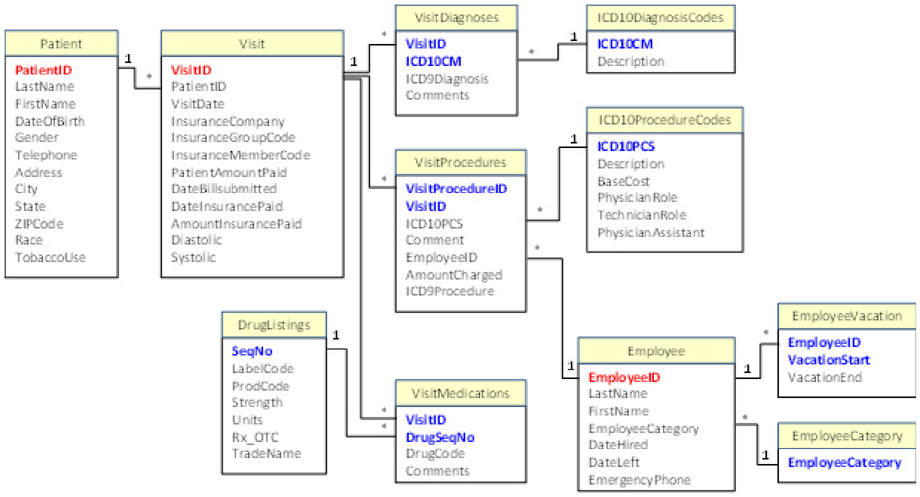


Rolling Thunder Bicycles

Write the SQL statements that will answer questions 26 through 50 based on the tables in the Rolling Thunder database.

26. List customers (name, phone) who bought race bikes in 2012 with a frame size greater than 60 cm.
27. List the component product number and weight that are in the SRAM Red 2012 groupo.
28. Which full suspension bikes sold in 2012 were equipped with SRAM (manufacturer) cranks?
29. List the employees who sold race bikes with a sale price of more than $9000 in 2010.
30. List the retail stores (ID > 2) that participated in selling hybrid bikes in 2012.
31. List the phone number of all women who purchased race bikes with white in the color in 2012.
32. For future correlation analysis, list the sale price, most recent population and per capita income for the city where it was purchased in 2013.
33. List all of the employees who placed purchase orders with Shimano (manufacturer) in 2012 with a total list value over 120,000.

34. Looking for tall riders, list all customers who purchased road bikes in 2012 with a frame size greater than or equal to 62 cm for men and 60 cm for women.
35. ✓ Find the greatest percentage discount (Discount/TotalList) received on a purchase order placed in 2013.
36. Compute the total price of components installed on each Road bike sold in 2013 (without using the ComponentList value in the Bicycle table).
37. Compute the average price paid for Campy Record 11 cranks purchased in 2013.
38. What was the most popular letter style in 2012 on all Mountain bikes?
39. ✓ How many Race bikes were sold to women in each state in 2013?
40. On average, not counting frames, what is the most expensive (list price) category of component carried for the 2012 component year?
41. In 2012, which model type carried the highest average sale price?
42. For the years 2010-2013 (inclusive), did men or women pay higher average prices for Road bikes?
43. ✓ Compute the total value and count of sales by month for 2010 through 2013. Hint: In Access, use the format string "yyyy-mm", or Year(…)*100+Month(…) for others.
44. ✓ Which customer purchased the most number of Road and Race bikes combined (for all dates)?
45. Show how the average weight of rear derailleurs for Road bikes has changed over time.
46. What is the total tax value collected and owed to each state for bikes sold in 2013?
47. ✓ Which employee sold the most bikes by value in November 2012?
48. What is the average percentage of shipping cost per total list by manufacturer for orders placed in 2013?
49. For 2013, did the average percent discount on bike prices vary significantly based on model type?
50. Which manufacturer made the most popular crank installed on Race bikes in 2013?



Corner Med

51. Which patient paid the most amount of money directly for a single visit in April 2013?
52. List all of the drugs prescribed on June 24, 2013.
53. List the employees who treated patients on July 4, 2013; without duplicates.
54. List the female patients who do not use tobacco and were between 50 and 55 at the time of the visit. Hint: Use the DateDiff function to compute the age.
55. List all of the procedures and amounts charged for patients treated by Dr. Johnson on May 15.
56. List any patients who were older than 60 and had a systolic pressure below 120 (but not zero).
57. List all of the ICD10 diagnostic codes that refer to the esophagus.
58. List the visits where the patient paid nothing up front and the insurance company took longer than 90 days to pay the bill (DateInsurancePaid – DateBillSubmitted).
59. List all of the physicians who prescribed the drug Ambien in October.
60. List all of the patients (name and phone) seen by Dr. Sanchez in February.
61. What was the most commonly used insurance company in March?
62. What was the highest total amount billed for a single visit in August?
63. What were the total amounts charged for each week in the year?
64. Compare the total number of visits by day of the week for the year.
65. Based on the first seven letters of the trade name, what was the most commonly prescribed drug in July?
66. For December, which group performed more total procedures: Physicians or everyone else?

67. Which employee took the most total vacation time for the year?

68. Which patient was prescribed the most number of drugs (count) in September?

✓ 69. Which 2-level ICD10 treatment procedure was most common in January – March?

70. What was the total amount charged to each insurance company for the month of November?

XQuery

71. Create the ShippingInvoice table and add the sample data. Run the sample queries from the figures and verify the results.

✓ 72. Create an XQuery that retrieves the items with a quantity of more than 20.

73. Create an XQuery that retrieves the price for any description equal to "Collar."

74. Create an XQuery that retrieves the Item Descriptions shipped where ShippingID=1 and ItemID is 15. Use the text( ) function to return just the value without the tags.

75. Write an XQuery using the "for" statement that returns all of the ItemIDs in ascending order from the first shipment. Hint: The result should be: <ItemID>15</ItemID><ItemID>32</ItemID>.

Regular Expressions

76. If you are using SQL Server, write the code to create and deploy the Visual Studio C# RegexMatch function and enable the CLR code in the Pet Store database. In all cases, create and test the RegEx expressions from the figures in the chapter.

✓ 77. Create a regular expression pattern that retrieves all Merchandise items that refer to dry food which might be written as in either order (dry…food or food…dry).

78. Create a regular expression pattern that retrieves all Merchandise items with a weight of 10 pounds. Do not include 100 pound items and do not assume there is a space after the 10.

79. Get a list of customers who have a street address that contains less than 4 digits. Hint: The number must appear at the start of the Address and look up the options for the { } repeating specification.

✓ 80. Check the breed entries to list all of the Terriers, but also check for misspellings that use only a single "r" (Terier).

81. Using the CornerMed database and RegEx, list all of the Descriptions in the ICD10 procedures that include the words (Left, Artery, and Endoscopic) in any order. Hint: Search for RegEx lookahead examples.

82. Using the CornerMed database and RegEx, list all of the patients with a last name of McCarthy; which might be spelled with or without a space between the "C"s and might have only one C.

Web Site References

http://www.jcc.com/sql.htm	Blog on SQL Standards
http://jtc1sc32.org/ http://www.wiscorp.com/SQLStandards.html	Standards documents. Free versions of some drafts.
http://www.sqlmag.com	Magazine with SQL emphasis.
http://www.sqlteam.com	SQL hints and comments.
http://www.sqlcourse.com	Online SQL notes.
http://www.w3.org/TR/xquery/	XQuery reference
http://docs.oracle.com/cd/E13214_01/wli/docs92/xref/xqlangxml.html	Oracle XML documentation
http://www.aivosto.com/vbtips/regex.html	One regex tutorial
http://msdn.microsoft.com/en-us/magazine/cc163473.aspx	Microsoft article with the CLR regex function.

Additional Reading

Gulutzan, P. and T. Pelzer, SQL-99 Complete, Really, Gilroy, CA: CMP Books, 2000. [In depth presentation of the SQL-99/SQL3 standard.]

Melton, J. and A. R. Simon. SQL 1999: Understanding Relational Language Components, 2002. San Mateo: Morgan Kaufmann Publishers, 1993. [An in-depth presentation of SQL 1999, by those who played a leading role in developing the standard.]

Appendix: SQL Syntax

SQL Commands

Alter Table

```
ALTER TABLE table
        ADD COLUMN column datatype (size)
        DROP COLUMN column
```

Commit Work

```
COMMIT WORK
```

Create Index

```
CREATE [UNIQUE] INDEX index
ON table (column1, column2, …)
WITH {PRIMARY | DISALLOW NULL | IGNORE NULL}
```

Create Table

```
CREATE TABLE table
(
        column1 datatype (size) [NOT NULL] [index1],
        column2 datatype (size) [NOT NULL] [index2],
        … ,
        CONSTRAINT pkname PRIMARY KEY (column, …),
        CONSTRAINT fkname FOREIGN KEY (column)
                REFERENCES existing_table (key_column)
                ON DELETE CASCADE
)
```

Create Trigger

```
CREATE TRIGGER triggername { BEFORE | AFTER }
        {DELETE | INSERT | UPDATE}
        ON table { FOR EACH ROW }
        { program code block}
```

Create View

```
CREATE VIEW viewname AS
SELECT …
```

Delete

```
DELETE
FROM table
WHERE condition
```

Drop Index

```
DROP INDEX index ON table
```

Drop Table

```
DROP TABLE table name
```

Drop Trigger

```
DROP TRIGGER trigger name
```

Drop View

```
DROP VIEW view name
```

Insert

```
INSERT INTO table (column1, column2, …)
VALUES (value1, value2, …)
```

Insert (copy multiple rows)

```
INSERT INTO newtable (column1, column2, …)
SELECT …
```

Grant

```
GRANT privilege
ON object
TO user | PUBLIC
```

Revoke

```
REVOKE privilege
ON object
FROM user | PUBLI
```

Privileges for Grant and Revoke

```
ALL, ALTER, DELETE, INDEX,
INSERT, SELECT, UPDATE
```

Rollback

```
ROLLBACK WORK
TO savepoint
```

SavePoint

```
SAVEPOINT savepoint
```

Select

```
SELECT DISTINCT table.column {AS alias}, …
FROM table/view
INNER JOIN table/view ON T1.ColA = T2.ColB
WHERE (condition)
GROUP BY column
HAVING (group condition)
ORDER BY table.column
{ UNION, INTERSECT, EXCEPT, … }
```

Select Into

```
SELECT column1, column2, …
INTO newtable
FROM tables
WHERE condition
```

Update

```
UPDATE table
SET column1 = value1, column2 = value2, …
WHERE condition
```

Chapter 5

Advanced Queries and Subqueries

Chapter Outline

What You Will Learn in This Chapter

- How can SQL be used to answer more complex questions?
- Why are some business questions more difficult than others?
- What common uses for subqueries?
- How do you find something that did not happen?
- How do you include rows from tables in a join even if the rows do not match?
- What are correlated subqueries?
- What tricky problems arise and how do you handle them in SQL?
- What are the SQL data definition commands?
- What SQL commands alter the data stored in tables?
- How do you know if your query is correct?

A Developer's View

Ariel: Hi Miranda. You look happy.

Miranda: I am. This query system is great. I can see how it will help the managers. Once I get the application done, they can get answers to any questions they have. They won't have to call me for answers every day. Plus, I can really see how the query system relates to data normalization. With normalization I split the tables so the database could store them properly. Now the query system helps me rejoin them to answer my questions.

Ariel: Does that mean you're finally ready to create the application?

Miranda: Close, but I'm not quite ready. Yesterday my uncle asked me a question that I don't know how to answer.

Ariel: Really, I thought you could do anything with SQL. What was the question?

Miranda: Something about customers who did not order anything last month. I tried several times to get it to work, but the answers I get just aren't right.

Ariel: It doesn't sound like a hard question.

Miranda: I know. I can get a list of customers and orders that were placed any time except last month. But every time I join the Customer table to the Order table, all I get are the customers who did place orders. I don't know how to find something that's not there.

Getting Started

SQL has several powerful capabilities, including subqueries (the ability to nest a query inside another one), and outer joins (returning all rows from one table in a join instead of ignoring unmatched data). You need to think of data and questions in terms of sets. To answer complex questions, break it into pieces and create a query to return the data set for each piece. Then combine the pieces using joins, subqueries, or set operations.

Introduction

How can SQL be used to answer more complex questions? Now that you understand the basics of the SQL SELECT statement as described in Chapter 4, it is time to study more complex questions. The basic SELECT statement you have learned is useful for returning filtered rows and columns of data and for computing subtotals.However, some business questions are more complex than those examples. For instance, how would you find items that were not sold? The database only stores things that did happen and note that when tables are joined, only the rows with matching data are returned. How can you get to the data that is not matched—that is, data in one table (Merchandise) but not in the other (SaleItem)? Also, what if you need to combine data from multiple queries? A classic example is percentages. To compute percentages within a group, you must first compute the group totals and then divide by the overall total. One of the most powerful features of the SQL SELECT command is known as a **subquery** or **nested query**.

This feature enables you to ask complex questions that entail retrieving different types of data or data from different sources.

SQL is also more than a query language. The language can be used to create tables, as well as insert, delete, and update data. It can be used to create the entire database (data definition language). SQL has powerful commands to alter the data (data manipulation language). SQL also has a couple of commands to set security conditions (data control language).

Two key points will help you learn how to use subqueries: (1) SQL was designed to work with sets of data—avoid thinking in terms of individual rows, and (2) you can split nested queries into their separate parts and deal with the parts individually. Sometimes it is helpful to write a query to answer part of a question and save it. This saved query or view can then be used in part of a second query.

The features of SQL covered in Chapter 4 are already quite powerful. Why do you need more features? Consider this common business question for Sally's Pet Store: Which merchandise items have not been sold? Think about how you might answer that question using the SQL you know to this point.

The first step might be to choose the tables: Merchandise and SaleItem appear to be likely choices. Second, select the columns as output: ItemID and Description. Third, specify a condition. Fourth, join the tables. These last two steps cause the most problems in this example. How do you specify that an item has not been sold? The big catch is that you have to be careful when examining data in the SaleItem table. Because the item has not been sold, the SaleItem table will not contain any entries for it. The SaleItem table records things that have happened. You are looking for something that has not happened.

Actually, the fourth step (joining the tables) causes even more problems. Say you wrote a query like this: SELECT ItemID, Name FROM Merchandise INNER JOIN SaleItem ON (Merchandise.ItemID = SaleItem.ItemID). As soon as you write that JOIN condition, you eliminate all the items you want to see. The JOIN clause restricts the output—just like a WHERE clause would. In this example, you told the DBMS to return only those items that are listed in both the Merchandise and SaleItem tables. But only items that have been sold are listed in the SaleItem table, so this query can never tell you anything about items that have not been sold. The following sections describe two solutions to this problem: either fix the JOIN statement so that it is not as restrictive or use a subquery.

Two-Minute Chapter

Some business questions are harder to answer than they first appear. Chapter 4 showed how to create basic SQL queries—selecting columns and rows, making basic calculations, and computing aggregations such as averages and sums. Computing subtotals using the GROUP BY statement is an important part of many queries. This foundation is used again in this chapter, but with a few twists. From a SQL perspective, four primary elements are added in this chapter: (1) subqueries, where you can embed a second SELECT statement into another one to look up different data, (2) LEFT JOINs, which keep rows of data from a table even if no values are matched on the other side of the join, (3) inequality JOINs where values can be compared using conditions beyond a simple equals sign, and (4) data manipulation language commands that enable you to INSERT, UPDATE, and DELETE data, not just retrieve it.

Just looking at the SQL capabilities, it is not always clear why these new features are needed. But some business questions can be tricky. How do you find

- Which items have not been sold?
- Which items were not sold in July 2013?
- Which cat merchandise sold for more than the average sale price of cat merchandise?
- Compute the merchandise sales by category in terms of percentage of total sales.
- List all of the customers who bought something in March and who bought something in May. (Two tests on the same data!)
- List dog merchandise with a list price greater than the sale price of the cheapest cat product.
- Has one salesperson made all of the sales on a particular day?
- Use Not Exists to list customers who have not bought anything.
- Which merchandise has a list price greater than the average sale price of merchandise within that category?
- List all the managers and their direct reports.
- Convert age ranges into categories.
- Classify payments by number of days late.
- Which employees sold merchandise from every category?
- List customers who adopted dogs and also bought cat products.

Figure 5.1

Harder questions. Even though there are few constraints on the problems, these questions are more complex. To answer many of them, we need to use subqueries or outer joins.

something that did not happen? A database table only stores things that did happen, so you need a way to find items in one list that are not in a second list. For instance, find Employees who did not make a Sale in a specific month. This question can be answered with a NOT IN subquery, or using a LEFT JOIN to connect the tables. Some business questions require separate sets of data—such as listing customers who bought items in March and June. Those lists have to be defined with separate queries and then combined—either through subqueries or as two saved views. Similarly, subqueries are useful when you need to compute percentage values—such as the percentage of total monthly sales attributed to each employee. At almost any point in SQL where you need a new value (divide by total), you can add parentheses and write a new SELECT statement to retrieve that value.

SQL contains a full set of commands to CREATE and DROP tables, indexes, and other items. It also has commands to UPDATE, INSERT, or DELETE rows of data. When working with these commands it is best to think in terms of sets of data. Using the power of the WHERE command (including subqueries) you can modify specific collections of data with one command.

When working with complex SQL commands, it is critical to build queries in pieces and test each piece along the way. The scariest part of SQL is that in most cases, a SELECT statement will return values—but you need to be sure that the query was interpreted the way you intended and the values accurately answer the business question.

Sally's Pet Store

Why are some business questions more difficult than others? Figure 5.1 shows some more business questions that Sally needs to answer to manage her business. Again, think about how you might answer these questions using the basic SQL of

Chapter 4. At first glance they do not seem too difficult. However, even the easiest question—to identify cats that sold for more than the average price—is harder than it first appears.

The common feature of these questions is that they need to be answered in multiple steps. All of these questions require an additional tool: the subquery. Actually, you can also answer multi-step questions by writing and saving the first part as a view and then using the view in another query. However, you should generally try to use subqueries so the DBMS query optimizer can use the complete query to find the most efficient solution.

Outer Joins (LEFT JOIN)

How do you find something that did not happen? One question that commonly arises in business settings is illustrated in Figure 5.2 with the question: Which merchandise has not been sold? This question is deceptive. At first glance it looks like you could just join the Merchandise table to the SaleItem table. But then what? The standard INNER JOIN statement will display only that merchandise that appears in both the Merchandise and SaleItem tables. As soon as you enter the JOIN statement, you automatically restrict your list to only that merchandise that has been sold.

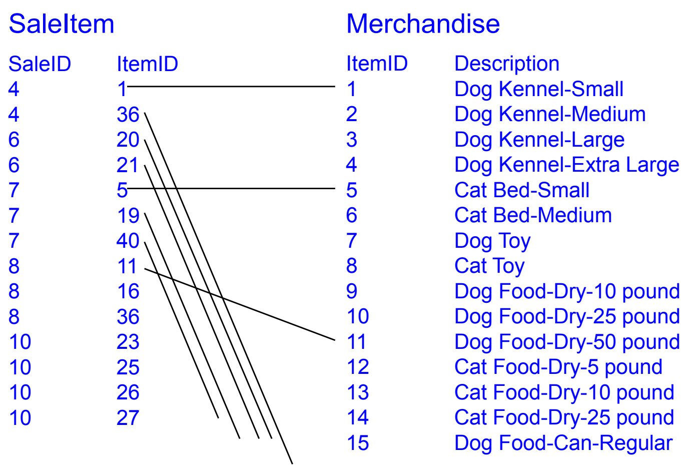
Figure 5.2

INNER JOIN is a filter. Rows that are not in both tables are ignored. Because SaleItem includes only merchandise that has been sold, INNER JOIN discards the very data that you want to see.

Which items have not been sold?

```
Try:
SELECT *
FROM Merchandise
INNER JOIN SaleItem
  ON Merchandise.ItemID = SaleItem.ItemID

But INNER JOIN is a filter that returns only rows that exist in both tables.
```



SaleItem

SaleID	ItemID
4	1
4	36
6	20
6	21
7	5
7	19
7	40
8	11
8	16
8	36
10	23
10	25
10	26
10	27

Merchandise

ItemID	Description
1	Dog Kennel-Small
2	Dog Kennel-Medium
3	Dog Kennel-Large
4	Dog Kennel-Extra Large
5	Cat Bed-Small
6	Cat Bed-Medium
7	Dog Toy
8	Cat Toy
9	Dog Food-Dry-10 pound
10	Dog Food-Dry-25 pound
11	Dog Food-Dry-50 pound
12	Cat Food-Dry-5 pound
13	Cat Food-Dry-10 pound
14	Cat Food-Dry-25 pound
15	Dog Food-Can-Regular

Which merchandise has not been sold?

```
SELECT Merchandise.ItemID, Merchandise.Description, SaleItem.
SaleID
FROM Merchandise
LEFT JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE SaleItem.SaleID Is Null;
```

ItemID	Description	SaleID
12	Cat Food-Dry-5 pound	
13	Cat Food-Dry-10 pound	

Figure 5.3

LEFT JOIN. The left outer join includes all rows from the Merchandise (left) table and any matching rows from the SaleItem table. If an item has not been sold, there will be no entry in the SaleItem table, so the corresponding entries will be NULL.

One way to solve this problem is to change the behavior of the JOIN command. SQL provides the **OUTER JOIN** specifically to include the data that would otherwise be ignored with the INNER JOIN. In particular, the OUTER JOIN describes what should happen when values in one table do not exist in the second table.

In joining two tables, you have to consider two basic situations: (1) A value might exist in the left table with no matching value in the right table, or (2) a value might exist in the right table with no matching value in the left table. Of course, it really does not matter which table is on the left or right. However, you have to be careful about not mixing them up after you list the tables.

The query in Figure 5.3 illustrates a typical **LEFT JOIN**. With a LEFT JOIN, all rows in the table on the left will be displayed in the results, regardless of what rows exist in the other table. If there is no matching value from the table on the

Figure 5.4

Partial results from the left outer join. Note the missing (Null) values for items that have not been sold. To list just a single SaleID, use GROUP BY and use the FIRST option to pick a single SaleID.

M.ItemID	Description	SA.ItemID	SaleID
1	Dog Kennel-Small	1	4
2	Dog Kennel-Medium	2	54
3	Dog Kennel-Large	3	17
4	Dog Kennel-Extra Large	4	18
5	Cat Bed-Small	5	7
6	Cat Bed-Medium	6	46
7	Dog Toy	7	64
8	Cat Toy	8	13
9	Dog Food-Dry-10 pound	9	48
10	Dog Feed-Dry-25 pound	10	60
11	Dog Food-Dry-50 pound	11	8
12	Cat Food-Dry-5 pound		
13	Cat Food-Dry-10 pound		

right, NULL values will be inserted into the output. Note how the LEFT JOIN resolves the problem of identifying items that have not been sold. Because the query will now list all Merchandise items, the rows where the SaleID is Null represent items that are not in the SaleItem table and have not been sold.

Figure 5.4 shows the sample data without the "Is Null" condition. The data has also been reduced using a GROUP BY and First statement to focus on the individual Merchandise items. Notice the two values with the missing or null values.

The **RIGHT JOIN** behaves similarly to the LEFT JOIN. The only difference is the order of the tables. If you want to use all the rows from the table on the right side, use a RIGHT JOIN. Why not just have a LEFT JOIN and simply rearrange the tables? Most of the time, that is exactly what you will do. However, if you have a query that joins several tables, it is sometimes easier to use a RIGHT JOIN instead of trying to rearrange the tables. And with visual tools such as the Microsoft Access query editor, the position of the displayed table does not have to match the SQL statement. In every case, the Left/Right applies to the way the SQL statement is written.

Another join is the full OUTER JOIN (**FULL JOIN**) that combines every row from the left table and every row from the right table. Where the rows do not match (from the ON condition), the join inserts NULL values into the appropriate columns. Many systems do not support the FULL or OUTER JOIN on both tables at the same time. If you encounter a question that requires both a left and right join, you can use a LEFT JOIN and a RIGHT JOIN against a full list of the ID values--which can be obtained using a saved UNION query.

Warning: Be careful with OUTER JOINs—particularly full joins. With two large tables that do not have much data in common, you end up with a very large result that is not very useful. Also be careful when using outer joins on more than two tables in one query. You get different results depending on the order in which you join the tables. Many times you will find it necessary to create a view with only two tables to create an outer join. You can then use that view in other queries to add more tables.

Finally, note that these examples rely on the SQL 92 syntax, which is fairly easy to read and understand. Unfortunately, you will most likely encounter some queries that use older, proprietary syntax for outer joins. Figure 5.5 shows the query using the syntax for SQL Server and Oracle. SQL Server uses *= to indi-

Figure 5.5

Older syntax for LEFT JOIN. Note the asterisk in SQL Server to indicate the LEFT side table. Note the plus-sign in Oracle and note that it is on opposite side from what you would expect.

```
SELECT *                         (SQL Server)
FROM Merchandise, SaleItem
WHERE Merchandise.ItemID *= SaleItemID.ItemID
And SaleItem.SaleID Is Null

SELECT *                         (Oracle)
FROM Merchandise, SaleItem
WHERE Merchandise.ItemID = SaleItemID.ItemID (+)
And SaleItem.SaleID Is Null
```

Find a Customer with first name of Tim, David, or Dale

```
SELECT *
FROM Customer
WHERE FirstName=N'Tim' Or FirstName=N'David' Or FirstName=N'Dale'
```

```
SELECT *
FROM Customer
WHRE FirstName IN (N'Tim', N'David', N'Dale')
```

Figure 5.6

IN function. The IN function compares a column to a set of values. The WHERE condition is true if the column/row matches any one of the entries.

cate a left join, where the asterisk can be interpreted as the "all rows" side of the join. Oracle uses a plus sign, and it confusingly puts it on the opposite side of the equals sign. Be careful when reading older queries to look for the asterisk or plus sign. The query results are quite different if you ignore these left join indicators. Fortunately, all of the major systems now accept the newer syntax, so you should convert older queries to the new syntax to improve readability.

Subqueries: IN and NOT IN

How is a subquery used for IN and NOT IN conditions? There is another way to answer the question of which items have not been sold. This new approach has considerable power and can be used for many types of questions. The main tool is the subquery, but for the problem of finding things that did not happen it is tied to a special WHERE condition known as the **IN** statement. So this section begins with a brief explanation of the IN function.function. The IN function defines a set of values. You can think of it as a shortcut way of combining several entries with an "Or" condition. For example, say you want to search for a Customer but you are not certain about his first name. You think it might be "Tim" or "David" or "Dale." As shown in Figure 5.6, you could build a query using "Or" conditions: WHERE FirstName="Tim" or FirstName="David" or FirstName="Dale". However, the figure also shows an easier way to write the query using the IN function. Simply list all possible values separated by commas and enclose them in parentheses. The IN function essentially defines a set of possible matches. It can

Figure 5.7

Subquery to find data for an IN set of values. This subquery essentially functions as a JOIN condition. Matching ItemID in the Merchandise table to the ItemID in the SaleItem table.

List Merchandise based on ItemID that has been sold.

```
SELECT * FROM Merchandise
WHERE ItemID IN (1,2,3,4,5,6,7,8,9,10,11,14,15);
```

```
SELECT *
FROM Merchandise
WHERE ItemID IN
  (SELECT ItemID FROM SaleItem);
```

be used in many situations, just be sure to match the data types with the search column. In this case, the set contains possible FirstName values.

Now consider a more relevant set of data shown in Figure 5.7, using a different question: List Merchandise where ItemID is one of 1,2,3,4,5,6,7,8,9,10,11,14,15. The list of items is a bit long, but the process is identical to that used for the names: *SELECT * FROM Merchandise WHERE ItemID IN (1,2,3,4,5,6,7,8,9,10,1 1,14,15)*. Using the raw numbers, this list is not particularly interesting. However, rewrite the query as shown in the second half of the figure. Instead of a fixed list of numbers, use a new query (SELECT ItemID FROM SaleItem) to retrieve a list of ItemID values. This subquery is embedded directly into the main query; however, note that it is surrounded by parentheses. Also, the subquery text is indented to make it easier to read. The parentheses are required, the indentation is not. This subquery performs the same role as an INNER JOIN statement. Rows from the Merchandise table will be returned only if the ItemID exists in the SaleItem table. Notice that with this formulation, only data from the top-most query (Merchandise) can be displayed. The subquery acts as a filter, but data from the subquery table cannot be displayed in the results.

Finally, as shown in Figure 5.8, it is possible to answer the original question: *List the merchandise that has not been sold*. Note that the previous version listed merchandise that was sold. That is, list the Merchandise items that are in the

Figure 5.8

NOT IN. The top-level query retrieves items from the complete list (Merchandise) and subtracts items that are in the second list (SaleItem). Leaving the results of items in the first list that are not in the second list—or things that did not happen.

List Merchandise that has not been sold.

```
SELECT *
FROM Merchandise
WHERE ItemID NOT IN
  (SELECT ItemID FROM SaleItem);
```

Merchandise

ItemID	Description
~~1~~	~~Dog Kennel-Small~~
~~2~~	~~Dog Kennel-Medium~~
~~3~~	~~Dog Kennel-Large~~
~~4~~	~~Dog Kennel-Extra Large~~
~~5~~	~~Cat Bed-Small~~
~~6~~	~~Cat Bed-Medium~~
~~7~~	~~Dog Toy~~
~~8~~	~~Cat Toy~~
~~9~~	~~Dog Food-Dry-10 pound~~
~~10~~	~~Dog Food-Dry-25 pound~~
~~11~~	~~Dog Food-Dry-50 pound~~
12	Cat Food-Dry-5 pound
13	Cat Food-Dry-10 pound
~~14~~	~~Cat Food-Dry-25 pound~~
~~15~~	~~Dog Food-Can-Regular~~

Which merchandise was not sold in July 2013?

```
SELECT *
FROM Merchandise
WHERE ItemID NOT IN
 (SELECT ItemID
  FROM SaleItem
  INNER JOIN Sale ON Sale.SaleID=SaleItem.SaleID
  WHERE SaleDate BETWEEN
     '01-JUL-2013' AND '31-JUL-2013'
  );
```

Figure 5.9

Subquery with a Date condition. Subqueries can be relatively complex. They can even be nested several levels deep. Often, subqueries can be used to write a single complex query that would need to be broken into pieces if handled differently.

SaleItem table. To answer the main question, start with the main list (Merchandise) and subtract the items that were sold (SaleItem). The process is similar to the way you would answer the question by hand if you had only paper lists. You would begin with the main Merchandise list, go through the SaleItem list and cross off all of the entries that you found. The ones that remain are the Merchandise items that never appeared on the SaleItem list so they were not sold.

When would you use the NOT IN subquery versus the LEFT JOIN? Ultimately, there is no fixed rule—use whichever method you feel is easiest to answer the question correctly. There are often multiple ways to write complex queries. Initially, the most important aspect is that you build the query correctly to answer the question. But, is one method faster to process than the other? Possibly, but ultimately that answer is up to the specific DBMS you are using. The high-end query processors automatically optimize every query, sometimes rewriting it to make it more efficient. On the other hand, if you work with a lower-end DBMS, you might have to rewrite some queries yourself to make them faster—particularly if the query needs to be run multiple times on large datasets.

Consider one more example to point out some other difficulties in creating queries that search for things not in the database. Which merchandise was not sold in July 2013? The change is to add the date condition. First, look at the subquery ap-

Figure 5.10

LEFT JOIN. This query might not run, and if it does, it might not return the correct results. The problem is that the question requires filtering the data rows in the SaleItem table first and then performing the LEFT JOIN.

Which merchandise was not sold in July 2013?

```
SELECT Merchandise.*
FROM Sale
INNER JOIN (Merchandise
  LEFT JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID)
  ON Sale.SaleID = SaleItem.SaleID
WHERE SaleDate BETWEEN '01-JUL-2013' AND '31-JUL-2013';
```

Which merchandise was not sold in July 2013?

```
CREATE VIEW JulyItems AS
SELECT Sale.SaleID, ItemID
FROM Sale
INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
WHERE SaleDate BETWEEN '01-JUL-2013' AND '31-JUL-2013';
```

```
SELECT Merchandise.*
FROM Merchandise
LEFT JOIN JulyItems ON Merchandise.ItemID=JulyItems.ItemID
WHERE JulyItems.Sale Is Null;
```

Figure 5.11

Saved View. To ensure the proper sequencing, save the view that filters the list of sale items to July.

proach. Figure 5.9 shows how to answer the question with a subquery. Essentially, the approach is the same as before—with a more complex subquery. Simply add the Sale table to the subquery and add the date condition. The overall structure is the same. Running the query results in 27 rows or items that were not sold in July.

Now consider writing the same query using the LEFT JOIN approach. As shown in Figure 5.10, try building the query directly. Note that it requires three tables: Merchandise, Sale, and SaleItem. Sale and SaleItem are connected with an INNER JOIN and Merchandise with a LEFT JOIN. Because of these links, it is likely that this query will not run. Even if it does return results, they might not be the correct results. The problem is that the question requires that the data be extracted in a specific order—and SQL does not guarantee that processing is handled in a specific sequence. To work correctly, the query must first filter the rows in the Sale+SaleItem tables to just sales that took place in July. This result must then use an outer join with the Merchandise table.

If you want (or need) to use LEFT JOIN to answer the question, you should build the query in two steps. As shown in Figure 5.11, in step 1, create and save a view that retrieves the ItemID for merchandise sold in July. In step 2, LEFT JOIN the Merchandise table to the new view. The result should be the same 27 items found using the subquery. The key to this query is that the view is created to ensure that the rows for sales in July are extracted first and then the LEFT JOIN is applied to the Merchandise table.

Subqueries

What are the common uses for subqueries? The most difficult step in creating a query is determining the overall structure. Chapter 4 shows you how to use the four big questions to determine the structure of simple queries. But you need to recognize when subqueries are needed. If you fail to use a subquery, you are likely to end up with bad results, and waste considerable time in the process. This section presents the most common situations that require the use of subqueries. The main situations are: (1) Calculations or lookup comparisons, (2) matching sets of data, (3) existence checks, and (4) finding items that are not in a list. The last example was covered in the previous section.

Which cat merchandise sold for more than the average sale price of cat merchandise?

```
SELECT Merchandise.ItemID, Merchandise.Description, Merchandise.
Category, SaleItem.SalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE Merchandise.Category=N'Cat' AND SaleItem.SalePrice > 9;
```

```
SELECT Merchandise.ItemID, Merchandise.Description, Merchandise.
Category, SaleItem.SalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE Merchandise.Category=N'Cat' AND SaleItem.SalePrice >
  (SELECT Avg(SaleItem.SalePrice) AS AvgOfSalePrice
  FROM Merchandise
  INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
  WHERE Merchandise.Category=N'Cat')
```

Figure 5.12

Subqueries for calculation. If you know the average price is 9, the query is straightforward. If you do not know the average price, you can use a subquery to compute it. The subquery is always written inside a separate set of parentheses. The subquery in parentheses replaces the 9 in the original query).

Calculations or Simple Lookup

Perhaps the easiest way to see the value of a subquery is to consider the relatively simple question: Which cat merchandise sold for more than the average price of cat merchandise? If you already know the average sale price of cat merchandise (say, $9), the query is easy, as shown in the top half of Figure 5.12.

Chapter 4 showed that it is also straightforward to write a query to compute the average price of cat merchandise. If you do not know anything about subqueries, you could write the average value on a piece of paper and then enter it into the main query in place of the 9. However, with a subquery, you can go one step further: The result (average) from the query can be transferred directly to the original query. Simply replace the value ($9) with the complete SELECT AVG query as shown in the lower half of Figure 5.12. In fact, anytime you want to insert a value or comparison, you can use a subquery instead. You can even go to several levels, so a subquery can contain another subquery and so on. The DBMS generally evaluates the innermost query first and passes the results back to the higher level.

Calculations for Percentages

Typically, subqueries for calculations arise in WHERE clauses similar to the prior example when you need to make a comparison. You can also add subqueries to the SELECT statement to retrieve a value for a calculation. For instance, you might issue a subquery to retrieve a tax rate that is multiplied times a total.

Another interesting business problem is the need to compute percentages. Figure 5.13 shows a typical question to compute the percentage of merchandise sales by category. The first step is to compute the total sales by category—which is a straightforward question from Chapter 4. That query contains the subtotal calcula-

Compute the merchandise sales by category in terms of percentage of total sales.

```
CREATE VIEW CategorySubtotals AS
SELECT Merchandise.Category, Sum([Quantity]*[SalePrice]) AS [Value]
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
GROUP BY Merchandise.Category;
```

```
SELECT CategorySubtotals.Category, CategorySubtotals.Value,
[Value] /
  (SELECT Sum(Value) FROM CategorySubtotals) AS Percentage
FROM CategorySubtotals;
```

Category	Value	Percentage
Bird	$631.50	7.45063292035315E-02
Cat	$1,293.30	0.152587546411603
Dog	$4,863.49	0.573809638983505
Fish	$1,597.50	0.188478006179955
Mammal	$90.00	1.06184792214059E-02

Figure 5.13

To obtain percentages, first compute the group subtotals and save the view. Select the values from the saved view and use a subquery in the SELECT clause to divide by the total. Ultimately, format the new percentage column to make it readable.

tion: Sum([Quantity]*[SalePrice]). To compute the percentages, add a new column that uses the same subtotal and divides by the overall total. The trick is that the overall total is computed using a subquery: SELECT Sum([Quantity]*[SalePrice]) FROM SaleItem. So the entire calculation becomes:

```
SELECT ... Sum([Quantity]*[SalePrice]) /
     (SELECT Sum([Quantity]*[SalePrice] FROM SaleItem)
GROUP BY Category...
```

Of course, most problems are even more complex and trying to jam everything into one query can lead to mistakes. So you might want to first create a saved query that computes totals by category and use a second query to compute percentages, which makes it easier to check the results. Once the subtotals have been computed and saved, the small addition to compute percentages is almost always the same.

You should realize by now that there are other ways to answer the original question. For example, keep the first view that computes the subtotals. Create a second view to compute the overall total. This second view will contain only one row as a result. Now build a third query that joins these two results. Simply do not enter a JOIN condition—let the DBMS build a cross-join so that the overall total is matched to every row of the first query. Figure 5.14 shows the new view and the query that performs the cross join and division. The results should match those with the subquery method.

Two useful practices you should follow when building subqueries are to indent the subquery to make it stand out so humans can read it and to test the subquery before inserting it into the main query. Fortunately, most modern database systems make it easy to create a subquery and then cut and paste the SQL into the main

Compute the merchandise sales by category in terms of percentage of total sales.

```
CREATE VIEW TotalItemSales AS
SELECT Sum(Value) AS MainTotal
FROM CategorySubtotals;
```

```
SELECT Category, Value, Value/MainTotal AS Percentage
FROM CategorySubtotals, TotalItemSales;
```

Figure 5.14

Percentages using a cross join. Create a view to compute the total. A third query uses a cross join to connect this single value to every row in the subtotal query and then divide to get the percentage.

query. Similarly, if you have problems getting a complex query to work, cut out the inner subqueries and test them separately. And always remember to enclose the subquery in parentheses.

The main drawback to subqueries is that they are difficult to read and understand. It is easy to make mistakes and it is difficult to read complex queries created by other developers. You should always document your work when creating complex queries. Whenever possible, use the SQL comment characters (--) to add notes to the query to explain its purpose and how it is supposed to work. Sometimes, it is better to store complex subqueries as views and use a final query to retrieve data from the carefully-named views.

The other trick you will quickly learn is that QBE grids are not very useful when designing subqueries. You almost always need to work with plain SQL statements. If you want to save some typing, you can use QBE to write the join statements, but eventually, you need to copy and paste the SQL text.

Subqueries and Sets of Data

A key to understanding SQL is to focus on sets of data. Complex queries generally can be broken down into multiple pieces, where each piece of the question refers to a set of data. Then you have to figure out how to combine those sets to answer the business question. So far you have seen two ways to combine sets of data: (1) By saving each piece and using a JOIN statement, or (2) Using a subquery, typically with an IN function. In effect, these two methods work the same way. Which one you choose depends on which is easiest or fastest to use. Keep in mind that subqueries enable you to put the entire SQL into a single query, which reduces the risk of someone accidentally deleting a supporting saved view—because no one knew what it was for.

To understand the issue of sets of data, think about an apparently simple question: List all of the customers who bought something in March and in May. As shown in Figure 5.15, a beginner might try to answer the question by creating a simple query with the WHERE clause: SaleDate Between 01-Mar And 31-Mar AND SaleDate Between 01-May and 31-May. What is wrong with this approach? Try it. The query will run, but you will not get any matches. Why not? Because the clause is asking the DBMS to return rows where the SaleDate is in March and in May, at the same time! It is not possible for a date to be in two months at the same time.

List all of the customers who bought something in March and who bought something in May.

```
SELECT Customer.CustomerID, Customer.Phone, Customer.
LastName, Sale.SaleDate
FROM Customer
INNER JOIN Sale ON Customer.CustomerID = Sale.CustomerID
WHERE Sale.SaleDate Between ’01-MAR-2013’ And ’31-MAR-2013’
  AND Sale.SaleDate Between ’01-MAY-2013’ And ’31-MAY-2013’;
```

Figure 5.15

The wrong approach. Why does this query always return no rows? Because it is checking the date on each row to see if it falls in March AND May. No date can be in two months at the same time.

The answer to the question lies in realizing that you need to get two separate lists of people: those who bought something in March and those who bought something in May. Then you combine the lists to identify the people in both sets. You can answer this question with a subquery, or you can create two separate views and join them. The subquery illustrates the set operations.

Figure 5.16 shows the subquery approach. The outermost (top) query retrieves customers who bought something in March, and the subquery retrieves ID numbers for customers who bought something in May. Either month could be tested first, but it is critical to recognize that you need two separate queries to create the two separate WHERE clauses. The IN operator performs the matching so that the final query displays only those customers who fall in both sets of data.

Figure 5.17 shows how to answer the same query with a JOIN statement on saved views. The views are used to retrieve the desired sets, and they highlight that the sets are separate. The final query uses the JOIN command to retrieve only the values that exist in both of the saved views (March and May).

Both approaches (subquery and saved views) provide the same answer and you generally get to choose which approach you want to use. The drawback to saving views is that you end up with a huge collection of views, and no one remembers

Figure 5.16

Combining two separate lists. The question requires you to create two separate lists and then compare the matching values. This query uses the IN statement to find the customers that appear in both lists.

List all of the customers who bought something in March and who bought something in May.

```
SELECT Customer.LastName, Customer.FirstName
FROM Customer INNER JOIN Sale ON Customer.CustomerID =
Sale.CustomerID
WHERE (SaleDate Between ’01-MAR-2013’ And ‘31-MAR-2013’)
AND Customer.CustomerID IN
   (SELECT CustomerID
    FROM Sale
    WHERE (SaleDate Between ‘01-MAY-2013’ And ’31-MAY-2013’) );
```

List all of the customers who bought something in March and who bought something in May. (Saved views.)

```
CREATE VIEW MarchCustomers AS
SELECT CustomerID
FROM Sale
WHERE (SaleDate Between '01-MAR-2013' And '31-MAR-2013');

CREATE VIEW MayCustomers AS
SELECT CustomerID
FROM Sale
WHERE (SaleDate Between '01-MAY-2013' And '31-MAY-2013');

SELECT Customer.LastName, Customer.FirstName
FROM Customer
INNER JOIN MarchCustomers ON Customer.
CustomerID=MarchCustomers.CustomerID
INNER JOIN MayCustomers ON MarchCustomers.
CustomerID=MayCustomers.CustomerID;
```

Figure 5.17

Combining two separate lists with JOIN. You can save separate lists as views and use the JOIN command to retrieve only the values that match.

which views depend on other views. Some administrator could accidentally delete a view that is required by another query. On the other hand, the views could be re-used in multiple queries, which might save a developer time on a different project. The bottom line is that you need to know how to write the queries both ways, and choose the method that is best in each situation.

Subquery with ANY, ALL, and EXISTS

The **ANY** and **ALL** operators combine comparison of numbers with subsets. In the previous sections, the IN operator compared a value to a list of items in a set: however, the comparison was based on equality. The test item had to exactly match an entry in the list. The ANY and ALL operators work with a less than (<) or greater than (>) operator and compare the test value to a list of values.

Figure 5.18 illustrates the use of the ANY query. It is hard to find a solid business example that needs the ANY operator. In the example, it would be just as easy to use the subquery to find the minimum value (MIN function) in the list and then do the comparison. However, sometimes it is clearer to use the ANY operator.

The ALL operator behaves similarly, but the test value must be greater than all of the values in the list. In other words, the test value must exceed the largest value in the list. Hence, the ALL operator is much more restrictive.

The ALL operator can be a powerful tool—particularly when used with an equals (=) comparison. For instance, you might want to test whether one salesperson made all of the sales on a particular day. Figure 5.19 shows that the WHERE clause contains the statement: EmployeeID = ALL (SELECT EmployeeID FROM Sale WHERE SaleDate = '28-MAR'). The subquery returns a list of IDs for all employees who sold something on that date. The "= ALL" clause checks to see if all of the values are the same and match a single employee. This query is some-

List dog merchandise with a list price greater than the sale price of the cheapest cat product.

```
SELECT Merchandise.ItemID, Merchandise.Description,
Merchandise.Category, Merchandise.ListPrice
FROM Merchandise
WHERE Category=N'Dog'
AND ListPrice > ANY
  (SELECT SalePrice
   FROM Merchandise
   INNER JOIN SaleItem ON Merchandise.ItemID=SaleItem.ItemID
   WHERE Merchandise.Category=N'Cat')
;
```

Figure 5.18

Subquery with ANY and ALL. The example identifies any animal that sold for more than any of the prices of cats. Effectively, it returns values greater than the smallest entry in the subquery list.

what contrived, but it can be useful when you need to find a specific answer. The alternative in this situation is to count the number of sales by each employee on the specified date and visually check to see if there is more than one value. But, sometime you might want to find the exact answer using the ANY query.

Sometimes it is difficult to control the details returned from a subquery. Perhaps the data exists in a table created by someone else, such as a system table. In these cases, only the WHERE clause matters. Does the query return any rows that match the conditions? The **EXISTS** key word handles these situations. It is true if the subquery returns any rows of data—otherwise it is false. The specific columns returned are irrelevant. Figure 5.20 shows a simple example. In actual practice, the example would be better written with a JOIN statement, but it does illustrate how the EXISTS term works. The EXISTS term is useful when you need to see if rows are retrieved in a subquery but you do not want to match the actual values.

Figure 5.19

Subquery with All and equality test. The subquery returns a list of EmployeeID values who made sales on the specified date. The "= ALL" test checks to see if they are all the same value and returns the matching employee.

Has one salesperson made all of the sales on a particular day (Mar 28)?

```
SELECT Employee.EmployeeID, Employee.LastName
FROM Employee
WHERE EmployeeID = ALL
   (SELECT EmployeeID
    FROM Sale
    WHERE SaleDate = '28-MAR-2013')
;
```

Use Not Exists to list customers who have not bought anything.

```
SELECT Customer.CustomerID, Customer.Phone, Customer.
LastName
FROM Customer
WHERE NOT EXISTS
  (SELECT SaleID, SaleDate
   FROM Sale WHERE Sale.CustomerID=Customer.CustomerID);
```

Figure 5.20

Subquery with Exists. If the only thing that matters is the WHERE clause, you can use the EXISTS phrase to test if rows are returned or not. It is also useful when the details of the subquery are difficult to change.

Correlated Subqueries

What are correlated subqueries? Recall the example in Figure 5.12 that asked: Which cat merchandise sold for more than the average sale price of cat merchandise? This example used a subquery to first find the average sale price of cat merchandise and then examined all sales of cat merchandises to display the ones that had higher prices. It is a reasonable business question to extend this idea to other categories of animals. Managers would like to identify all merchandise that was sold for a price greater than the average price of other merchandise within their respective categories (dog merchandise compared to other dog merchandise, fish compared to fish, and so on).

As shown in Figure 5.21, building this query is tricky. The merchandise category in the subquery has to match that in the outer query. This task is accomplished by setting the categories equal to each other. But, the Merchandise table is used in both queries, so the condition can only be written by assigning aliases to the Merchandise table in both queries. Here, it is renamed as Merchandise1 and

Figure 5.21

Correlated subquery. The condition in the subquery depends on values in the outermost query. In some query systems, this query could run slowly if large tables are involved.

Which merchandise has a list price greater than the average sale price of merchandise within that category?

```
SELECT Merchandise1.ItemID, Merchandise1.Description,
Merchandise1.Category, Merchandise1.ListPrice
FROM Merchandise AS Merchandise1
WHERE Merchandise1.ListPrice>
(
SELECT Avg(SaleItem.SalePrice) AS AvgOfSalePrice
FROM Merchandise As Merchandise2 INNER JOIN SaleItem ON
Merchandise2.ItemID = SaleItem.ItemID
WHERE Merchandise2.Category=Merchandise1.Category
);
```

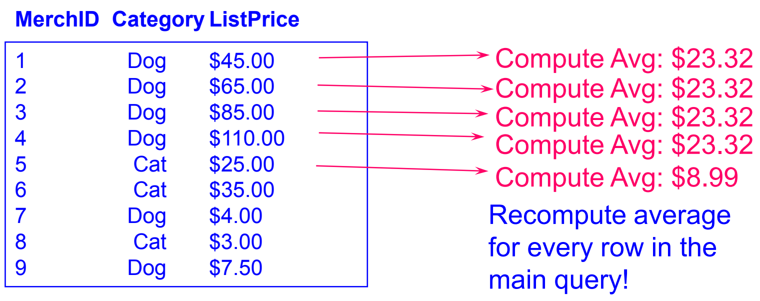
Merchandise2, but any distinct names would work. This type of query is called a **correlated subquery**, because the subquery refers to data rows in the main query.

The query in Figure 5.21 will run. However, it might be inefficient. Performance depends on the query optimizer, but systems might have problems computing this query for large sets of data. Even on a fast computer, queries of this type have been known to run for several days without finishing. If the query is run as written, the calculation in the subquery must be recomputed for each entry in the main table. The problem is illustrated in Figure 5.22. Consider an inefficient DBMS that starts at the top row of the Merchandise table. When it sees the category is Dog, it computes the average sale price of dog merchandise ($23.32). Then it moves to the next row and computes the average sale price for dogs again. In the worst case, the DBMS recomputes the average for every single row in the Merchandise table. Recomputing the average sale price for every single row in the main query is time-consuming. To compute an average, the DBMS must go through every row in the SaleItem table that has the same category of animal. Consider a relatively small query of 100,000 rows and five categories of animals. On average, there are 20,000 rows per category. To recompute the average each time, the DBMS will have to retrieve 100,000 * 20,000 or 2,000,000,000 rows!

Unfortunately, you cannot just tell the manager that it is impossible to answer this important business question. Is there an efficient way to answer this question? Some query processors can automatically cache the averages. In other cases, you will have to do it yourself. The answer illustrates the power of SQL and highlights the importance of thinking about the problem before you try to write a query. The problem with the correlated subquery lies in the fact that it has to continually recompute the average for each category. Think about how you might solve this problem by hand. You would first make a table that listed the average for each category and then simply look up the appropriate value when you needed it. As shown in Figure 5.23, the same approach can be used with SQL. Just create the query for the averages using GROUP BY and save it. Then join it to the Merchandise table to do the comparison.

Figure 5.22

Potential problem with correlated subquery. The average is recomputed for every row in the main query. Every time the DBMS sees a dog product, it computes the average to be $23.32. It is inefficient and slow to force the machine to recalculate the average each time.



MerchID	Category	ListPrice
1	Dog	$45.00
2	Dog	$65.00
3	Dog	$85.00
4	Dog	$110.00
5	Cat	$25.00
6	Cat	$35.00
7	Dog	$4.00
8	Cat	$3.00
9	Dog	$7.50

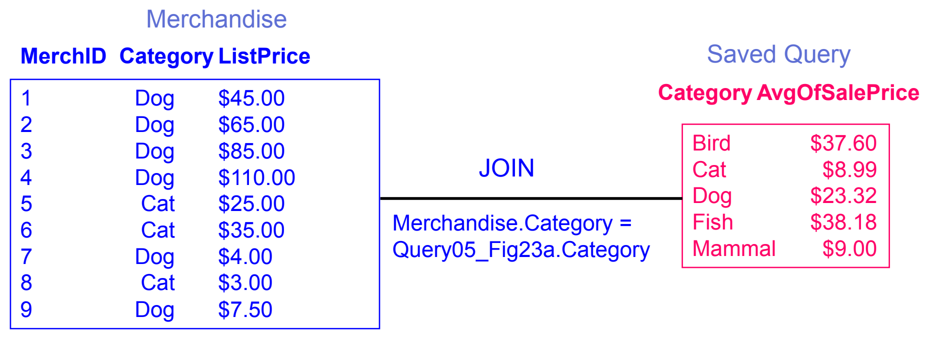


Figure 5.23

More efficient solution. Create and save a query to compute the averages using GROUP BY Category. Then join the query to the Merchandise table to do the comparison.

Today, you probably do not have to worry too much about the performance of correlated subquries. The high-end DBMSs have good query optimizers that can recognize the problem and automatically find the solution to compute the values quickly and store them in a cache. However, some queries still require hand tuning. Also, you need to remember to look for different ways to approach queries. The solution in Figure 5.23 is much easier to read and verify that the answer is correct.

More Features and Tricks with SQL SELECT

What tricky problems arise and how do you handle them in SQL? As you may have noticed, the SQL SELECT command is powerful and has plenty of options. There are even more features and tricks that you should know about. Business questions can be difficult to answer. It helps to study different examples to gain a wider perspective on the problems and solutions you will encounter. One of the first big questions you will face is the need to combine rows from different tables. You also need to know how to handle several other complications, such as joining tables with multiple columns or inequality joins.

UNION, INTERSECT, EXCEPT

Codd originally conceived of tables as sets of data. The basic filtering aspects of the SELECT command perform some operations on these sets, but it is sometimes nice to be able to use more traditional set operators. Up to this point, the tables you have encountered have contained unique columns of data. The JOIN command links tables together so that a query can display and compare different columns of data from tables. Occasionally you will encounter a different type of problem where you need to combine rows of data from similar tables. The set operations, such as the **UNION** operator are designed to accomplish these tasks.

As an example, assume you work for a company that has offices in Los Angeles and New York. Each office maintains its own database. Each office has an Employee file that contains standard data about its employees. The offices are linked by a network, so you have access to both tables (call them EmployeeEast and EmployeeWest). But the corporate managers often want to search the entire Em-

```
SELECT EID, Name, Phone, Salary, 'East' As Office
FROM EmployeeEast
UNION
SELECT EID, Name, Phone, Salary, 'West' As Office
FROM EmployeeWest;
```

EID	Name	Phone	Salary	Office
352	Jones	3352	45,000	East
876	Inez	8736	47,000	East
372	Stoiko	7632	38,000	East
890	Smythe	9803	62,000	West
631	Kim	7736	73,000	West

Figure 5.24

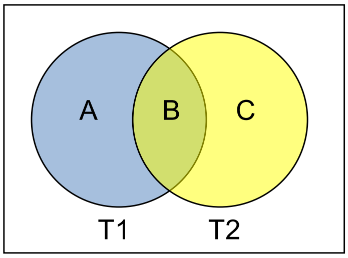
The UNION operator combines rows of data from two SELECT statements. The columns in both SELECT lines must match. The query is usually saved and used when managers need to search across both tables. Note the use of a new, constant column (Office) to track the source of the data.

ployee file—for example, to determine total employee salaries of the marketing department. One solution might be to run their basic query twice (once on each table) and then combine the results by hand.

As shown in Figure 5.24, the easier solution is to use the UNION operator to create a new query that combines the data from the two tables. All searches and operations performed on this new query will treat the two tables as one large table. By combining the tables with a view, each office can make changes to the original data on its system. Whenever managers need to search across the entire company,

Figure 5.25

Operators for combining rows from two tables. UNION selects all of the rows. INTERSECT retrieves only the rows that are in both tables. EXCEPT retrieves rows that exist in only one table.



T1 UNION T2	A + B + C
T1 INTERSECT T2	B
T1 EXCEPT T2	A

they use the saved query, which automatically examines the data from current versions of both tables.

The most important concept to remember when creating a UNION is that the data from both tables must match (e.g., EID to EID, Name to Name). Another useful trick is to insert a constant value in the SELECT statement. In this example the constant keeps track of which table held the original data. This value can also be used to balance out a SELECT statement if one of the queries will produce a column that is not available in the other query. To make sure both queries return the same number of columns, just insert a constant value in the query that does not contain the desired column. Make sure that it contains the same type of data that is stored in the other query (domains must match).

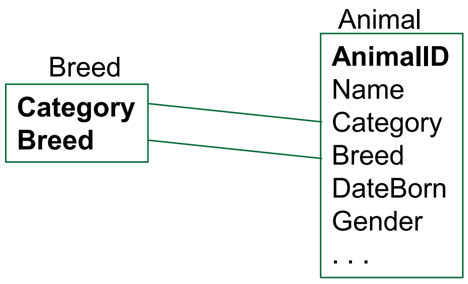
The UNION command combines matching rows of data from two tables. The basic version of the command automatically eliminates duplicate rows of data. If you want to keep all the rows—even the duplications, use the command UNION ALL. Two other options for combining rows are **EXCEPT** and **INTERSECT**. Figure 5.25 shows the difference between the three commands. They all apply to sets of rows and the Venn diagram shows that the tables might have some data in common (area B). The UNION operator returns all the rows that appear in either one of the tables, but rows appearing in both tables are only listed once. The INTERSECT operator returns the rows that appear in both tables (area B). The EXCEPT operator returns only rows that appear in the first table (area A). Notice that the result of the EXCEPT operator depends on which table is listed first. Microsoft Access supports only the UNION command. SQL Server (and other DBMSs) support all three. These set operators are another way to handle complex business questions, similar to the NOT IN problem of finding things in one set that are not in the second set. Just remember that there are often many ways to create a query.

Multiple JOIN Columns

Sometimes you will need to join tables based on data in more than one column. In the Pet Store example, each animal belongs to some category (Cat, Dog, Fish,

Figure 5.26

Multiple JOIN columns. The values in the tables are connected only when both the category and the breed match.



```
SELECT *
FROM Breed INNER JOIN Animal
ON Breed.Category = Animal.Category
AND Breed.Breed = Animal.Breed
```

```
SELECT Employee.EmployeeID, Employee.LastName, Employee.
ManagerID, E2.LastName
FROM Employee INNER JOIN Employee AS E2
ON Employee.ManagerID = E2.EmployeeID
```

EID	Name	Manager	Name
1	Reeves	11	Smith
2	Gibson	1	Reeves
3	Reasoner	1	Reeves

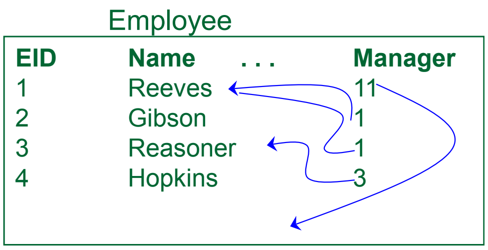


Figure 5.27

Reflexive JOIN to connect Employee table with itself. A manager is also an employee. Use a second copy of the Employee table (renamed to E2) to get the manager's name.

etc.). Each category of animal has different breeds. For example, a Cat might be a Manx, Maine Coon, or Persian; a Dog might be a Retriever, Labrador, or St. Bernard. A portion of the class diagram is reproduced in Figure 5.26. Notice the two lines connecting the Breed and Animal tables. This relationship ensures that only breeds listed in the Breed table can be entered for each type of Animal. A real store might want to include additional features in the Breed table (such as registration organization, breed description, or breed characteristics). The key point is that the tables must be connected by both the Category and the Breed.

In Microsoft Access QBE, the JOIN can be created by marking both columns and simultaneously dragging the two columns to the Animal table, but it is often easier to edit in SQL. The syntax for the SQL JOIN command is given in Figure 5.26. Simply expand the ON statement by listing both column connections. In this case, you want both sets of columns to be equal at the same time, so the statements are connected with an AND.

Reflexive Join

A **reflexive join** or **self-join** means simply that a table is joined to itself. One column in the table is used to match values in a second column in the same table. A common business example arises with an Employee table as illustrated in Figure 5.27. Employees typically have one manager. Hence the manager's ID can be stored in the row corresponding to each employee. The table would be Employee(EID, Name, Phone, . . ., Manager). The interesting feature is that a manager is also an employee, so the Manager column actually contains a value for EID. To get the corresponding name of the manager, you need to join the Employee table to itself.

List all the managers and their direct reports.

```
WITH DirectReports(EmployeeID, LastName, ManagerID, Title, Level)
AS
(
            --Root/anchor member (find employee with no manager)
            SELECT EmployeeID, LastName, ManagerID, Title, 0 As Level
            FROM Employee WHERE ManagerID=0       -- starting level
            UNION ALL
            -- Recursive members
            SELECT Employee.EmployeeID, Employee.LastName,
                        Employee.ManagerID, Employee.Title, Level +1
            FROM Employee INNER JOIN DirectReports
            ON Employee.ManagerID = DirectReports.EmployeeID
)
-- Now exectue the common table expression
SELECT ManagerID, EmployeeID, LastName, Title, Level
FROM DirectReports
ORDER BY Level, ManagerID, LastName
```

ManagerID	EmployeeID	LastName	Title	Level
0	11	Smith	Owner	0
11	1	Reeves	Manager	1
1	2	Gibson	Manager	2
1	3	Reasoner	Manager	2
2	6	Eaton	Animal Friend	3
2	7	Farris	Animal Friend	3
2	5	James	Animal Friend	3
2	9	O'Connor	Animal Friend	3
2	10	Shields	Animal Friend	3
3	8	Carpenter	Worker	3
3	4	Hopkins	Worker	3

Figure 5.28

Recursive query. The employee-manager relationship is a classic recursive example. The recursive query requires three steps: (1) Define the root level, (2) Define the recursion member that links to the higher level, and (3) Run the SELECT statement to execute the expression and sort the results.

The only trick with this operation is that you have to be careful with the ON condition. For instance, the following condition does not make sense: ON Employee.Manager = Employee.EID. The query would try to return employees who were their own managers, which is not likely to be what you wanted. Instead, you must use two instances of the Employee table and use an alias (say, E2) to rename the second copy. Then the correct ON condition becomes ON Employee.Manager = E2.EID. The key to self-joins is to make sure that the columns contain the same type of data and to create an alias for the second copy of the table.

SQL 1999 provides an even more powerful feature related to reflexive joins. Consider the employee example where you want to list all of the people who work for someone—not just the direct reports, but also the people who work for them, and the people who work for that group, and so on down the employee hierarchy tree. The standard provides the WITH RECURSIVE command that has several options to search a data tree. Consider the pet store case with the partial Employee table: Employee(EmployeeID, LastName, Title, ManagerID). You want to start at the top with the CEO/owner and list all of the employees who report directly to a manager. For example, EmployeeID 1 (Reeves) is the only person who reports directly to Sally (EmployeeID=11), but two people (EmployeeID 2 and 3) report directly to Reeves. The actual syntax can be slightly different across systems. See the Workbooks for examples. Figure 5.28 shows the syntax used by SQL Server. The main difference with the standard is that the standard uses WITH RECURSIVE instead of just the WITH keyword. The main step is to define the common table expression to handle the recursion. You give a unique name (e.g., DirectReports) to the new expression and specify the columns that will be retrieved. The three main steps are: (1) Define the root starting point for the tree with a SELECT statement, (2) Define the recursive members with a second SELECT statement that links to the level above, and (3) Write the final SELECT statement to execute the recursive table and sort or group the results. In the example, root level is defined by choosing the owner who does not report to anyone (ManagerID=0). You might need to examine the data to know how to define the root level—it might be set by title, or by a Null value in some column. The second step is the one that does most of the work. You retrieve data from the Employee table, but the JOIN statement is the key. Notice that you join the Employee.ManagerID column to the higher-level DirectReports.EmployeeID table. The DirectReports table represents the parent level entry, and this SELECT statement will always have a similar JOIN condition. The third step is the easiest, because now you can treat the DirectReports entity as just another table. Open the pet store's Employee table and work through the results given here to see how the organization structure chart is created.

The Level column is also a useful trick. You define it with the root-level SELECT statement, and increment it with the recursive SELECT. It provides an easy way to specify the distance from the root. Picture the organizational chart with Smith at the top, followed by Reeves at Level 1, and Gibson and Reasoner at Level 2 because both report to Reeves. The recursive query is a powerful statement. Without it, you need to write substantial code to accomplish the same task.

Note that many lower-end systems (such as Microsoft Access) do not support recursive joins. In these cases, you will have to write programming code to iterate through each employee to build the tree. Also be cautious when building recursive queries—it is possibly to accidentally create an infinite loop. You might want to set time limits on queries when testing recursive designs.

CASE Function

SQL 92 added the **CASE** function to simplify certain types of queries. However, many database systems have not yet implemented all the features of SQL 92. The CASE function evaluates a set of conditions and returns a single value. Similar to the Oracle decode function, the conditions can be simple (R=1) or complex.

Perhaps the managers want to classify the animals in Sally's Pet Store based on their age. Figure 5.29 shows the SQL statement that would create four categories based on different ages. Note the use of date arithmetic using today's date—Date( )—and DateBorn. Whenever this query is executed, it will use the current day to assign each animal to the appropriate category. Of course, the next logical step is to run a GROUP BY query against this view to count the number of animals falling within each age category.

Convert age ranges into categories.

```
Select AnimalID,
         CASE
                   WHEN Date()-DateBorn < 90 Then ‘Baby’
                   WHEN Date()-DateBorn >= 90
                     AND Date()-DateBorn < 270 Then ‘Young’
                   WHEN Date()-DateBorn >= 270
                     AND Date()-DateBorn < 365 Then ‘Grown’
                   ELSE ‘Experienced’
         END
FROM Animal;
```

Figure 5.29

CASE function to convert DateBorn into age categories. Note the use of date arithmetic to generate descriptions that are always current.

Inequality Joins

A JOIN statement is actually just a condition. Most problems are straightforward and use a simple equality condition or **equi-join**. For example, the following statement joins the Customer and Order tables: FROM Customer INNER JOIN Order ON (Customer.CustomerID = Order.CustomerID).

SQL supports complex conditions including **inequality joins**, where the comparison is made with inequality operators (less than, greater than) instead of an equals sign. The generic name for any inequality or equality join is a theta join.

This type of join can be useful in some tricky situations. For example, consider a common business problem. You have a table for AccountsReceivable(TransactionID, CustomerID, Amount, DateDue). Managers would like to categorize the customer accounts and determine how many transactions are past due by 30, 90, and 120 or more days. This query can be built in a couple of ways. For instance, you could write three separate queries, or you could build a complex

Figure 5.30

Inequality join. Managers want to classify the AccountsReceivable (AR) data into three categories of overdue payments. First, store the business rules/categories in a new table. Then join the table to the AR data through inequality joins.

Classify payments by number of days late.

AR(TransactionID, CustomerID, Amount, DateDue)
LateCategory(Category, MinDays, MaxDays, Charge, …)

Month	30	90	3%
Quarter	90	120	5%
Overdue	120	9999	10%

```
SELECT *
FROM AR INNER JOIN LateCategory
ON ((Date( ) – AR.DateDue) >= LateCategory.MinDays)
AND ((Date( ) – AR.DateDue) < LateCategory.MaxDays)
```

CASE statement. However, what happens if managers decide to change the business rules or add a new category? Then someone has to find your three queries and modify them. A more useful trick is to create a new table to hold the business rules or categories. In the example shown in Figure 5.30, create the table LateCategory(Category, MinDays, MaxDays, Charge). This table defines the late categories based on the number of days past due. Now use inequality conditions to join the two tables. First, compute the number of days late using the current date (Date( ) – AR.DateDue). Finally, compare the number of days late to minimum and maximum values specified in the LateCategory table.

The ultimate value of this approach is that the business rules are now stored in a simple table (LateCategory). If managers want to change the conditions or add new criteria, they simply alter the data in the table. You can even build a form that makes it easy for managers to see the rules and quickly make the needed changes. With any other approach, a programmer needs to rewrite the code for the queries.

Exists and Crosstabs

Some queries need the EXISTS condition. Consider the business question: Which employees have sold merchandise in every category? The word *every* is the key here. Think about how you would answer that question if you did not have a computer. For each employee you would make a list of merchandise categories (Bird, Cat, Dog, etc.). Then you would go through the list of ItemSales and cross off each merchandise category sold by the employee. When finished, you would look at the employee list to see which people have every category crossed off (or an empty list). You will do the same thing using queries.

Remember, if this query returns any rows at all, then the selected employee has not sold every one of the categories. What you really want then is a list of employees for whom this query returns no rows of data. In other words, the rows from this query should NOT EXIST.

The next step is to examine the entire list of employees and see which ones do not retrieve any rows from the query in Figure 5.31. The final query is shown in Figure 5.32. Note that the specific EmployeeID 5 has been replaced with the EmployeeID matching the value in the outer loop, which creates a correlated subquery. Unfortunately, you cannot avoid the correlated subquery in this type of

Figure 5.31

List the animal categories that have not been sold by EmployeeID 5. Use a basic NOT IN query.

List the Animal categories where merchandise has not been sold by an employee (#5).

```
SELECT Category
FROM Category
   WHERE (Category <> N'Other') And Category NOT IN
      (SELECT Merchandise.Category
       FROM Merchandise INNER JOIN (Sale INNER JOIN SaleItem
           ON Sale.SaleID = SaleItem.SaleID)
           ON Merchandise.ItemID = SaleItem.ItemID
       WHERE Sale.EmployeeID = 5)
```

Which employees have sold merchandise from every category?

```
SELECT Employee.EmployeeID, Employee.LastName
FROM Employee
WHERE Not Exists
 (SELECT Category
   FROM Category
   WHERE (Category NOT IN (N'Other', N'Reptile', N'Spider')
         And Category NOT IN
     (SELECT Merchandise.Category
      FROM Merchandise INNER JOIN (Sale INNER JOIN SaleItem
          ON Sale.SaleID = SaleItem.SaleID)
          ON Merchandise.ItemID = SaleItem.ItemID
      WHERE Sale.EmployeeID = Employee.EmployeeID)
 );
```

Figure 5.32

Example of NOT EXISTS clause. List the employees who have sold merchandise from every category (except "Other").

problem. This query returns four employees who have sold every type of animal merchandise. Observe that categories for Other, Reptile, and Spider have been removed from the list because the shortened product list does not contain any items for these categories. Another way to handle this problem would be to select the Distinct Category from the Merchandise table instead of the Category table.

The type of query in Figure 5.32 is commonly used to answer questions that include some reference to "every" item. In some cases, a simpler solution is to

Figure 5.33

Using CASE to count items. The hard way to count items in each category. It works, but needs to be edited if categories are added.

Which employees have sold merchandise from every category?

```
SELECT Employee.EmployeeID,Employee.LastName,
         Count(CASE Category WHEN 'Bird' THEN 1 END) As Bird,
         Count(CASE Category WHEN 'Cat' THEN 1 END) As Cat,
         Count(CASE Category WHEN 'Dog' THEN 1 END) As Dog,
         Count(CASE Category WHEN 'Fish' THEN 1 END) As Fish,
         Count(CASE Category WHEN 'Mammal' THEN 1 END) As Mammal,
         Count(CASE Category WHEN 'Reptile' THEN 1 END) As Reptile,
         Count(CASE Category WHEN 'Spider' THEN 1 END) As Spider
FROM Employee
INNER JOIN Sale ON Sale.EmployeeID=Employee.EmployeeID
INNER JOIN SaleAnimal ON Sale.SaleID=SaleAnimal.SaleID
INNER JOIN Animal ON Animal.AnimalID=SaleAnimal.AnimalID
GROUP BY Employee.EmployeeID, Employee.LastName
ORDER BY Employee.LastName;
```

just count the number of categories for each employee. One catch to this approach is that the DBMS must support the Count(DISTINCT) format. In general, these complex questions are probably better answered with multiple queries, or with tools provided by a data warehouse approach.

The query in Figure 5.32 is an interesting application of the EXISTS clause. However, there is an easier way to answer the question. You should build a cross-tab or pivot query that counts the number of items sold by each employee and by each category. Notice that this question contains two "by each" statements. You could write a simple query that contains both of those variables (Employee and Category) in the GROUP BY section. However, most people find it easier to read the results if they are presented in a table, with one Group By variable (Employee) as the rows and the other (Category) as the columns. Then each cell can contain the count of the number of items sold for a specific employee in a given category.

Figure 5.33 shows the basic query. Microsoft Access has a simpler crosstab query, but with traditional SQL, you need to compute each column separately. Hence, you have to use the CASE function to select each category of animal—which means you have to know the categories ahead of time. Essentially, you compute each column separately by using a CASE statement to select only rows that match the group condition you want for the column.

Figure 5.34 shows the result of the crosstab query. It is relatively easy to see the types of animals sold by each employee. To answer the overall question of who sold items from each category, you simply look for a row with no zeros. With this sample data, four employees have sold at least one item from each category. Technically, this query contains more information that required to answer the question. However, additional data is often useful. If you write the EXISTS query to return exactly the information requested, it will return no names. Oftentimes, it is preferable to see that several other employees come close to meeting the conditions, instead of simply saying that no one meets them exactly.

Figure 5.34

Crosstab query. The columns are built using the CASE statement to select each specific category. The rows are formed by the GROUP BY clause. Note that Oracle uses the DECODE function instead of the CASE statement.

EID	LastName	Bird	Cat	Dog	Fish	Mammal
1	Reeves		4	15	6	
2	Gibson	1	25	24	9	2
3	Reasoner	2	9	26	5	2
4	Hopkins	3	21	33		
5	James	3	7	8	11	2
6	Eaton	1	2	8		1
7	Farris	1	4	24	1	1
8	Carpenter	3	1	11	5	
9	O'Connor		5	10	3	1
10	Shields	1		5		
11	Smith		1			

```
SELECT DISTINCT Table.Column {AS Alias}, …
FROM Table/Query
INNER JOIN Table/Query ON T1.ColA = T2.ColB
WHERE (Condition)
GROUP BY Column
HAVING (Group Condition)
ORDER BY Table.Column
{UNION  Second Select }
```

Figure 5.35

SQL SELECT options. Remember that WHERE statements can have subqueries.

SQL SELECT Summary

The SQL SELECT command is powerful and has many options. To help you remember the various options, they are presented in Figure 5.35. Each DBMS has a similar listing for the SELECT command, and you should consult the relevant Help system for details to see if there are implementation differences. Remember that the WHERE clause can have subqueries. Also remember that you can use the SELECT line to perform computations, both in-line and aggregations across the rows.

Most database systems are picky about the sequence of the various components of the SELECT statement. For example, the WHERE statement should come before the GROUP BY statement. Sometimes these errors can be hard to spot, so if you receive an enigmatic error message, verify that the segments are in the proper order. Figure 5.36 presents a mnemonic that may help you remember the proper sequence. Also, you should always build a query in pieces, so you can test each piece. For example, if you use a GROUP BY statement, first check the results without it to be sure that the proper rows are being selected.

SQL Data Definition Commands

What are the SQL data definition commands? Everything to this point has focused on only one aspect of a database: retrieving data. Clearly, you need to perform many more operations with a database. SQL was designed to handle all common operations. One set of commands is described in this section: data definition commands to create and modify the database and its tables. Note that the SQL commands can be cumbersome for these tasks. Hence, most modern database sys-

Figure 5.36

Mnemonic to help remember the proper sequence of the SELECT operators.

Someone	SELECT
From	FROM
Ireland	INNER JOIN
Will	WHERE
Grow	GROUP BY
Horseradish and	HAVING
Onions	ORDER BY

```
CREATE SCHEMA AUTHORIZATION DBName Password
CREATE TABLE TableName (Column Type, …)
ALTER TABLE Table {Add, Column, Constraint, Drop}
DROP {Table TableName | Index IndexName ON TableName}
CREATE INDEX IndexName ON TableName (Column ASC/DESC)
```

Figure 5.37

Primary SQL data definition commands. In most cases you will avoid these commands and use a visual or menu-driven system to define and modify tables.

tems provide a visual or menu-driven system to assist with these tasks. The SQL commands are generally used when you need to automate some of these tasks and set up or make changes to a database from within a separate program.

The five most common data definition commands are listed in Figure 5.37. In building a new database, the first step is to **CREATE a SCHEMA**. A **schema** is a collection of tables. In some systems, the command is equivalent to creating a new database. In other systems, it simply defines a logical area where each user can store tables, which might or might not be in one physical database. The Authorization component describes the user and sets a password for security. Most DBMSs also have visually-oriented tools to perform these basic tasks. However, the SQL commands can be scripted and stored in a file that can be run whenever you need to recreate the database.

CREATE TABLE is one of the main SQL data definition commands. It is used to define a completely new table. The basic command lists the name of the table along with the names and data types for all of the columns. Figure 5.38 shows the format for the data definition commands. Additional options include the ability to assign default values with the DEFAULT command.

SQL 92 provides several standard data types, but system vendors do not yet implement all of them. SQL 92 also enables you to create your own data types with the **CREATE DOMAIN** command. For example, to ensure consistency you

Figure 5.38

The CREATE TABLE command defines a new table and all of the columns that it will contain. The NOT NULL command typically is used to identify the key column(s) for the table. The ALTER TABLE command enables you to add and delete entire columns from an existing table.

```
CREATE TABLE Customer
(           CustomerID          INTEGER NOT NULL,
            LastName            NVARCHAR(10),
            …
);

ALTER TABLE Customer
            DROP COLUMN ZIPCode;

ALTER TABLE Customer
            ADD COLUMN CellPhone NVARCHAR(15);
```

```
CREATE TABLE Order
        (OrderID         INTEGER NOT NULL,
         OrderDate      DATE,
         CustomerID   INTEGER,

        CONSTRAINT pkOrder PRIMARY KEY (OrderID),
        CONSTRAINT fkOrderCustomer FOREIGN KEY (CustomerID)
          REFERENCES Customer (CustomerID)
);
```

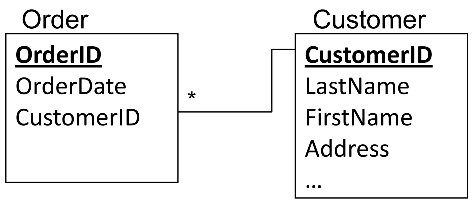


Figure 5.39

Identifying primary and foreign keys in SQL. Keys are defined as constraints that are enforced by the DBMS. The primary key constraint lists the columns that make up the primary key. The foreign key lists the column (CustomerID) in the current table (Order) that is linked to a column (CustomerID) in a second table (Customer).

could create a domain called DomAddress that consists of CHAR (35). Then any table that used an address column would refer to the DomAddress.

With SQL 92, you identify the primary key and foreign key relationships with constraints. SQL **constraints** are rules that are enforced by the database system. Figure 5.39 illustrates the syntax for defining both a primary key and a foreign key for an Order table. First, notice that each constraint is given a name (e.g., pkOrder). You can choose any name, but you should pick one that you will recognize later if problems arise. The primary key constraint simply lists the column or columns that make up the primary key. Note that each column in the primary key should also be marked as NOT NULL.

The foreign key constraint is easier to understand if you examine the relevant class diagram. Here you want to place orders only to customers who have data in the Customer table. That is, the CustomerID in the Order table must already exist in the Customer table. Hence, the constraint lists the column in the original Order table and then specifies a REFERENCE to the Customer table and the CustomerID.

The **ALTER TABLE** and **DROP TABLE** commands enable you to modify the structure of an existing table. Be careful with the DROP command, as it will remove the entire table from the database, including its data and structural definition. The ALTER TABLE command is less drastic. It can be used to ADD or DELETE columns from a table. Obviously, when you drop an entire column, all the data stored in that column will be deleted. Similarly, when you add a new column, it will contain NULL values for any existing rows.

You can use the CREATE INDEX and DROP INDEX commands to improve the performance of the database. Indexes can improve performance when retrieving data, but they can cause problems when many transactions take place in a short period of time. In general, these commands are issued once for a table. Typically, indexes are built for primary key columns. Most DBMSs automatically build those indexes.

Finally, as described in Chapter 4, the **CREATE VIEW** creates and saves a new query. The basic syntax is straightforward: *CREATE VIEW myview AS SELECT….* The command simply gives a name and saves any SELECT statement. Again, these commands are almost always easier to create and execute from a menu-driven interface. However, because you may have to create SQL data definition statements by hand sometime, so it is good to know how to do so.

SQL Data Manipulation Commands

What SQL commands alter the data stored in tables? A third set of SQL commands demonstrates the true power of SQL. The SELECT command retrieves data, whereas data manipulation commands are used to change the data within the tables. The basic commands and their syntax are displayed in Figure 5.40. These commands are used to insert data, delete rows, and update (change) the values of specific cells. Remember two points when using these commands: (1) They operate on sets of data at one time—avoid thinking in terms of individual rows, and (2) they utilize the power of the SELECT and WHERE statements you already know.

INSERT and DELETE

As you can tell from Figure 5.40, the **INSERT** command has two variations. The first version (VALUES) is used to insert one row of data at a time. Except for some programming implementations, it is not very interesting. Most database systems provide a visual or tabular data entry system that makes it easy to enter or edit single rows of data. Generally, you will build forms to make it easy for users to enter and edit single rows of data. These tools automatically build the single-row INSERT command. On most systems, the data will be inserted directly to the tables. In a few cases, you might have to write your own INSERT statement.

The second version of the INSERT command is particularly useful at copying data from one table into a second (target) table. Note that it accepts any SE-

Figure 5.40

Common SQL commands to add, delete, and change data within existing tables. The commands operate on entire sets of data, and they utilize the power of the SELECT and WHERE statements, including subqueries.

```
INSERT INTO target (column1, column2, …)
          VALUES (value1, value2, …)

INSERT INTO target (column1, column2, …)
          SELECT … FROM …

DELETE FROM table WHERE condition

UPDATE table
          SET Column1=Value1, Column2=Value2, …
          WHERE condition
```

```
INSERT INTO OldCustomers
SELECT *
FROM Customer
WHERE CustomerID IN
(SELECT Sale.CustomerID
          FROM Customerr INNER JOIN Sale
          ON Customer.CustomerID=Sale.CustomerID
          GROUP BY Sale.CustomerID
          HAVING Max(Sale.SaleDate) < '01-Jul-2013')  );
```

Figure 5.41

INSERT command to copy older data rows. Note the use of the subquery to identify the rows to be copied.

LECT statement, including one with subqueries, making it far more powerful than it looks. For example, in the Pet Store database, you might decide to move older Customer data to a different computer. To move records for customers who have not purchased anything since the start of July, you would issue the INSERT command displayed in Figure 5.41. Notice that the subquery selects the customers based on the date they placed their latest sale. The INSERT command then copies the associated rows in the Customer table into an existing OldCustomers table.

The query in Figure 5.41 just copies the specified rows to a new table. The next step is to delete them from the main Customer table to save space and improve performance. The **DELETE** command performs this function easily. As Figure 5.42 illustrates, you simply replace the first two rows of the query (INSERT and SELECT) with DELETE. Be careful not to alter the subquery. You can use the cut-and-paste feature to delete only rows that have already been copied to the backup table. Be sure you recognize the difference between the DROP and DELETE commands. The DROP command removes an entire table. The DELETE command deletes rows within a table.

UPDATE

The syntax of the **UPDATE** command is similar to the INSERT and DELETE commands. It, too, makes full use of the WHERE clause, including subqueries. The key to the UPDATE command is to remember that it acts on an entire collec-

Figure 5.42

DELETE command to remove the older data. Use cut and paste to make sure the subquery is exactly the same as the previous query.

```
DELETE
FROM Customer
WHERE CustomerID IN
          (SELECT FROM Customerr INNER JOIN Sale
          ON Customer.CustomerID=Sale.CustomerID
          GROUP BY Sale.CustomerID
          HAVING (Max(Sale.SaleDate) < '01-Jul-2013')  );
```

```
UPDATE Merchandise
SET ListPrice = ListPrice * 1.10
WHERE Category = 'Cat';
```

```
UPDATE Merchandise
SET ListPrice = ListPrice * 1.20
WHERE Category = 'Dog';
```

Figure 5.43

Sample UPDATE command. If the CASE function is not available, use two separate statements to increase the list price by 10 percent for cats and 20 percent for dogs.

tion of rows at one time. You use the WHERE clause to specify which set of rows need to be changed.

In the example in Figure 5.43, managers wish to increase the ListPrice of the merchandise for cats and dogs. The price for cat merchandise should increase by 10 percent and the price for dog merchandise by 20 percent. Because these are two different categories, you will often use two separate UPDATE statements. However, this operation provides a good use for the CASE function. You can reduce the operation to one UPDATE statement by replacing the 1.10 and 1.20 values with a CASE statement that selects 1.10 for Cats and 1.20 for Dogs.

The UPDATE statement has some additional features. For example, you can change several columns at the same time. Just separate the calculations with a comma. You can also build calculations from any row within the table or query. For example, merchandise list price could take into consideration the quantity on hand with the command SET ListPrice = ListPrice*(1 - 0.001*QuantityOnHand). This command takes 1/10 of 1 percent off the price for extra items in inventory.

Notice the use of the internal Date( ) function to provide today's date in the last example. Most database systems provide several internal functions that can be used within any calculation. These functions are not standardized, but you can generally get a list (and the syntax chart) from the system's Help commands. The Date, String, and Format functions are particularly useful.

When using the UPDATE command, remember that all the data in the calculation must exist on one row within the query. There is no way to refer to a previous or next row within the table. If you need data from other rows or tables, you can build a query to join tables. However, you can update data in only a single table at a time.

Quality: Testing Queries

How do you know if your query is correct? The greatest challenge with complex queries is that even if you make a mistake, you usually get results. The problem is that the results are not the answer to the question you wanted to ask. The only way to ensure the results are correct is to thoroughly understand SQL, to build your queries carefully, and to test your queries.

Figure 5.44 outlines the basic steps for dealing with complex queries. The first step is to break complex queries into smaller pieces, particularly when the query involves subqueries. You need to examine and test each subquery separately. You can do the same thing with complex Boolean conditions. Start with a simple condition, check the results, and then add new conditions. When the subqueries are

- Break questions into smaller pieces.
- Test each query.
  - Check the SQL.
  - Look at the data.
  - Check computations.
- Combine into subqueries.
  - Use the cut-and-paste features to reduce errors.
  - Check for correlated subqueries.
- Test sample data.
  - Identify different cases.
  - Check final query and subqueries.
  - Verify calculations.
- Test SELECT queries before executing UPDATE queries.
- Optimize queries that run multiple times.
  - Run a query optimizer.
  - Think about new ways to structure the query.

Figure 5.44

Steps to building quality queries. Be sure there are recent backups of the database before you execute UPDATE or DELETE queries.

correct, use cut-and-paste techniques to combine them into one main query. If necessary, save the initial queries as views, and use a completely new query to combine the results from the views. The third step is to create sample data to test the queries. Find or create data that represents the different possible cases. Optimize queries that will become part of an application and run multiple times. Most DBMSs have an optimizer that will suggest performance improvements. You should also look for alternate ways to write the query to find a faster approach.

Figure 5.45

Sample query: Which customers who adopted dogs also bought cat products (at any time)? Build each query separately. Then paste them together in SQL and add the connecting link. Use sample data to test the results.

```
SELECT  DISTINCT Animal.Category, Sale.CustomerID
FROM Sale INNER JOIN Animal
 ON Animal.SaleID = Sale.SaleID
WHERE (Animal.Category=N'Dog')

        AND Sale.CustomerID IN (

        SELECT DISTINCT Sale.CustomerID
        FROM Sale INNER JOIN (Merchandise INNER JOIN
SaleItem
         ON Merchandise.ItemID = SaleItem.ItemID)
         ON Sale.SaleID = SaleItem.SaleID
        WHERE (Merchandise.Category=N'Cat')
);
```

In terms of quality issues, consider the example in Figure 5.45: List customers who adopted dogs and also bought cat products. The query consists of four situations:

1. Customers adopted dogs and cat products on the same sale.
2. Customers adopted dogs and then cat products at a different time.
3. Customers adopted dogs and never bought cat products.
4. Customers never adopted dogs but did buy cat products.

Because there are only four cases, you should create data and test each one. If there were thousands of possible cases, you might have to limit your testing to the major possibilities.

The final step in building queries involves data manipulation queries (such as UPDATE). You should first create a SELECT query that retrieves the rows you plan to change. Examine and test the rows to make sure they are the ones you want to alter. When you are satisfied that the query is correct, make sure you have a recent backup of the database—or at least a recent copy of the tables you want to change. Now you can convert the SELECT query to an UPDATE or DELETE statement and execute it.

Summary

Always remember that SQL operates on sets of data. The SELECT command returns a set of data that matches some criteria. The UPDATE command changes values of data, and the DELETE command deletes rows of data that are in a specified set. Sets can be defined in terms of a simple WHERE clause. The key to understanding SQL is to think of the WHERE clause as defining a set of data.

To create queries to answer complex business questions, break the question into pieces and build simple queries to retrieve data for each piece. Then combine the sets of data using inner joins, outer joins, subqueries, or set operators. Subqueries are powerful, but be careful to ensure that the query accurately represents the business questions. You must test subqueries in pieces and make sure you understand exactly what each piece is returning.

In everyday situations, data can exist in one table but not another. For example, you might need a list of customers who have not placed orders recently. The problem can also arise if the DBMS does not maintain referential integrity—and you need to find which orders have customers with no matching data in the customer table. Outer joins (or the NOT IN subquery) are useful in these situations.

The most important thing to remember when building queries is that if you make a mistake, most likely the query will still execute. Unfortunately, it will not give you the results you wanted. That means you have to build your queries carefully and always check your work. Begin with a smaller query and then add elements until you get the query you want. To build an UPDATE or DELETE query, always start with a SELECT statement and check the results. Then change it to UPDATE or DELETE.

A Developer's View

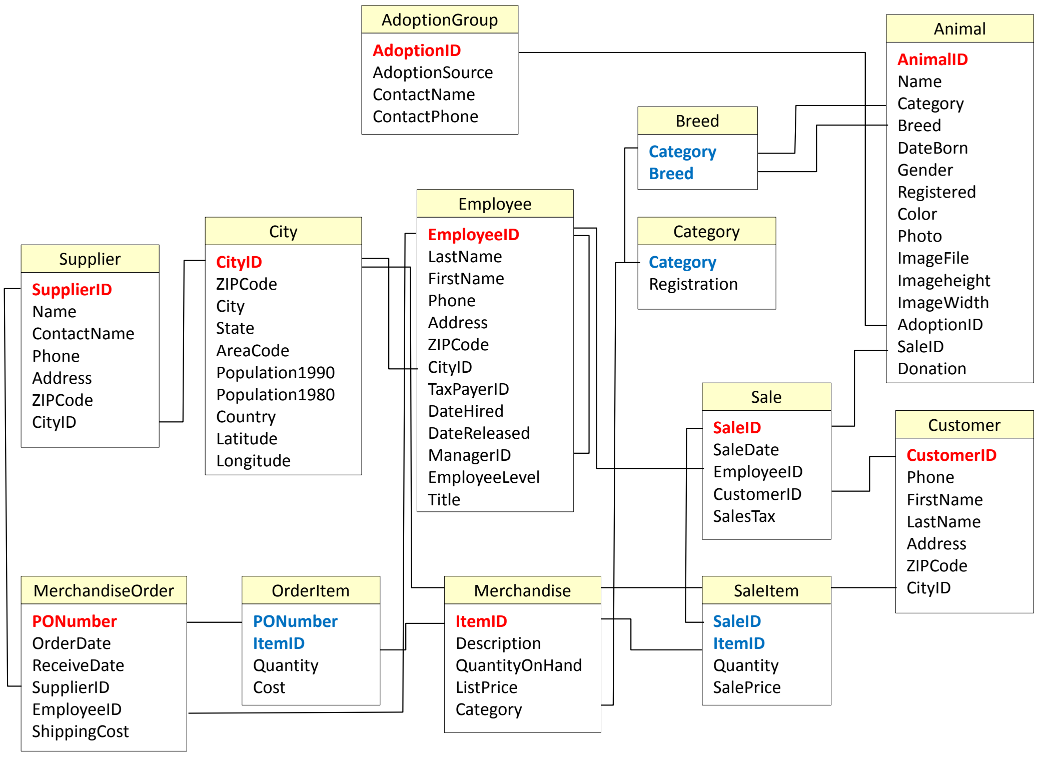
Miranda saw that some business questions are more complex than others. SQL subqueries and outer joins are often used to answer these questions. Practice the SQL subqueries until you thoroughly understand them. They will save you hundreds of hours of work. Think about how long it would take to write code to answer some of the questions in this chapter! For your class project, you should create several queries to test your skills, including subqueries and outer joins. You should build and test some SQL UPDATE queries to change sets of data. You should be able to use SQL to create and modify tables.

Key Terms

ALL
ALTER TABLE
ANY
CASE
constraint
correlated subquery
CREATE DOMAIN
CREATE SCHEMA
CREATE TABLE
CREATE VIEW
DELETE
DROP TABLE
equi-join
EXCEPT
EXISTS
FULL JOIN
IN
inequality join
INSERT
INTERSECT
LEFT JOIN
nested query
outer join
reflexive join
RIGHT JOIN
schema
self join
subquery
UNION
UPDATE

Review Questions

1. What is a subquery and in what situations is it useful?
2. What is a correlated subquery and why does it present problems?
3. ✓ How do you find items that are not in a list, such as customers who have not placed orders recently?
4. How do you join tables when the JOIN column for one table contains data that is not in the related column of the second table?
5. How do you join a column in one table to a related column in the same table?
6. What are inequality joins and when are they useful?
7. ✓ What is the SQL UNION command and when is it useful?
8. What is the purpose of the SQL CASE function?
9. What are the basic SQL data definition commands?
10. What are the basic SQL data manipulation commands?
11. ✓ How are UPDATE and DELETE commands similar to the SELECT statement?



Exercises

Sally's Pet Store

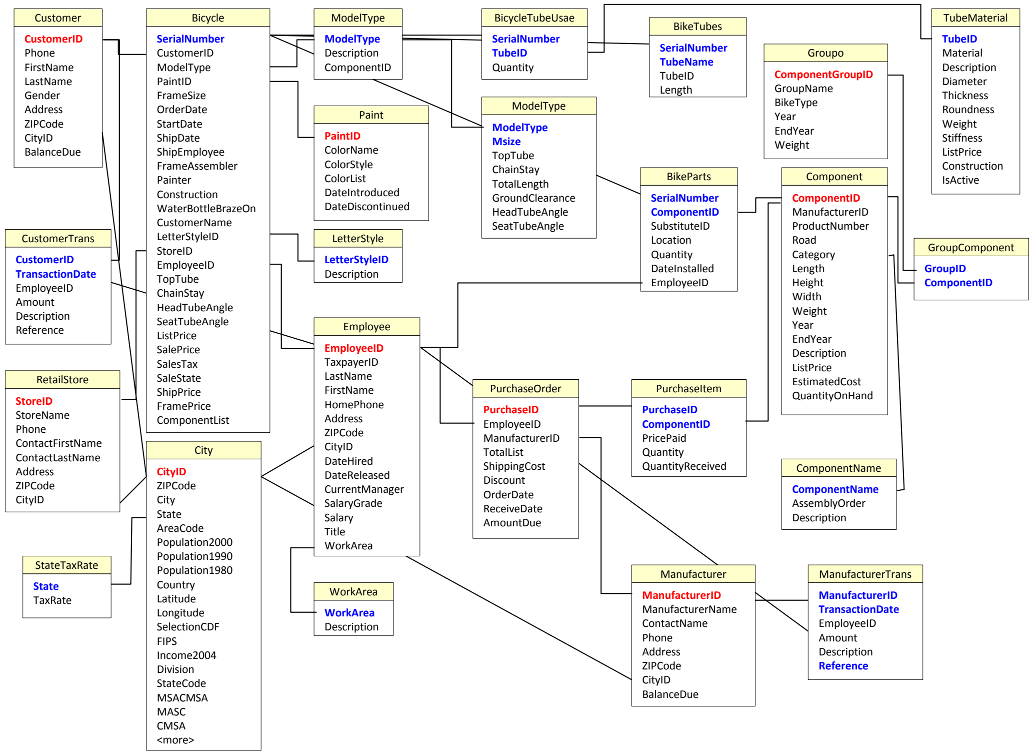
Write the SQL statements that will answer questions 1 through 16 based on the tables in the Pet Store database. Test your queries in the database. Hint: Many are easier if you split the question into multiple queries.

1. Which suppliers did not deliver any items in September?
2. ✓ Which employees did not sell any items in June?
3. Which categories of merchandise were not sold during May?
4. ✓ Which breed of Cat has never been adopted through the store?
5. What was the percentage of sales value by merchandise category in March?
6. Which category of animal was most likely (percent) to be adopted in the first three months?
7. ✓ Which employee had the highest percent of the number of sales (not value) in January?
8. Which supplier has the highest average percentage of shipping cost to total order value?
9. List the total adoptions and percentage by adoption group in April.

10. Which employee has been the top monthly seller the most number of times?
11. ✓ What is the amount of money customers spent on cat products after they adopted a cat?
12. List customers who purchased Cat merchandise in January and March?
13. List employees who ordered items from the same supplier in March and April (could be different products).

Make a backup copy before attempting the remaining Pet Store queries.

14. ✓ Write the SQL CREATE TABLE command to create a new Employee table with no data.
15. Write the SQL command to copy the data to the new Employee table for employees who did not sell anything in December.
16. Write the SQL command to delete the employees from the original Employee table who did not sell anything in December—except for Ms. Smith, the owner.
17. ✓ Write a query to increase the list price of Dog merchandise by 5 percent.



Rolling Thunder Bicycles

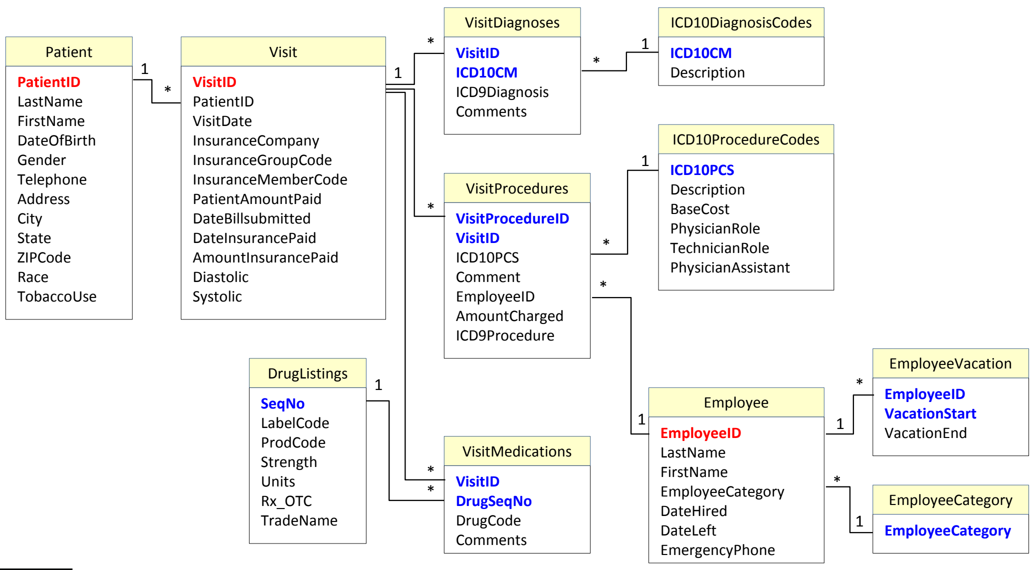
Write the SQL statements that will answer questions 17 through 32 based on the tables in the Rolling Thunder database. Build your queries in Access.

18. Which employee has been #1 in monthly sales value for the most number of months in 2010-2011?
19. Which paint colors were not used in 2012?
20. What percent of race bikes sold in 2012 used Shimano, Campy, and SRAM cranks? (Give the percent of the total for each manufacturer.)
21. List customers who bought a full suspension mountain bike after they had purchased a regular mountain bike.
22. List all of the people who are managed by Roland Venetiaan.
23. In 2012, which employees who took an order for a bicycle also shipped that same bicycle? Hint: Connect the Employee table to ShipEmployee.
24. Compute the percentage of value of sales by model type for each year 2010 – 2013.
25. Using a UNION query, list the employees who painted bicycles on March 15, 2012 or framed them on that date (StartDate) (or both). Hint: Join the Employee table to Painter and then to FrameAssembler.

26. In 2012, what percent of bicycle sales (by count) were made without the help of a retail store (StoreID=1 or 2).
27. Which manufacturers did not sell any items to Rolling Thunder Bicycles in 2012?
28. For Road component groups in 2012 (Component.Year), what is the average percent of the total group weight contributed by the crank?
29. ✓ How has the percent share of sales value for Race bikes (to total) changed over time (by year)?
30. Use SQL to create a new SalesRanking table as shown.

Category	SalesLow	SalesHigh
Top	0.10	1.0
Acceptable	0.05	0.10
Weak	0.0	0.05

31. Write the query to insert the rows of data row into the table shown in the previous exercise.
32. Create a query to compute sales by employee by month and the employee's percent of total monthly sales. Combine the table data from the table in the previous exercise to assign the appropriate category to each employee for each month.
33. ✓ Write a query to delete the last row (Weak) in the new SalesRanking table.
34. Write a query to delete the entire SalesRanking table.



Corner Med

35. List the physicians and the percentage of patients/visits seen by each one for the month of May. Do not include non-physicians in the computations.

36. For the year, list the top 10 diagnoses and the percentage of times each was applied.

37. For the month of March, list the percentage of visits covered by each type of insurance company.

38. For each month, compute the percentage of the number of visits by patient gender.

39. List the patients who returned for at least one visit after being diagnosed with J069 (respiratory infection).

40. Which two-letter procedures have not been performed?

41. What is the average number of medications prescribed per visit for each physician?

42. Which patients who have been diagnosed with ICD10 code E784 have also been diagnosed (at any time) with code E039?

43. Which patients have been seen by all three physicians (at any time)?

44. Create a summarization of patients that show the percentage by gender and tobacco use.

45. Use SQL to create a table (VisitCategory) that can be used to categorize patients by the number of visits in a year:

Category	MinVisits	MaxVisits
Many	2	20
Seldom	1	2
Rare	0	1

46. Write the INSERT commands to add the rows in the table for the previous exercise.

47. Write a query using the table in the previous query to categorize the patients by number of visits for one year.

48. Write the SQL command to change the MaxVisits value in the "Many" row to 30.

49. Write the SQL command to remove the table.

50. The GEMICD9xICD10_CM crosswalk table matches the older ICD9 diagnostic codes to the newer ICD10 codes. Create a query that ignores the NoDx entries. Create a second query to find the older ICD9 codes in the VisitDiagnoses table that do not have an official match in the new ICD10 code. (Ignore the ICD10 values in the VisitDiagnoses table—which were created using this process.) Bonus: How would you find codes for the ICD9 entries that are missing cross matches?

51. The GEMICD9xICD10_PCS crosswalk table matches procedure codes between the older ICD9 and newer ICD10 classifications. Assume that the VisitProcedures table has only the older ICD9 procedure code and blank values for the ICD10 codes. Write the query to use the crosswalk table to match the values and transfer the correct ICD10 entry into the VisitProcedures table. Note: If you run the query, make a backup copy of the table and database.

52. The GEMICD9xICD10_PCS crosswalk table maps older ICD9 procedure codes to the newer ICD10 codes. Are any of the ICD9 codes mapped to more than one ICD10 code? If so, in the process used in the previous exercise, what will happen? Which codes will be transferred?

Web Site References

http://www.sigmod.org/	Association for Computing Machinery—Special Interest Group: Management of Data.
http://www.acm.org/dl	ACM digital library containing thousands of searchable full-text articles. Check library.
http://www.oracle.com/technetwork/indexes/documentation/index.html	Oracle online documentation library, including SQL Reference. (Version db102 will change.)
http://msdn.microsoft.com/en-us/library/ms130214.aspx	Microsoft SQL Server Books Online reference.
http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp	IBM DB2 reference.
http://dev.mysql.com/doc/	MySQL Reference.

Additional Reading

Celko, J., Joe Celko's SQL Puzzles and Answers, 2e, San Mateo: Morgan Kaufmann, 2006. [Challenging SQL problems with solutions.]

Faroult, Stephane, The Art of SQL, O'Reilly, 2006. [Strategy and performance in building queries.]

Appendix: Introduction to Programming

Many books will help you learn to write computer programs. The purpose of this appendix is to review the highlights of programming and to point out some of the features that are important to programming within a DBMS. If you are new to programming, you should consider reading several other books to explain the details and logic behind programming.

Variables and Data

One of the most important consequences of programming in a database environment is that there can be three categories of data: (1) data stored in a table, (2) data held in a control on a form or report, and (3) traditional data variables that are used to hold temporary results. Chapter 3 focuses on storing data within tables. Chapter 6 describes how to create forms and the role of data controls. Chapter 8 provides more details of how the three types of variables interact when building applications. For now, you must learn details about basic programming variables.

Any procedure can create variables to hold data. A program **variable** is like a small box: it can hold values that will be used or transferred later. Variables have unique names. More importantly, variables can hold a certain **data type.** Common types of variables are displayed in Figure 5.1A. They can generally be classified into three categories: integers (1, 2, -10, …); reals (1.55, 3.14, …); and strings ("123 Main Street", "Jose Rojas", …).

Each type of variable takes up a defined amount of storage space. This space affects the size of the data that the variable can hold. The exact size depends on the particular DBMS and the operating system. For example, a short integer typically takes 2 bytes of storage, which is 16 bits. Hence it can hold 216 values or numbers between –32,768 and 32,767. Real numbers can have fractional values. There are usually two sizes: single and double precision. If you do not need many variables, it is often wise to choose the larger variables (integers and double-precision reals). Although double-precision variables require more space and take longer to process, they provide room for expansion. If you choose too small of a variable, a user might crash your application or get invalid results. For example, it would be a mistake to use a 2-byte integer to count the number of customers—since a firm could generally anticipate having more than 65,000 customers. Along the same

Figure 5.1A

Program variable types. Ranges are approximate but supported by most vendors. Note that decimal variables help prevent round-off errors

Type	Bytes	Range
Short	2	-32,768 to 32,767
Integer	4	+/- 2,147,483,647
Long	8	+/- 9,223,372,036,854,775,807
Float	4	+/- 1.5 10e45 (7 digits)
Double	8	+/- 5.0 10e324 (15 digits)
Decimal	16	+/- 1.0 10e28 (28 digits)
String	any	any

lines, you should use the Currency data type for monetary values. In addition to handling large numbers, it avoids round-off errors that are common to floating-point numbers.

Variable Scope

The scope and lifetime of a variable are crucial elements of programming, particularly in an event-driven environment. Variable **scope** refers to where the variable is accessible, that is, which procedures or code can access the data in that variable. The **lifetime** identifies when the variable is created and when it is destroyed. The two properties are related and are generally automatic. However, you can override the standard procedures by changing the way you declare the variable. In most systems, the scope and lifetime are based on where the variable is declared.

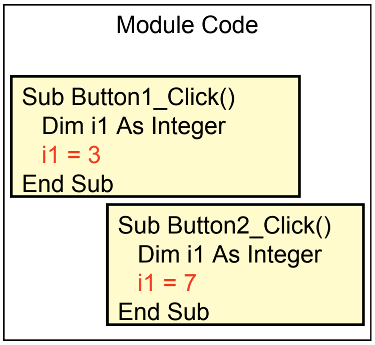
All data variables should be explicitly declared: they should be identified before they are used. The Basic language uses a Dim statement to declare variables. Many other languages declare variables by specifying the data type first. Most commonly, the variable is created within the event procedure and is a **local variable**. When the procedure starts, the local variable is created. Any code within that procedure can use the variable. Code in other procedures cannot see the variable. When the procedure ends, the local variable and its data are destroyed.

Figure 5.2A shows two buttons on a form. Each button responds to a Click event, so two procedures are defined. Each procedure can have a variable called i1, but these two variables are completely separate. In fact, the variables are not created until the button is clicked. Think of the procedures as two different rooms. When you are in one room, you can see the data for that room only. When you leave the room, the data is destroyed.

However, what if you do not want the data to be destroyed when the code ends, or you want to access the variable from other procedures? You have two choices: (1) Change the lifetime of the variable by declaring it static, or (2) Change the scope of the variable by declaring it in a different location. You should avoid declaring a static variable unless it is absolutely necessary (which is rare). If the variable is static, it keeps its value from the previous time the procedure was called. In the example, each time the button is clicked, the value for *i3* will remain from the prior click. You might use this trick if you need to count the number of times the button is clicked.

Figure 5.2A

Variable scope and lifetime. Each event has its own procedure with independent variables that are created and destroyed each time a routine is executed.



```
Sub Button1_Click()
  Dim i1 As Integer
  i1 = 3
End Sub
```

```
Sub Button2_Click()
  Dim i1 As Integer
  i1 = 7
End Sub
```

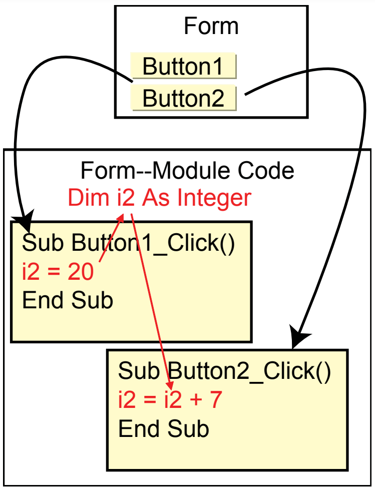


Figure 5.3A

Global variables. Variables that are defined in the form's General section are accessible by any function on that form (or module).

A more useful technique is to change where the variable is defined. Figure 5.3A shows that event procedures are defined within a form or a **module**, which is a collection of related procedures. The variable *i2* is defined for the entire form or module. The lifetime of the variable is established by the form, that is, the variable is created and destroyed as the form is opened and closed. The scope of the variable is that all procedures in the form can see and change the value. On the other hand, procedures in other forms or modules do not know that this variable exists.

Procedures or functions also have a scope. Any procedure that you define on a form can be used by other procedures on that form. If you need to access a variable or a procedure from many different forms or reports, you should define it on a separate module and then declare it as global (or public).

Be careful with global or public variables. A programmer who tries to revise your code might not know that the variable is used in other procedures and might accidentally destroy an important value. On forms the main purpose of a global variable is to transfer a value from one event to another one. For example, you might need to keep the original value of a text control—before it is changed by a user—and compare it to the new value. You need a global variable because two separate events examine the text control: (1) The user first enters the control, and (2) The user changes the data. It is sometimes difficult to create global or shared variables in certain systems. In these cases, you might need to store the global variables within a special database table.

Computations

One of the main purposes of variables is to perform calculations. Keep in mind that these computations apply to individual variables—one piece of data at a time. If you need to manipulate data in an entire table, it is usually best to use the SQL commands described in Chapter 5. Nonetheless, there are times when you need more complex calculations.

Operation	Common Syntax
Arithmetic	+ - * /
Exponentiation	^ or Power
Integer Divide	\
Modulus	mod

Figure 5.4A

Common arithmetic operators. Add (+), subtract (-), multiply (*), and divide (/). Exponentiation and integer arithmetic are often used for special tasks. For example, integer arithmetic is useful for dividing objects into groups.

Standard arithmetic operations (add, subtract, multiply, and divide) are shown in Figure 5.4A. These operators are common to most programming languages. Some nonstandard, but useful operators include exponentiation (raise to a power, e.g., 2^3 = 2*2*2 = 8), and integer divide (e.g., 9 \ 2 = 4), which always returns an integer value. The mod function returns the modulus or remainder of an integer division (e.g., 15 mod 4 = 3, since 15 - 12 = 3). These last two functions are useful when you need to know how many of some objects will fit into a fixed space. For example, if there are 50 possible lines on a page and you need to print a report with 185 lines, then 185 \ 50 = 3 pages, and 185 Mod 50 leaves 35 lines on a fourth page.

Most languages support string variables, which are used to hold basic text data, such as names, addresses, or short messages. A string is a collection (or array) of characters. Sometimes you will need to perform computations on string variables. How can you perform computations on text data? The most common technique is to **concatenate** (or add) two strings together. For example, if FirstName is "George" and LastName is "Jones", then FirstName & LastName is "GeorgeJones". Notice that if you want a space to appear between the names, you have to add one: FirstName & " " & LastName.

Figure 5.5A lists some of the common string functions. You can learn more about the functions and their syntax from the Help system. Commonly used functions include the Left, Right, and Mid, which examine portions of the string. For

Figure 5.5A

Common string functions to add strings, extract portions, examine characters, convert case, compare two strings, and format numerical data into a string variable.

Concatenation (& or +)
Left, Right, Mid, or SubStr
Trim, LTrim, RTrim
LCase, UCase
InStr or IndexOf
"Frank" + "Rose" → "Frank Rose"
Left("Jackson", 5) → "Jacks"
Trim("   Maria ") → "Maria"
Len("Ramanujan") → 9
"8764 Main".IndexOf(" ") → 5

Exp, Log	$x = \log_e(e^x)$
Atn, Cos, Sin, Tan	Trigonometric functions
Sqr or Sqrt	Square root
Abs	Absolute value: Abs(-35) → 35
Sgn	Signum: Sgn(-35) → -1
Int	Integer: Int(2.718) → 2
Rnd	Random number

Figure 5.6A

Standard mathematical functions. Even in business applications, you often need basic mathematical functions.

example, you might want to see only the first five characters on the left side of a string.

Standard Internal Functions

As you may recall from courses in mathematics, several common functions are used in a variety of situations. As shown in Figure 5.6A, these functions include the standard trigonometric and logarithmic functions, which can be useful in mapping and procedures involving measurements. You also will need a function to compute the square root and absolute value of numbers. The Int (integer) function is useful for dropping the fractional portion of a number. Most languages also provide a random number generator, which will randomly create numbers between 0 and 1. If you need another range of numbers, you can get them with a simple conversion. For example, to generate numbers between 40 and 90, use the following function: y = 40 + (90 - 40)*Rnd.

In a database environment, you will often need to evaluate and modify dates. It is also useful to have functions that provide the current date (Date) and time (Now). Two functions that are useful in business are the DateAdd and DateDiff functions. As illustrated in Figure 5.7A, the DateAdd function adds days to a given date to find some date in the future. The DateDiff function computes the difference between two dates. Usually, you will want to compute the number of days between various dates. However, the functions can often compute the number of months, weeks, years and so on.

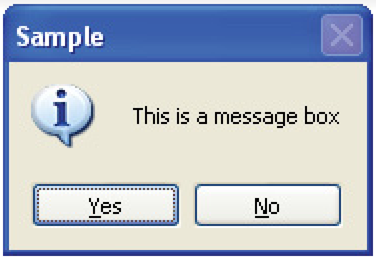
Input and Output

Handling input and output were crucial topics in traditional programming. These topics are still important, but the DBMS now performs most data-handling rou-

Figure 5.7A

Date and time functions. Business problems often require computing the number of days between two dates or adding days to a date to determine when payments are due.

Date, Now, Time	Current date and time
DateAdd, DateDiff	Date arithmetic: DateDue = DateAdd(“d”, 30, Date())



This is a message box

Figure 5.8A

Sample message box. The message box interrupts the user and displays a few limited choices. It often handles errors or problems.

tines and the operating system or Web browser handles most of the user interface. Common forms and reports (Chapters 6 and 7) are used for most input and output tasks.

Remember that an important feature of a Windows interface is that users control the flow of data entry; that is, the designer provides a form, and users work at their own pace without interruption. Occasionally, you might choose to interrupt the user—either to provide information or to get a specific piece of data. One common reason is to display error messages. Two basic functions serve this purpose: MsgBox and InputBox. As shown in Figure 5.8A, a message box can contain buttons. The buttons are often used to indicate how the user wants to respond to some problem or error.

An InputBox is a special form that can be used to enter very small amounts of text or a single number. Neither the user nor the developer has much control over the form. In most cases you would be better off creating your own blank form. Then you can have more than one text box, and you can specify and control the buttons. The InputBox is usually for temporary use when development time is extremely limited.

Conditions

The ability to test and respond to conditions is one of the most common reasons for writing your own procedures. The basic conditional statement (if …then … else) is relatively easy to understand. The structure is shown in Figure 5.9A. A

Figure 5.9A

Conditions. Basic conditions are straightforward. Indenting conditions highlights the relationships.

```
If (Condition1) Then
  Statements for true
Else
  Statements for false
  If (Condition2) Then
    Statements for true
  End If
End If
```

```
response = 1, 2, 3, 4, 5
If (response = 1) Then
  ‘ Statements for 1
Else
  If (response = 2) Then
    ‘ Statements for 2
  Else
    If (response = 3) Then
      ‘ More If statements
    End If
  End If
End If
```

Figure 5.10A

Nested conditions to test for a user response. The code becomes harder to read as more conditions are added.

condition is evaluated to be true or false. If it is true, then one set of statements is executed; otherwise, the second set is performed.

Conditions can be complex, particularly when the condition contains several AND, and OR connectors. Some developers use a NOT statement to reverse the value of a condition. Be careful when writing conditions. Your goals are to make sure that the condition evaluates to the correct value and to make sure that other developers can understand the code.

You should always include parentheses to specify the order of evaluation and, for complex conditions, create sample data and test the conditions. Also, indent your code. Indenting is particularly important for **nested conditions**, in which the statements for one condition contain another conditional statement.

The Select Case statement is a special type of conditional statement. Many procedures will need to evaluate a set of related conditions. As a simple example, consider what happens if you use a message box with three buttons (Yes, No, and Cancel). You will have to test the user’s choice for each option. Figure 5.10A shows how the code might look when you use nested conditions.

Figure 5.11A

The Select statement. The select statement tests the response variable against several conditions. If the response matches a case in the list, the corresponding code is executed.

```
Response = 1, 2, 3, 4, 5
Select Case response
  Case 1
    ‘ Statements for 1
  Case 2
    ‘ Statements for 2
  Case 3
    ‘ More Case statements
  Default
End Case
```

Figure 5.11A shows the same problem written with the Select Case statement. Note that this code is much easier to read. Now think about what will happen if you have 10 choices. The If-Then code gets much worse, but the Select Case code just adds new lines to the bottom of the list.

Loops

Iteration or **loops** are another common feature in procedures. Although you should use SQL statements (UPDATE, INSERT, etc.) as much as possible, sometimes you will need to loop through a table or query to examine each row individually.

Some of the basic loop formats are illustrated in Figure 5.12A. The For/Next loop is generally used only if you need a fixed number of iterations. The Do loop is more common. An important feature of loops is the ability to test the condition at the top or the bottom of the loop. Consider the example in which the condition says to execute the statements if (*x* <= 10). What happens when the starting value of *x* is 15? If you test the condition at the top of the loop, then the statements in the loop will never be executed. On the other hand, if you test the condition at the bottom, then the statements in the loop will be executed exactly one time—before the condition is tested.

Just as with conditions, it is good programming practice to indent the statements of the loop. Indents help others to read your code and to understand the logic. If there are no problems within a loop, your eye can easily find the end of the loop.

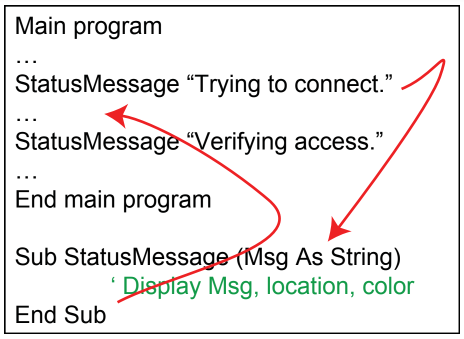
Be careful with loops: if you make a mistake, the computer may execute the statements of your loop forever. (On most personal computers, Ctrl+Break will usually stop a runaway loop.) A common mistake occurs when you forget to change the conditional variable (*x* in the examples). In tracking through a data query, you might forget to get the next row of data, in which case your code will perform the same operations forever on one row of data. A good programming practice is to always write loops in four steps: (1) Write the initial condition, (2) Write the ending statement, (3) Write a statement to update the conditional variable, and (4) Write the interior code. The first three statements give you the structure. By writing and testing them first, you know that you will be using the correct data.

Figure 5.12A

Iteration. All versions of loops follow a common format: initialize a counter value, perform statements, increment the counter, and test the exit condition. You can test the condition at the start or end of the loop.

Do Until (x > 10)
  ' Statements
  x = x + 1
Loop
	Do While (x <= 10)
	  ' Statements
	  x = x + 1
	Loop

Do
  ' Statements
  x = x + 1
Loop Until (x > 10)
	For x = 1 to 10
	  ' Statements
	Next x



```
Main program
…
StatusMessage "Trying to connect."
…
StatusMessage "Verifying access."
…
End main program

Sub StatusMessage (Msg As String)
         ' Display Msg, location, color
End Sub
```

Figure 5.13A

Subroutine. The StatusMessge subroutine can be called from any location. When the subroutine is finished, it returns to the calling program.

Subroutines

An important concept in programming is the ability to break the program into smaller pieces as subroutines or functions. A **subroutine** is a portion of code that can be called from other routines. When the subroutine is finished, control returns to the program code that called it. The goal of using subroutines is to break the program into smaller pieces that are relatively easy to understand, test, and modify.

A subroutine is essentially a self-contained program that can be used by many other parts of the program. For example, you might create a subroutine that displays a status message on the screen. As illustrated in Figure 5.13A, you would write the basic routine once. Then anytime you need to display a status message, your program calls this routine. By passing the message to the subroutine, the actual message can change each time. The advantage of using the subroutine is that you have to write it only once. In addition, your status messages can be standardized because the subroutine specifies the location, style, and color. To change the format, you simply modify the few lines of code in the one subroutine. Without the subroutine, you would have to find and modify code in every location that displayed a status message.

A data variable that is passed to a function or a subroutine is known as a **parameter**. There are two basic ways to pass a parameter: by reference and by value. The default method used by Microsoft Access is pass-by-reference. In this case the variable in the subroutine is essentially the same variable as in the original program. Any changes made to the data in the subroutine will automatically be returned to the calling program. For example, consider the two examples in Figure 5.14A. Changes to the variable *j2* in the subroutine will automatically be passed back to the calling program. However, when only the value is passed, a copy is made in the subroutine. Changes made to the data in the subroutine will not be returned to the calling program. Unless you are absolutely certain that you want to alter the original value in the calling program, you should always pass variables by value. Subroutines that use pass-by-reference can cause errors that are difficult to find in programs. Some other programmer might not realize that your subroutine changed the value of a parameter.

Main: j = 3 DoSum(j) ‘ j is now equal to 8 … Sub DoSum(By Ref j2 As Integer) j2 = 8 End Sub	By Reference Changes to data in the subroutine are passed back to the calling program.
Main: j = 3 DoSum(j) ‘j is still equal to 3 … Sub DoSum(By Val j2 As Integer) J2 = 8 End Sub	By Value Creates a copy of the variable, so changes are not returned.

Figure 5.14A

Two methods to pass data to a subroutine. Pass parameters by value as much as possible to avoid unwanted changes to data.

Most languages also enable you to create new functions. There is a slight technical difference between functions and subroutines. Although subroutines and functions can receive or return data through pass-by-reference parameters, a function can return a result or a single value directly to the calling program. For instance, your main program might have a statement such as $v1 = \text{Min}(x, y)$. The function would choose the smaller of the two values and return it to the main program, where it is assigned to the variable v1.

Summary

The only way to learn how to program is to write your own programs. Reading books, syntax documentation, and studying code written by others will help, but the only way to become a programmer is through experience.

As you write programs, remember that you (or someone else) might have to modify your code later. Choose descriptive variable names. Document your statements with comments that explain tricky sections and outline the purpose of each section of code. Write in small sections and subroutines. Test each section, and keep the test data and results in the documentation. Keep revision notes so that you know when each section was changed and why you changed it.

Part 3

Applications

Building business applications in a database environment begins with creating forms and reports. Most database management systems have tools to help you construct the basic forms and reports. However, Chapter 6 shows you that you have to design and modify forms and reports to make them useful and user-friendly.

Chapter 7 focuses on the concepts of database integrity and transactions. It explores ways of maintaining data quality and preventing common problems that arise with multiple users in a large database.

Chapter 8 shows the additional steps needed to turn forms and reports into a cohesive application. Applications are integrated collections of consistent forms and reports to accomplish specific tasks. They include startup forms, menus, and help systems.

Chapter 9 shows the conflicts between transaction-processing databases and systems designed for analysis. It introduces the issues of transferring data to data warehouses and explores some of the modern techniques for interactive data analysis in large databases.

Chapter 6

Forms and Reports

Chapter Outline

What You Will Learn in This Chapter

- How do users interact with the database?
- What is the difference between a good form and a bad form?
- What common structures are used in forms?
- What are the main steps used to create forms?
- Can form usability be improved?
- What are the basic roles of reports?

A Developer's View

Ariel: Why the concerned look?

Miranda: Well, I finally figured out how to answer those hard questions. But I'm a little worried. Lots of times I got answers, but they were wrong. I have to be really careful with SQL.

Ariel: Oh, I'm sure you'll do fine. You're always careful about testing your work.

Miranda: I suppose it'll get easier.

Ariel: That's the spirit. Now, are you finally ready to start building the application?

Miranda: I sure am. I looked at some of the information about forms and reports. This is going to be easy.

Ariel: Really?

Miranda: Sure. And you know the best part? All the forms and reports are based on SQL. To get the initial forms and reports, all I have to do is build queries to get the data I want. There are even wizards that will help create the basic forms and reports.

Ariel: I always knew that someday people would call on spirits again.

Getting Started

Users should never see the underlying tables. Instead, data is entered through forms that match the business processes and managers explore data through reports and interactive forms. It is a challenge to build forms and reports that match users' needs, are easy to use, and are nice to look at. This is the stage where you get to start building the application.

Introduction

How do users interact with the database? The true power of databases lies in the ability to create business applications. Applications provide a way for users to do their jobs more efficiently—storing and retrieving data through forms and reports. You never want to give users direct access to tables. Imagine the chaos that would result if you ask a user to enter an OrderID, ItemID, price and quantity directly into a SaleItem table. Stop for a second and think about the applications you already use—particularly Web-based that run on browsers, cell phones, and tablets. Almost all of those applications rely on a database to store data. But users are never directly aware of that relationship. They simply focus on the application. Business data is often more complex than simple consumer applications, so the underlying database is more complex, but the principle of emphasizing the user interaction and hiding the actual database is still important.

Forms and reports are an important part of the database application. Designers use them to create an integrated application, making it easier for users to perform their tasks. Decision makers and clerical workers use forms and reports on a daily basis. Years ago forms were used primarily as input devices, and reports were used to display results. Today, forms and reports can be distributed electronically, display data interactively, and can display a variety of outputs. The Internet, and specifically the World Wide Web, is becoming an increasingly popular means of

- Collect data
- Display query results
- Display analyses and computations
- Startup for other forms and reports
- Direct manipulation of objects
    - Graphics
    - Drag-and-drop

Figure 6.1

Basic uses of database forms. It is important to understand the use of a form, since forms designed for data collection will be different from those designed to analyze data.

distributing data as electronic forms and reports. The same design principles used for database forms also apply to the Web.

As summarized in Figure 6.1, forms are used to collect data, display results of queries, display analysis, and perform computations. They are also used as menus, or connectors, to other forms and reports. With the proper devices, forms can used for drag-and-drop or touch-based interactions. With this type of form, users interact visually with a model of the firm.

Reports are often printed on paper, but they are increasingly being created as Web pages for display on the screen. Reports are used to format the data and present results from complex analysis. Reports can be detailed and cover several pages, such as a detailed inventory report. Alternatively, reports can present summary data, incorporating graphs and totals. A common business example would be a weekly sales report comparing sales by division for the past few weeks. The report would generally be presented graphically and would occupy one page.

At this stage in the project, you need to create all of the forms and reports needed by the users. Fortunately, most systems have tools or wizards to help you create rough drafts of forms and reports. You will still spend considerable time improving the layout, formatting data, and establishing a consistent design scheme. But, it is relatively easy to come back later and modify the forms and reports. In many cases, you will want to take an interactive or prototyping approach where you obtain feedback from the user several times and improve the layout and design at each iteration.

This chapter looks at some basic issues in designing forms and reports in terms of human factor elements. It then discusses the most common types or layouts of forms used in business. Common report layouts are covered. The most common form layouts are: (1) Single-row showing one row on a page with detailed layout control, (1) Tabular showing multiple rows at a time in simple columns, (3) Main/subform that combines the first two types for parent-child relationships, and (4) menu forms with buttons and controls to support navigation and connect forms.

The chapter also looks at the basic elements of reports. Most report writers contain an innermost detail section to display rows of data, and then use headers and footers to display column headings and subtotals.

The details of creating forms and reports for a specific DBMS are covered in the associated workbooks, because these tools can be quite different across the various systems. Dealing with local desktop applications, mobile apps, and browser-based Web apps, quickly leads to even greater differences across the tools. The chapter in this book focuses on the overall concepts which provide a powerful structure for understanding the overall goals. The workbooks show how to build actual forms and reports using specific tools.

Two-Minute Chapter

By itself, a database is a useful way to store data, but business users really need applications. Users should never deal directly with tables and queries. Instead, you create forms that are used to enter data in a layout that makes it easy for users to understand. Reports are used to display standard information, including subtotals and charts.

Think about how the process started with design questions. User forms and reports were collected to identify the data needed. These were subdivided into separate tables that can efficiently store and retrieve data. But split data is difficult for users to work with, so now you need to create electronic versions of the forms and reports that put everything back together. Users will enter data into the forms and it will be stored in the appropriate underlying tables.

Four main layouts are commonly used for forms: tabular, single row, main/subforms, and menu forms. Report writers typically define sections including page headers/footers, and grouping sections: header, detail, footer. Controls, such as text boxes, checkboxes, and drop-down lists, are placed on forms and reports to display or collect data for tables. Forms and controls use properties to set visual attributes. They also support events that can activate custom programming code. For instance, when data is entered into a text box, a program might check the data for proper format or compare it to other values. The overriding goal of forms and reports is to make it easier for users to perform their tasks.

Effective Design of Reports and Forms

What is the difference between a good form and a bad form? Designing forms and reports has much in common with creating art. You start with a blank page and choose layout, colors, and design elements. In fact, you might want to take a graphics design class or consult a graphics designer to help with the artistic side. However, you have the additional burden of making the forms and reports easy to use. You also have to ensure that they convey correct and useful information. The most important concept to remember when designing forms and reports is to understand that they are the primary contact with the users. Each form and report must be tailored to specific situations and business uses. For example, some forms will be used for **heads-down data entry**—where touch typists concentrate on entering data without looking at the screen. Other forms present exploratory analyses, and the decision maker will want to examine various scenarios. The features, layout, and capabilities of these two types of forms are radically different. If you choose the wrong design for the user, the form (or report) will be virtually useless. Keep in mind that these forms and reports will be used every day. Good design and attractive forms are critical.

The key to effective design is to determine the needs of the user. The catch is that users often do not know what they need (or want). In particular, they may not be aware of the capabilities and limitations of a modern DBMS. As a designer, you talk with the users to learn what they want to accomplish. Then you use your experience to provide features that make the form more useful. Just be careful to find the fine line between helping the user and trying to sell users an application they do not need.

Human Factors	Examples
User Control	Match user tasks
	Respond to user control and events
	User customization
Consistency	Layout, design, and colors
	Actions
Clarity	Organization
	Purpose
	Terminology
Aesthetics	Art to enhance
	Graphics
	Sound
Feedback	Methods
	Visual
	Text
	Audio
	Uses
	Acceptance of input
	Changes to data
	Completion of tasks
	Events / Activation
Forgiveness	Anticipation and correction of errors
	Confirmation on delete and updates
	Backup and recovery

Figure 6.2

Basic human factors design elements. All designs should be evaluated in terms of these basic features.

Researchers in human factors have developed several guidelines to help you design forms. To begin, all forms and reports within an application (or even within an organization) should be as consistent as possible. Keystrokes, commands, and icons should be used for the same purposes throughout the application. Color, layout, and structure should be coordinated so users can understand the data and context on any form or report. Basing applications on a set of common tasks reduces the time it takes for users to learn new applications. The increasing importance of Web-based applications simplifies design to some extent, because you now have a limited set of tools that are understood by almost all users. Research into **human factors design** has also led to several hints and guidelines that designers should follow when building forms and reports.

Human Factors Design

Figure 6.2 summarizes some human factors design elements that system designers should incorporate in their applications. With current operating systems, the primary factor is that the users—not the programmer and not the application—should always have control. For example, do not expect (or force) users to enter data in a particular sequence. Instead, set up the base forms and let users choose the data entry order that is easiest for them. In this approach, the user's choices trigger various events. Your application responds to these events or triggers by performing calculations, retrieving or storing data, and offering new choices.

Also, whenever possible, provide options for user customization. Many users want to change display features such as color, typeface, or size. Similarly, users have their own preferences in terms of sorting results and the data to be included. Today, most development tools can automatically pick up the color scheme defined on the user's computer. The key is to use those color choices and not override them.

Both the layout (design and color) and the required actions should be consistent across an application. In terms of user actions, be careful to ensure consistency in basic features, such as whether the user must press the Enter key at the end of an input, which function key invokes the Help system, how the tab and arrow keys are used, and the role of each icon. These actions should be consistent across the entire application. This concept seems obvious, but it can be challenging to implement—particularly when many designers and programmers are creating the application. Two practices help ensure **consistency**: (1) At the start establish a design standard and basic templates for all designers to use, and (2) toward the end of the application development always go back and check for consistency.

Always strive for **clarity**. In many cases clarity means keeping the application simple and well organized. If the application has multiple forms and reports, organize them according to user tasks. It helps to have a clear purpose for an application and to make sure the design enhances that purpose. Use precise terminology, avoid jargon, and stick with terms that are used within the organization. If a company refers to its employees as "Associates," use that term, instead of "Employees."

Aesthetics also play an important role in the user interface. The goal is to use color and design (and sometimes sound) to enhance the forms and reports. Avoid the beginner's mistake of using different colors for every form or placing 10 different fonts on a page. Although design and art are highly subjective, bad designs are immediately obvious to others. If you have minimal experience in design aesthetics, consider taking a course or two in art or design. If nothing else, study work done by others to gain ideas, to train your artistic sensibility, and to stay abreast of current trends. Remember that graphics and art are important, and they provide an attractive and familiar environment for users.

Feedback is crucial to most human-computer interactions. People want to know that when they press a key, choose an option, or select an icon the computer recognized their action and is responding. Typical uses of feedback include accepting input, acknowledging changes of data, highlighting completion of a task, or signifying the start or completion of some event. Several options can be used to provide feedback. Visually, the cursor can be changed, text can be highlighted, a button can be "pushed in," or a box may change color. More direct forms of feedback, such as displaying messages on the screen, can be used in more complicated cases. Some systems use audio feedback, playing a musical theme or sound when the user selects a task or when the computer finishes an operation. If you decide to use audio feedback, be sure that you give users a choice—some people do not like "noisy" computers. On the other hand, do not be hasty to discard the use of audio feedback—it is particularly effective for people with low vision. Similarly, audio responses are useful when users need to focus their vision on an external task and cannot look at the computer screen.

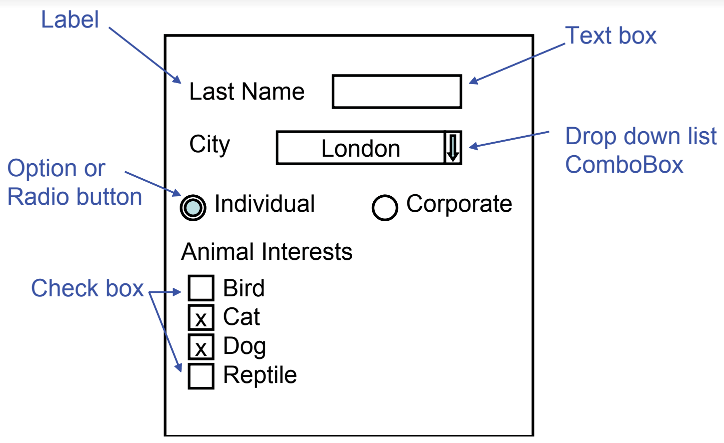


Figure 6.3

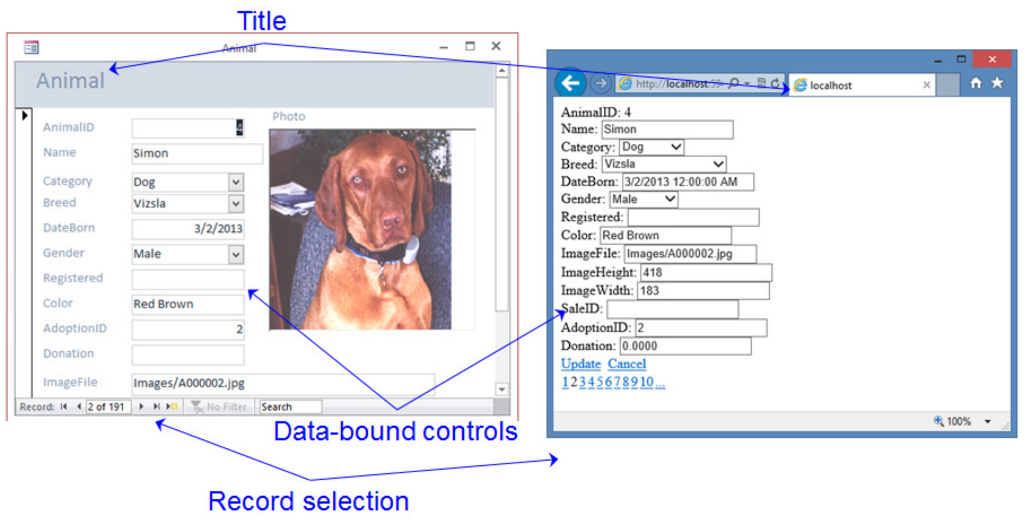
Common form controls. All DBMSs have these basic controls. Labels and text boxes are used the most. Round option buttons are used for mutually exclusive items.

Humans occasionally make mistakes or change their minds. As a designer, you need to understand these possibilities and provide for them within your application. In particular, your application should anticipate and provide for correction of errors. You should confirm deletions and major updates—giving users a chance to verify the changes, or even undo them. Finally, your overall application should include mechanisms for backup and recovery of data—both in case of natural disasters and in case of accidental deletions or loss of data.

Standard Form Controls

Most systems use an object-oriented approach to building forms. The standard form is drawn in a window or Web page that controls the size and provides scroll bars to display sections of the form that do not fit on the page. Some systems allow you to control these page features, but you should almost always leave them alone, so they continue to work the way the users expect. Today, most form-builder tools have adopted a relatively standard approach. Although the tools and options are different, the basic concepts are the same.

Figure 6.3 shows the most common form **controls** that exist. **Labels** are fixed text that is displayed on the form. Some systems attach the label to the data control, but you can generally set its font properties and move it around on the page. A **text box** displays data from a table cell. Generally, the displayed value can be edited or new data can be entered. The text box is a **data-bound** control, which means that you specify the table and column in the database. The form processor automatically retrieves the data from a row and displays the value in the text box. Any changes or new data entered is transferred back to the specified database column. You can set format properties to control the way the data is displayed. You can also specify an **input mask** to limit the type of data that can be entered by the user. For instance, you could specify that only numbers can be entered, or you can automatically format data to a fixed layout. Most systems have sample input masks to handle phone numbers and ZIP codes. However, be careful with input masks. Restricting the user to entering data a certain way might cause problems.



AnimalID	
Name	Simon
Category	Dog
Breed	Vizsla
DateBorn	3/2/2013
Gender	Male
Registered	
Color	Red Brown
AdoptionID	2
Donation	
ImageFile	Images/A000002.jpg

AnimalID: 4
Name: Simon
Category: Dog
Breed: Vizsla
DateBorn: 3/2/2013 12:00:00 AM
Gender: Male
Registered:
Color: Red Brown
ImageFile: Images/A000002.jpg
ImageHeight: 418
ImageWidth: 183
SaleID:
AdoptionID: 2
Donation: 0.0000

Figure 6.4

Form structure. A basic form has a title and some method of selecting new records. The data-bound controls retrieve data from the table and enable users to change and update the database.

Postal codes present a classic example. If your data is going to contain international addresses, you cannot use an input mask. Postal codes outside of America include letters and hyphens as well as digits. Similar problems arise with phone numbers. When you have to include country codes and other extensions, the standard American formatting will not work.

Figure 6.4 shows examples of a simple form to edit Animal data. A basic form has a title and some method to select the data record to be displayed. The data-bound controls then display the data for the specified record and authorized users can edit the data on the form and update the changes to the database. Most forms also have scroll bars when the form is too large to display on one screen window. There is also a mechanism to close the edit form or use a menu to switch to a different view

Commonly used controls include option buttons and check boxes. **Option buttons** are sometimes called radio buttons because on many systems they are drawn as filled circles that look like the knobs on old radios. **Check boxes** are almost always displayed as square boxes using some type of check mark to indicate selection. You need to be careful when choosing between option buttons or check boxes. Most systems are flexible and allow you to use either one for any type of problem. However, the design convention is that option buttons should be used for mutually exclusive events. Users know when they see the round buttons that they can select only one of the items. They also know that they can select multiple items if the square check boxes are used. Follow this convention to avoid confusing users. Option buttons and check boxes are usually data bound controls. Usually, they are bound to columns with yes/no or bit data types. On some systems, they are bound to integer data, where a zero represents the unchecked state.

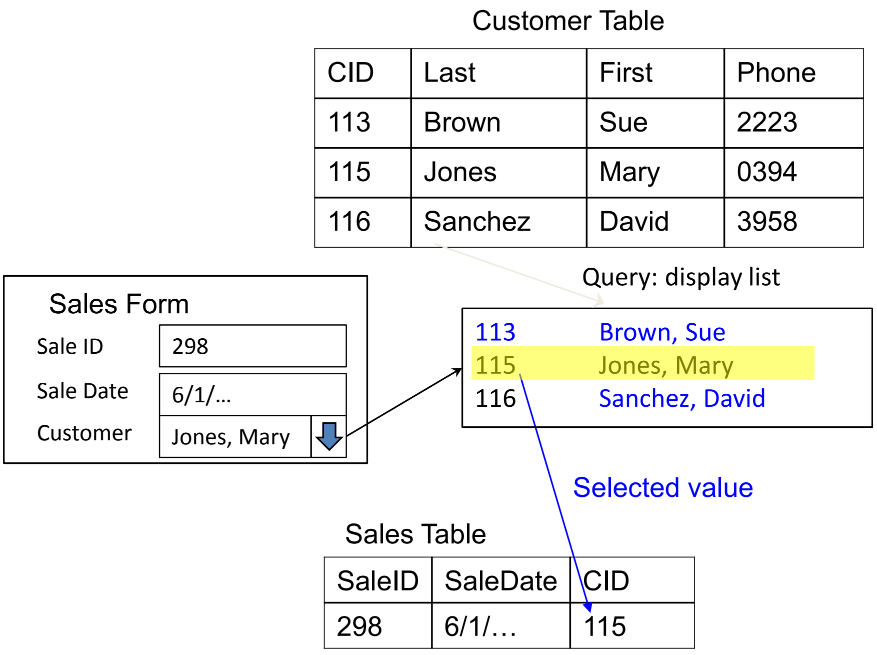


Figure 6.5

Drop down list for foreign key link. The values displayed in the list come from the Customer table. When the user selects one customer, the matching ID value is stored in the Sales table that underlies the Sales form.

The **drop down list** is sometimes called a combo box because it is a combination of a text box and a list box. It is one of the more complicated form controls, but it is extremely useful in relational databases. It retrieves lists of items from a table or query. When the user picks an item from the list, the chosen value (usually an ID) is entered into the main form field. This approach can be useful for lookup lists. For instance, users can pick a city from a preset list and store the CityID value as a link. In addition to saving user time, lookup lists ensure that data is always entered consistently—without abbreviations and typographical errors.

Drop down lists are also commonly used on referential links across tables. Figure 6.5 illustrates the process for a typical sales form. Recall that the Sales table contains the CustomerID column, and has a foreign key relationship to the Customer table. You cannot expect the sales clerks or customers to memorize ID values. So, you add a drop down list that displays a list of customers retrieved from the customer table. When the clerk chooses a customer, the matching ID value is placed into the underlying Sales table. To define the drop down list, you have to specify the table or query that will provide the list of data. You have to specify a display column and a key (ID) column. You often create a query to generate the display column, such as appending first and last names. You also have to data bind the drop down list to the CustomerID column in the target Sales table. This step ensures that the chosen value is stored in the proper column.

Because relational databases tend to have many foreign keys, you will find many opportunities to use drop down lists. However, in a Web environment with relatively slow network links, drop down lists can cause problems. It could take a long time to transfer thousands of rows of data to display in the list. Even with a fast network, a drop down list with thousands of entries can be difficult to use. Users may find it difficult to scroll through thousands of entries in the list. You might need to implement some type of pop-up box to fill the list. Users can enter a simple search condition—such as the first letter of the name—to restrict the display list to a smaller set of entries that is faster to load and easier to scroll through.

Most systems have several other controls. For example, you can add **command buttons** and write code to open other forms or reports or perform some custom action. You can add background images, icons, or draw lines, rectangles, and circles. These items are generally decorations placed on the form and not bound to the database. Most systems have a control that stores and displays images or other binary objects in a data table. For example, you could store a photograph of each employee or product directly in the database.

User Interface—Events

Most systems treat forms using an **event** model. Each form (and control) can generate different events. You can write code to perform a custom action when some event is triggered. For example, opening and closing a form are basic events for every form. You can control how the form operates by creating actions that are taken when each event occurs. Many of the events involve the individual controls. One of the most common events to control is the click event when a button is clicked or submitted.

Although a form can have multiple controls (e.g., text boxes) on it, only one control at a time has the focus. The control that has the **focus** will receive keystrokes entered by the user. This control is often highlighted with an outline or a different color. The same concept applies when an application displays multiple forms on a screen. Only one form has the focus at a time. Within a form, users can usually use the Tab key to move to the next control in a sequence, which is known as the **tab order**. You must be sure to check the tab order on all forms so that it visually matches the layout of the form. In some cases, you might even want to handle the tab or exit event. When a user leaves a control, your code can perform additional calculations or assign default values.

By handling the events with custom code, you can make the form respond to user requests and to handle common tasks automatically. With dozens of controls and dozens of possible events per control, the challenge is to choose exactly which event is needed to perform a specific task.

User Interface—Accessibility Issues

One of the greatest strengths of the Windows and Web interfaces is its graphical orientation—which makes it easy for people to perform complex operations with a few moves of the mouse or selections on the screen. One of the drawbacks to this type of interface is that it is more difficult to make a system that is **accessible** to users facing some physical challenges. As a designer you can make your applications accessible to a wider base of users. To begin, your application should accept multiple sources of inputs. Do not rely on just a mouse or a pointer but also use the keyboard, and increasingly, the user's voice. Similarly, it is helpful if your application can provide multiple types of output. Increasingly, you should consider how to integrate sound and voice output. The user must also be able to set the color and size of the output.

- Language and characters
- Currency
- Time zones
- Time and date formats
- Number formats
- Country names and maps
- National ID numbers—privacy

Figure 6.6

Common internationalization issues. Applications that will be used by people in different nations need to adapt to national conventions as much as possible.

Microsoft guidelines provide some suggestions for making your applications accessible to more users. Detailed ideas and current developments can be found on its Web site. Human factors experience with other applications has generated some specific suggestions. For example, do not use red-green color combinations. Approximately 10 percent of the U.S. male population experiences some difficulty distinguishing between red and green. Try to pick high-contrast colors that most people can distinguish (e.g., black and white, yellow, blue, and red). When in doubt, ask people to test your color combinations. Better yet, let users select their own colors, or use the system color scheme that is configured on the user's computer.

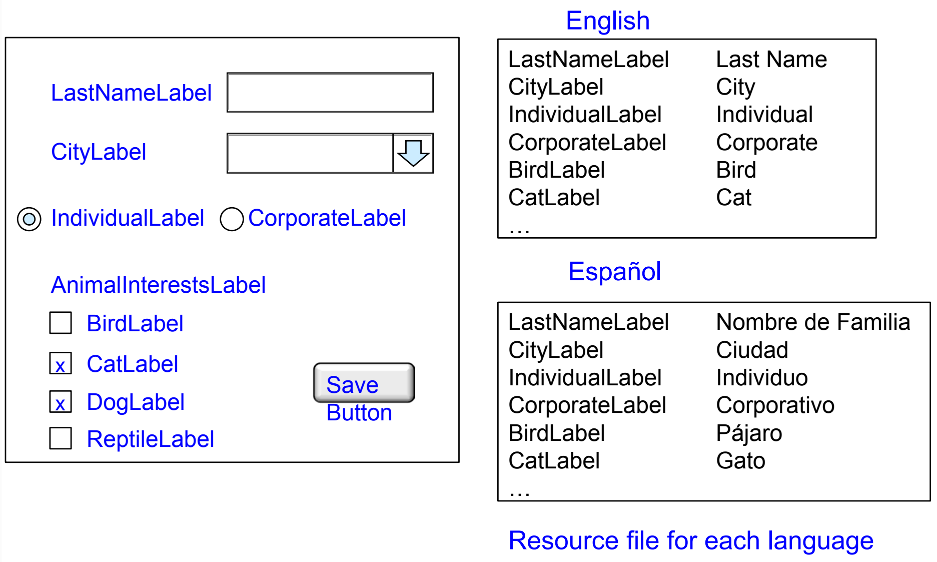
Second, avoid requiring rapid user responses. Do not put time limits on input. Although it might be fun in a game, many users have slower data entry skills. Some designers include pop-up messages to check on user progress after a delay in data entry. These messages are usually pointless and can be annoying. With modern screen-saver security systems, users can set their own delay controls and messages.

Third, avoid controls that flash rapidly on the screen. They tend to annoy most users. Worse, certain flash rates have been known to trigger epileptic seizures in some people. An interesting situation arose in Japan at the end of 1997 when a sequence of flashing graphics on an animated television show (Pokemon) sent about 700 children to the hospital.

Fourth, as much as possible, enable users to customize their screens. Let them choose typefaces, font sizes, and screen colors. That way, users can adjust the screen to compensate for any vision problems they may have. And if you use sound, let people control the volume, even pitch, if possible. In many cases, the Windows environment provides much of this functionality, so the key point is to avoid overriding that functionality. Also, you should test your applications on various computers. Some video systems may distort your choice of colors or will be incapable of displaying your forms at the desired resolution.

User Interface—International Environment

Today, you must build your forms and reports with the understanding that people from around the world will use them. Figure 6.6 summarizes some of the common issues you have to consider when building forms and reports. Language is the most obvious, and it can be the most difficult. First, all of your text data needs to be stored using **Unicode** characters. With many DBMSs, you need to specify



English

LastNameLabel	Last Name
CityLabel	City
IndividualLabel	Individual
CorporateLabel	Corporate
BirdLabel	Bird
CatLabel	Cat
…	

Español

LastNameLabel	Nombre de Familia
CityLabel	Ciudad
IndividualLabel	Individuo
CorporateLabel	Corporativo
BirdLabel	Pájaro
CatLabel	Gato
…	

Resource file for each language

Figure 6.7

Multiple languages. All text on a form is dynamically stored as a resource variable. A resource file is created for each language and it holds the translated entry for every control.

nvarchar instead of varchar. With Unicode characters, names, addresses, and other data elements can be stored in any language.

Dealing with multiple languages on forms and reports is more challenging, but most development systems have methods to simplify the task. Your first thought might be to create every form multiple times: once for each desired language. Abandon that thought early. It would be extremely painful to alter a form if you had to change it for every different language. Instead, you specify the layout and design of the form once. Then you make all the text items (titles, labels, buttons, and so on) dynamic. The actual method depends on the tools you are using. As shown in Figure 6.7, generally, you give each item a name and declare it in a resource file. You then create a resource file for each supported language for each form or report. For example, the Sales form might have resource files for English, Spanish, French, German, and Japanese. The resource file contains simple pairs: the item name and the text translated to the desired language. Of course, you have to find someone to translate all of the items. When the form is generated for the user, the system chooses the appropriate language resource file and displays each text item in the user's chosen language. You create a form only once, but it can display the text in any language with a matching resource file. You can simply ship the resource file to a translator, and all of the work is stored in one location. You will probably want to include copies of the forms so the translators can see the context to create a better translation. You also have to test the form with each language, particularly non-Latin languages, to ensure the translated words still fit correctly on the form and do not overlap other controls. Once you have developed the dynamic form page and set up the resource files, it is relatively easy to sup-

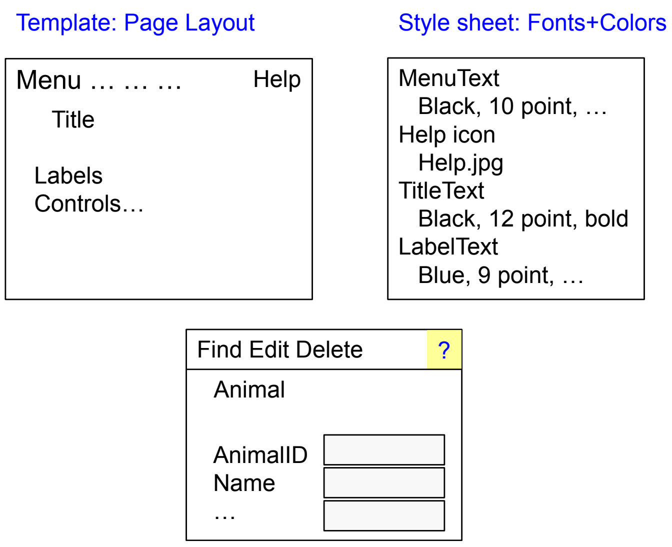


Figure 6.8

Style sheets and templates. Templates or master pages define the basic page layout and common controls. The style sheet sets colors, backgrounds, fonts, and so on for all items on the form. Ideally, style sheets are dynamic, so changes to a style are automatically picked up on every existing form.

port additional languages—simply have each resource file translated to the new language.

Data formats for currency, dates, and numbers also vary by region or nation. For instance, the U.S. uses commas to separate thousands, and a decimal point to separate fractional values in a number. Most European nations reverse these two, using dots to separate thousands. Dates can cause serious confusion if you use the shortened numeric format. In the United States, 1/5/2008 represents January 5, 2008; while the same short date in Europe indicates May 1, 2008. With Windows-based applications, you can query the regional settings on the user's computer to find the appropriate format for numbers and dates. However, it is probably safest to avoid the short-date format and use a format such as 05-Jan-2008, which is clearly understood by almost everyone. If you do retrieve local settings from the user's computer, be extremely careful about currency values. Picking up a local currency symbol does not convert the values. Serious problems could result if an American enters $100.00 into the database, but the system displays it as £100.00 to someone in England.

The issue of country names can become a problem, or even an international incident, if you are using a geographic information system or mapping program. Several areas around the globe are politically disputed or face different claims (sometimes even calling an area "disputed" might anger the participants). You need to have the correct map for each region. Country names also vary by language, but you can generally pick one language and use the names defined by the International Organization for Standardization (ISO) or the United Nations.

Finally, be careful to abide by all national laws with respect to security and access to the data. For example, many European nations have strong privacy statutes—particularly with respect to customer data and national ID numbers. In some cases, it may be illegal to transfer data collected in Europe to the United States.

Style Sheets and Templates

Consistency quickly becomes an issue for large projects—particularly when multiple developers are working on the forms and reports. It is important that all of the forms and reports use the same color schemes and as much as possible, place common items in the same location. For instance, links for help, search, or close should be the same across all forms and reports. Clearly, designers need to specify the layout details before forms and reports are built. But even then, it is difficult for developers to remember the exact specifications. Consequently, two tools are useful to help with consistency: style sheets and templates. Some systems combine the two topics, some give them different names (such as master pages), and others do not support them at all.

Figure 6.8 shows how style sheets and templates influence form development. Templates or master pages define the page layout and hold the controls that are common to every form. In the example, the template defines the location for the main menu, a Help icon, and the page title. The style sheet defines the colors, fonts, icons and so on that set the image for the page. Ideally, every item appearing on a form will be named and defined in the style sheet. Developers simply select the appropriate style when a page is created.

With some systems the style sheet is dynamically attached to each page. For example, Web browsers use the page definition and the style sheet to determine how to display a page. If the style sheet is changed, the Web browser automatically uses the latest values when displaying a form or report. Consequently, designers can alter the page style and colors at any time—even after the application has been completed and deployed.

Form Layout

What common structures are used in forms? Individual forms are your primary means of communicating with people who use your application. Forms are used to collect data, display results, and organize the overall system. From a database perspective, your application is built from several standard types of forms. As you begin working with these basic layouts, keep in mind that you can create complex forms that use features from several different form types. However, you should understand the layout and uses of each individual form type first.

As summarized in Figure 6.9, you will be working with four basic types of forms: (1) **tabular forms**, which display data in rows and columns, (2) **single-row forms**, which show data for one row at a time and in which the designer can arrange the values in any format on the screen; (3) **subform forms**, which display data from two tables that have a one-to-many relationship; and (4) **startup forms**, or menus, which direct the user to other forms and reports in the application.

You can think of a subform as a combination of a single row form and a tabular form. The Sales form is a classic example. The Sales table forms the foundation of the main form, and the SaleItems table data is handled in the subform. Because each sale can contain many items being sold, you need the subform to handle the repeating section.

Form Type	Common uses
Tabular Multiple rows of data.	Lookup lists or tables with a limited number of columns when it is useful to see several rows of data at a time.
Single row One row of data at a time.	The most common type. Provides complete control over page layout and space for many columns and links.
Subforms Combine row and details.	One-to-many relationships. Repeating section shows data related to main form. The items section of a sales form is a typical example.
Switchboard or startup Customized with buttons.	A designed form that is used as the main menu or switchboard to the other forms and reports.

Figure 6.9

Form layout. Once you understand the single row and tabular (repeating) forms, you can create more complex forms by creating subforms. The startup form is similar to a menu that links the forms and reports together.

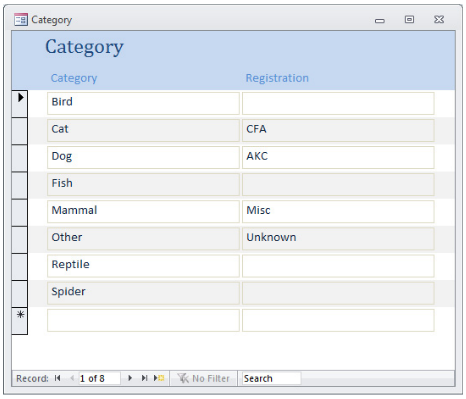
Tabular Forms

As shown in Figure 6.10, one of the simplest forms is the tabular form, which displays the columns and rows from a table or query. Some systems provide variations on the tabular form, such as the datasheet in Microsoft Access. The tabular form is used as the main form with a limited number of columns, and when users need to see multiple rows at the same time. It is particularly useful for administrative forms, such as editing values in a lookup table.

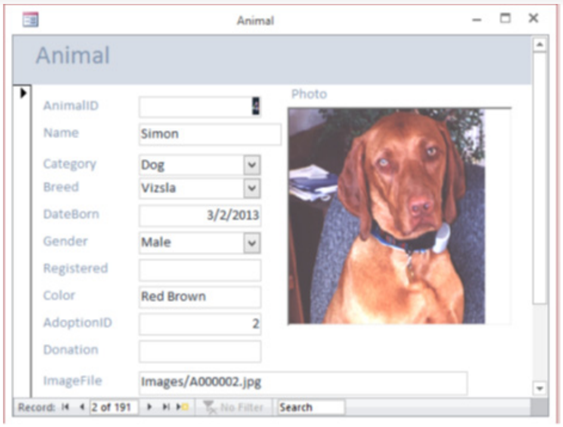
The primary feature of the form is that it displays multiple rows of data for editing. Consequently, users can see and compare data across several items. It is useful for short lists of data. It can cause problems when the table to be edited is

Figure 6.10

Sample tabular form. You define the controls for one row, and the DBMS displays data for all of the rows in the query. Scroll bars enable the user to see more rows (or columns of data).


Category

Category	Registration
Bird	
Cat	CFA
Dog	AKC
Fish	
Mammal	Misc
Other	Unknown
Reptile	
Spider	


Animal

Animal

AnimalID	
Name	Simon
Category	Dog
Breed	Vizsla
DateBorn	3/2/2013
Gender	Male
Registered	
Color	Red Brown
AdoptionID	2
Donation	
ImageFile	Images/A000002.jpg

Figure 6.11

A simple single-row form. This form displays data for one row at a time. You have substantial control over layout through color, graphics, and command buttons. The navigation buttons on the bottom enable the user to display different rows.

too large—both in terms of number of rows and too many columns. If the user has to scroll repeatedly to find a specific entry, a tabular form will not work very well.

Tabular forms are commonly used as subforms, where they collect and display a limited list of data that is related to the main form. For example, an order form often contains a subform tied to the OrderItem table to display the list of items being purchased on the currently displayed order.

Single-Row or Columnar-Forms

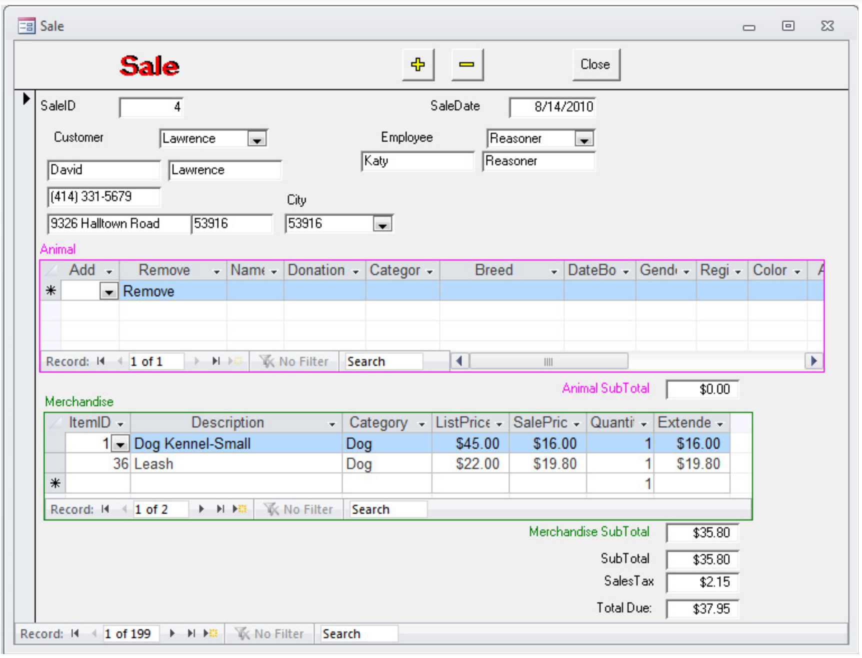
A single-row form displays data for one row at a time. The goal is to display every column on one screen. Its greatest feature is that the designer can display the data at any location on the form. It is useful for designing a form that looks like a traditional paper form. The designer can also use color, graphics, and command buttons to make the form easier to use. As illustrated in Figure 6.11, this form design requires navigation controls that enable the user to scroll backward and forward through the rows of data. Common navigation controls also include buttons to go to the first and last rows and to go to a particular row of data.

In general, you will want to include a Find command that enables the user to locate a particular row of data—based on the values in some row. For example, a form displaying customer data should have a search option based on customer name. Similarly, the user will often want to sort the rows in different orders.

The single-row form is generally the most-used form layout. With careful design you can use it to display substantial amounts of data. By including subforms, you can highlight relationships among various pieces of data and make it easy for users to enter data. You can also include charts to help users make decisions.

Subform Forms

A subform is usually a tabular form embedded on the main form. A subform generally shows a one-to-many relationship. In the example in Figure 6.12, a sale could include many adopted animals, so you need a subform to display this re-



Sale

Sale

Close

SaleID	4	SaleDate	8/14/2010
Customer	Lawrence	Employee	Reasoner
David	Lawrence	Katy	Reasoner
(414) 331-5679		City	
9326 Halltown Road	53916	53916	

Animal

Add	Remove	Name	Donation	Categor	Breed	DateBo	Gend	Regi	Color
	Remove								

Record: 1 of 1 No Filter Search

Animal SubTotal $0.00

Merchandise

ItemID	Description	Category	ListPrice	SalePric	Quanti	Extende
1	Dog Kennel-Small	Dog	$45.00	$16.00	1	$16.00
36	Leash	Dog	$22.00	$19.80	1	$19.80
					1	

Record: 1 of 2 No Filter Search

Merchandise SubTotal	$35.80
SubTotal	$35.80
SalesTax	$2.15
Total Due:	$37.95

Record: 1 of 199 No Filter Search

Figure 6.12

Subform example. The main form is based on the Sale table, which has a one-to-many relationship with the SaleAnimal table used on the subform. The subform contents are linked via the SaleID. The two subforms in this example are independent.

peating list. The main form must be a single-row form, and the subform should be a tabular view. The Sale example for the Pet Store is more complex than for other companies because it uses two subforms. Animals are treated as separate objects from merchandise, so each is recorded in a separate subform.

If you look at the underlying tables, you will see that SaleID links the main form (based on Sale) and merchandise subform (based on SaleItem) to each other. You will rarely display the linking column on the subform. In general, doing so would be pointless, since the linking column would always display the same value as the related column on the main form. Think about what that means for a minute. The Sale table has a SaleID that is generated by the DBMS when a new sale is created. The SaleItem table also has a SaleID column, and every animal sold must contain the same SaleID value from the main form. Yet it would be painful for the clerk to reenter the SaleID on the subform for each animal that is sold. By using the subform and specifying the SaleID as the link (Master and Child property), the DBMS automatically enters the main SaleID into the table for the subform.

Most database systems enable you to create forms that have multiple subforms. The subforms can be either independent—as separate boxes on the main form—or nested—where each subform lies inside another. In most cases, you will want the parent forms to be single-row forms. However, some systems support tabular lists for both the main and the subforms. In most applications, users would find this

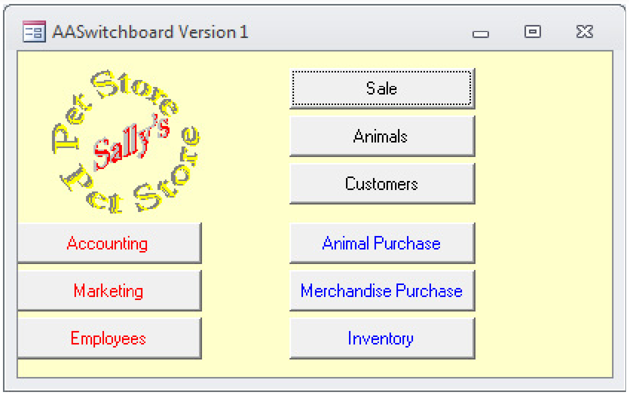


Figure 6.13

Sample startup form. The buttons match the user's tasks.

approach disconcerting: They would first have to select a single row in the parent form and then deal with the matching list in the subform.

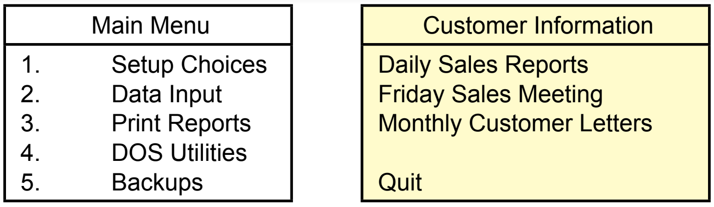
Startup Forms

Startup or menu forms provide the overall structure to an application. They are straightforward to create, although you may want the assistance of a graphics designer. The startup form often contains images, and the design reflects the style of the company. Startup forms are not required in all applications—in many cases the same functions can be provided through menus or the newer ribbon bars.

You begin with a blank form and remove any scroll bars and navigation controls. Pictures can be inserted as background images or as individual controls that can be used as buttons to open another form. As shown in Figure 6.13, command buttons or links are the most important feature of the startup form. When the user selects a button, a corresponding form or report is opened. The main startup form will be used quite often, so you should pay careful attention to its design. In Web-based applications, the buttons are usually replaced with hyperlinks to the desired form or report.

The key to a successful application begins with the startup form—not just its design but also its content. The startup form and links on other forms create the application flow which should match the user's tasks. One approach is to first identify the user and then provide a selection of buttons that matches his or her tasks. Consider a simple example. A manager needs to print a daily sales report of best-selling items. Every week the DBMS must print out a list of total sales by employee. The firm also sends letters to the best customers every month offering them additional discounts. A secretary will be in charge of printing these reports, so you create a simple menu that lists each report. The secretary chooses the desired report from the list. Some reports might ask questions, such as which week to use. The secretary enters the answers, and the report is printed.

The first step in creating an application is to think about the people who will use it. How do they perform their jobs? How do the database inputs and reports fit into their job? In object-oriented design terminology, each situation is called a **use case**. The goal is to devise a menu system that reflects the way they work.



Main Menu	
1.	Setup Choices
2.	Data Input
3.	Print Reports
4.	DOS Utilities
5.	Backups

Customer Information
Daily Sales Reports
Friday Sales Meeting
Monthly Customer Letters
Quit

Figure 6.14

Designing menus for users. Which menu is easier for a secretary to understand? When designing applications, you should organize the application to match the processes users perform.

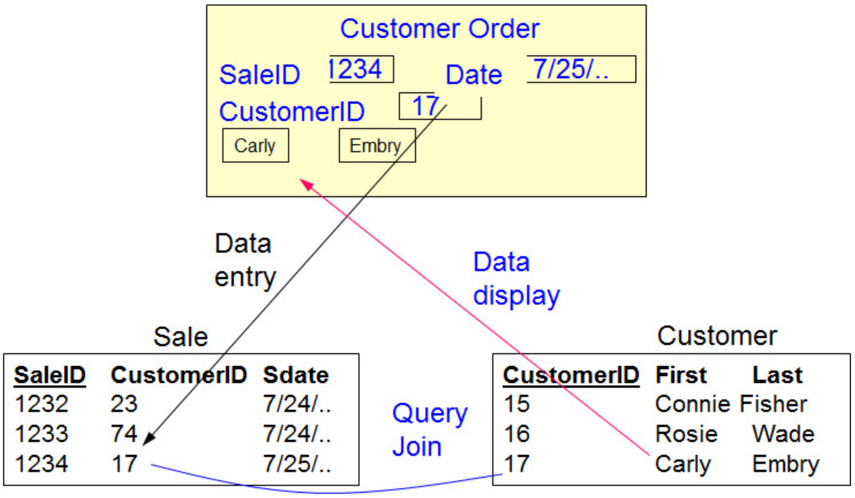
Two examples of a first menu are shown in Figure 6.14. Which menu is easier for a clerk to understand? Answer: The one that best relates to the job. Once you understand the basic tasks, write down a set of related menus. Some menu options call up other menus, some print reports, and others activate the input screens you created.

Creating Forms

What are the main steps used to create forms? The first step in creating a form is to be sure that you understand its purpose and how it will be used. Its usage dictates the specific data that needs to be displayed. Once you know the data needed, you can identify the database tables that hold that data. One important point to remember: A form should only attempt to update data to one table at a time. For example, a common sales form might display data about the Sale, the Customer, the SaleItems selected, and perhaps detailed data about the individual Products. Through the design process, this data needs to be stored in four related tables, so how can you create a form that updates only one table?

The answer to this question actually determines the characteristics of many database systems. The reason you can put only one table on a form is that multiple tables make it difficult for the form to understand exactly what the user is attempting to do. For instance, if the main Sales form contained all columns from the Sales table as well as the Customer table, what does it mean to add a new row? Should that row be added to the Sales table or the Customer table?

The process is complicated when you want to display data from multiple tables. For instance, you probably want to display a customer's name and phone number on a sale form. Depending on the DBMS you are using, you could have several options. You might be able to display the customer data on the sale form without needing to edit it. In this case, you could add a button or hyperlink to open the customer form with the corresponding data available for editing. Alternatively, you might be able to create sections within a form, where each section can hold data for a new, linked table. One of these sections could be a subform that contains repeating rows of data linked to the main form. Depending on the database system, you might be able to use an updateable query to display data from multiple tables.



SaleID	CustomerID	Sdate
1232	23	7/24/..
1233	74	7/24/..
1234	17	7/25/..

CustomerID	First	Last
15	Connie	Fisher
16	Rosie	Wade
17	Carly	Embry

Figure 6.15

Basing the Order form on a query. The query contains all the columns from the Sale table and some columns from the Customer table. The query must never include the CustomerID column from the Customer table, which is the column used to join the two tables.

Updateable Queries

The issue of multiple tables on a form is related to the problem of updateable queries. If a system can support queries that use multiple tables as updateable, then it is possible to put carefully selected columns from multiple tables on a single form. Figure 6.15 shows a common Sales Order main form. The Sale table holds the CustomerID as a foreign key. To record a sale, a salesperson simply needs to enter the ID number of the appropriate customer. But it is not easy to remember numbers, and it would be nice to display the matching customer data on the main form to verify the name and address.

Technically, a query could be built that includes all of the columns from the Sale Order table along with some of the descriptive columns from the Customer table. To remain updateable, the query must not include the primary key (CustomerID) from the Customer table. Figure 6.16 shows the basic query. One potential

Figure 6.16

Updateable query for Sales form. New rows will be entered into the Sale table, but some systems will also support updates to the three columns from the Customer table. Note that you should never include the Customer.CustomerID column in the SELECT statement.

```
SELECT Sale.SaleID, Sale.SaleDate, Sale.CustomerID,
  Customer.LastName, Customer.FirstName, Customer.Phone
FROM Sale
INNER JOIN Customer
ON Sale.CustomerID=Customer.CustomerID;
```

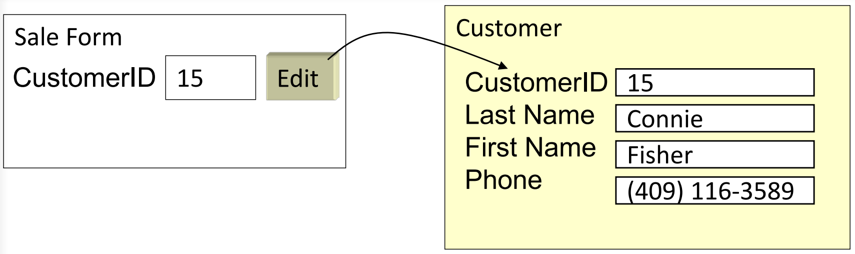


Figure 6.17

Linked forms. The Edit button opens the Customer form and loads the data that matches the current value of CustomerID on the Sale Form.

drawback to this approach is that whenever a new CustomerID is entered into a sale, the form must make a trip to the database to look up the matching customer data. Consequently, some forms systems do not support this approach. In particular, it can cause problems on Web-based forms because of the delay in sending data to the server and waiting for a reply.

Linked Forms

Another approach to the problem is to use linked forms. As shown in Figure 6.17, these forms might be displayed separately, or they might be separate sections displayed on a single form. This latter approach is used for parent/child forms, where the child form contains data from a second table that has a one-to-many link to the parent form. Conceptually, displaying a related table on one form versus in a separate window is equivalent. The main difference lies in the amount of screen space needed. If you put several sections on one form, users will need large screens to see all of the data. By using multiple windows, the user will not be able to see all of the data at one time but can switch between the windows to see and edit the data.

Linked forms work by using a query to match the data displayed in the secondary form to a key value from the original form. For instance, from the main Sale Order form, the CustomerID can be used to display the matching customer data in a linked form. Similarly, a SaleItem subform can display the rows that match the SaleID from the main form. Each related portion of the data can be displayed in a separate form or region. Ideally, the forms system will have a method to automatically perform the linking and retrieve the matching data; otherwise, you will have to write customized code to modify the underlying query and refresh the data as needed.

Properties and Controls

Most software packages are built using an object-oriented approach. With an OO design, each object has properties that describe it and methods or functions that it can perform. Objects are also closely tied to an event-driven system, where user actions and changes can trigger various events. Most database forms follow this methodology and contain dozens of objects to create and enhance forms. Your job is to assign properties and write short programs to respond to various events to make the application easier for users.

Category	Sample Properties
Data	Base table / query Filters Sort
Integrity	Edits Additions, deletions Locks
Format	Caption Scroll bars Record selectors Navigation buttons Size and centering Background/pictures Colors Tab order
Other	Pop-up menus Menu bar Help

Figure 6.18

Basic properties for forms. At a minimum, set the data source and basic format properties. Additional properties ensure consistency, protect data, and make the form easier to use.

As highlighted by the abbreviated list of properties in Figure 6.18, form and control properties can be grouped into primary categories. The first category (data) relates to the source of the data, where you set the base table or query. You can set filters to display only the data rows that meet a specific condition. You can also specify the sort order and a WHERE clause directly in the underlying query, which normally would be a more efficient approach. This step essentially binds the control element to the database.

A second set of properties refers to data integrity to help you control the type of editing and changes allowed on the form. For example, you might set the properties for individual users so that some users cannot add or delete data using a particular form. However, keep in mind that it is generally safer to set these conditions in SQL so that they apply throughout the database, and not just on one form. For example, sales clerks should probably be prevented from adding new suppliers.

A third level of properties controls the display of the form. Everything from the caption to scroll bars, form size, and background are set by display properties. Again, remember that consistency is a virtue. Before beginning a project, choose a design template and standards; then set all form and control properties to meet that standard.

Controls on Forms

The standard form controls are supported by every forms-development tool. Figure 6.19 shows some of the controls available in Visual Studio (2010). Many systems have wizards that quickly create standard labels and text box controls for

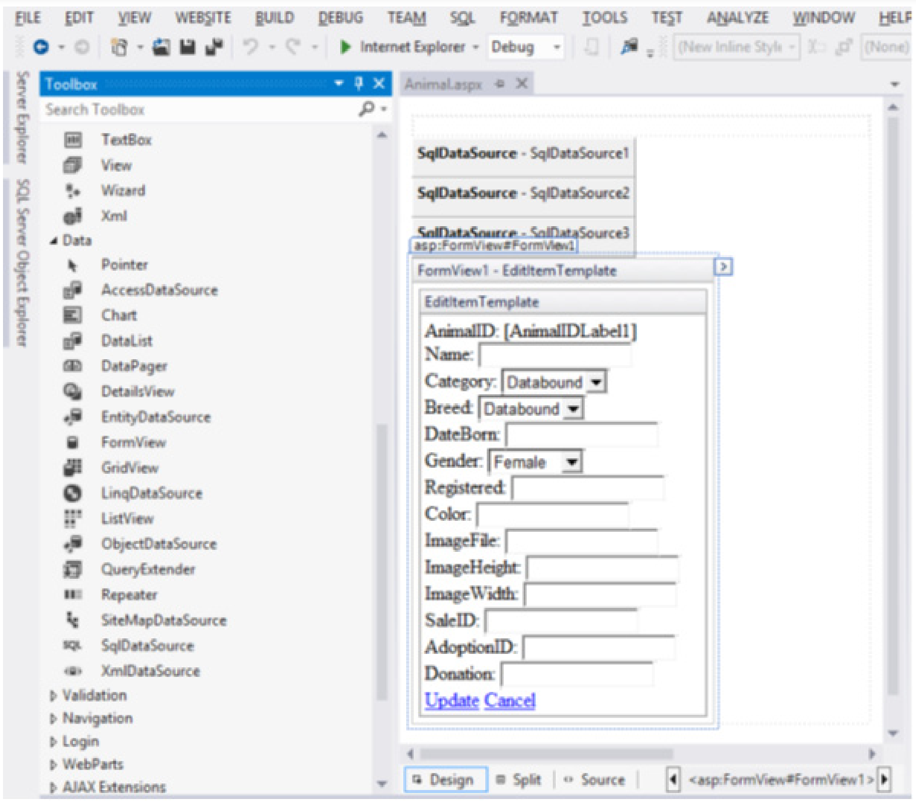


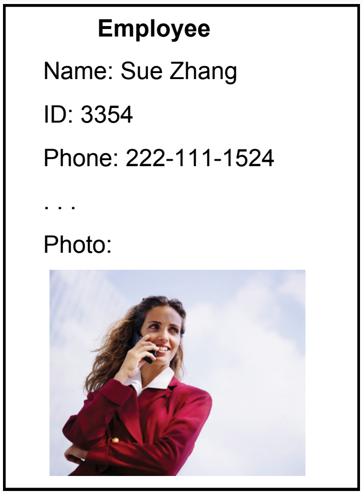
Figure 6.19

Some controls in Visual Studio. The controls are grouped into categories to make them slightly easier to find. You drag a control onto the form and assign its properties. You can buy additional controls or write your own.

desired columns. Each control needs to be given a name and bound to the appropriate data column. Be careful with the name; once you have set it, it is difficult to change. You need to pick a meaningful name when you first create the control. If you try to change it later, you will have to find every control or program that refers to that control—which can be a time-consuming and error-prone task. The name is used by other controls and program code (much like a variable name) to retrieve and store data on the form.

Some people name controls based on the type of control. For example, the name of a label control would end with the word Label, such as AddressLabel. Alternatively, some developers precede the name of the control with its type, such as lblAddress. The difference affects the sort order of the controls. If you end with the "Label" notation, control names sorted alphabetically will group by the data name (AddressLabel, AddressTextBox). If you start with the "lbl" notation, all of the labels will be listed together (lblAddress, lblPhone). You need to decide at the start how you want to locate and remember the control names.

Once the base controls are set, you will have to set formatting, and rearrange the controls on the page. You will also have to pay attention to the layout of controls on the page. Generally, you can drag the controls and labels to new positions, but be sure to use the alignment tools to match the edges.



Employee

Name: Sue Zhang

ID: 3354

Phone: 222-111-1524

. . .

Photo:

Figure 6.20

Image bound to a data column. Employee photo is scanned and stored in the database column.

Graphics Features

Occasionally you will want to add graphics features to your form. There are controls to add pictures, as well as simple lines and boxes. Lines and boxes are often used to create a three-dimensional effect for other controls by adding shading or highlighting.

To display an image from a table as shown in Figure 6.20, you must first define an Object or image column within that table. Then the bound object frame is used like a text box to position and display the image on the form.

To use an image or texture as a background, first use a graphics package to make sure the image is light enough to not interfere with the readability of the other boxes. Then set the Picture property of the form to the name of the image file. In most cases, you want to embed the image on the form so the picture is included directly with the database.

For Web-based applications, it is easier to store images as files on the server—instead of trying to store them within the database itself. Web pages ultimately use HTML controls to display pages to the user. Web servers and browsers know how to retrieve and display images from files. Hence, it is straightforward to store the image filename in the database and embed that name in the HTML so it is retrieved automatically. If you store the actual image within an object control in the table, you need a special program to convert the page HTML request into a database query to retrieve the image and deliver it to the browser. Besides the extra work, storing images in a database table quickly eats up storage space. And if you want to use the free versions of the commercial DBMSs, you need to hold down the size of the database.

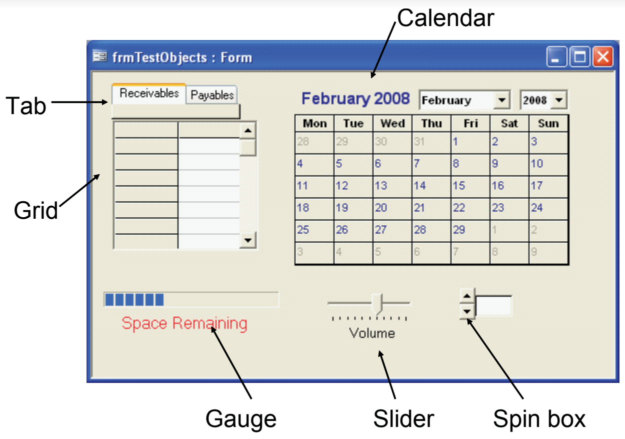


Figure 6.21

Additional controls. Thousands of controls are available to improve the user interface or perform specialized tasks. Common controls include tabs, grid, calendar, gauges, sliders, and the spin box. You can also create custom controls.

Complex Controls

Additional control objects can be created using a variety of computer languages or purchased from commercial vendors.

A few additional controls are shown in Figure 6.21. The Tab and Calendar controls are particularly useful in business applications. There is also a Grid control that enables you to display data in a spreadsheet layout. These types of controls are not as easy to use as the standard bound controls. The developer has to write short programs to load data into the control and to respond to the control's events. Note that the Calendar control was removed from Microsoft Office 2010. The date picker is available for text boxes that are formatted as dates. If you want calendars for other purposes, you will have to find or build a new one—or switch to Visual Studio.

Charts

Database applications used for making decisions often contain charts or graphs. Charts are another type of control that can be placed on a form (or report). The first step in creating a useful chart is to discuss with the user exactly what type of data and what type of chart will be needed. Then you usually build a new SQL query that will collect the data to be displayed on the chart. The chart control places the chart on the form and specifies the individual attributes (like type of chart, axis scale, and colors).

Two basic types of charts are used on database forms and reports: (1) graphs that show detail from the currently displayed row and (2) graphs that display summary data across all (or several) of the rows. The difference between the two approaches lies in the level of data displayed. Detail graphs change with each row of data displayed. Summary graphs are usually generated from totals or averages.

Figure 6.22 illustrates the two types of graphs as they might be used in the Pet Store database. Each chart shows the amount of money spent on animals versus the amount spent on merchandise. However, the top set of charts shows the split

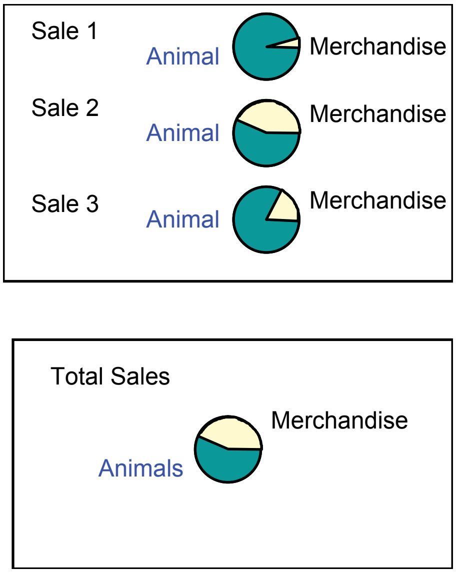


Figure 6.22

Charts on forms or reports. The top charts show the split in sales for each individual sale and will change with each row of data. The bottom graph shows the sales split for all sales and is not bound to an individual row.

for each individual sale, so the graphs vary with each row in the Sale query. The bottom chart shows the overall total for the store—even if it is placed on a Sales form that shows each row of data, it will not change (except over time). To create the two types of charts, the main difference is in the query. The query for the detail charts contains a column (SaleID) that is linked to the row of data being displayed on the form (based on its SaledID). The summary graph computes the totals across all of the sales and is not linked to any particular sale.

Multiple Forms

As you can guess, an application will quickly spawn many different forms. Of course, the forms should be linked to each other so users can quickly move between the forms by clicking a button, data value, or image. Startup menu forms play an important role in tying forms together. However, you can also connect forms directly. The most common example is the use of subforms placed on a main form. In this situation the forms are linked by setting the Master and Child properties of the subform. Then the database system keeps the data synchronized so that when the user selects a new row in the main form, the matching rows in the subform are located and displayed automatically.

When the forms contain related data, another approach is to build a link that opens the second form based on the ID value in the first form. For example, if the Order form contains customer data, when the user clicks an Edit button (or double-clicks on the customer name), the application should open the Customer form. The Customer form should display the data that corresponds to the customer on the Order form. The technique for linking forms varies by DBMS. The accompanying workbooks provide the syntax and examples needed for each DBMS. In the

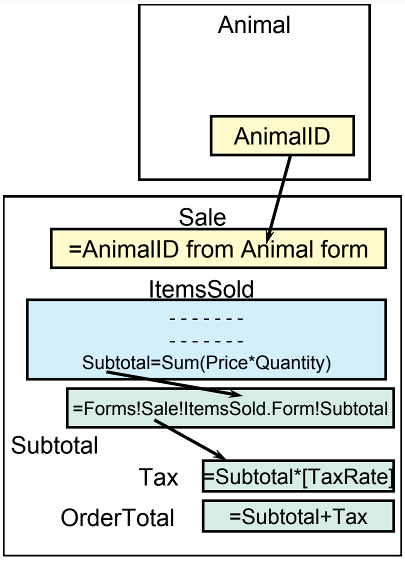


Figure 6.23

Copying data from a different form. The default AnimalID is copied into the Sale form from the Animal table. Likewise, the subtotal is first computed on the subform and then copied to the main form.

Sale form example, the link criteria specifies that the CustomerID in the Customer form must match the CustomerID from the Sales form.

In most cases, the user would close the Customer form and return to the main Sale form. But what if users commonly keep both forms open at the same time? Then they would expect the data between the two forms to be synchronized so that when a new row is displayed on the Sale form, the matching data would be displayed on the Customer form. You might have to write a couple of lines of code to enable this synchronization to work. Essentially, whenever the Sale is changed, your code grabs the new CustomerID, passes it to the Customer form, requeries the database, and redisplays the form with the matching data.

A third, related situation is shown in Figure 6.23. Perhaps while looking at animal data, the customer decides to adopt that animal. An adoption button on the Animal form could quickly bring up the Sale form. It would be convenient if the button then copied the AnimalID into the appropriate space on the Sale form.

Again, you need to write a couple of lines of code to insert the AnimalID into the appropriate control on the Sale form. In some situations, you might want the Sales form to refer back to the ID value on the Animal table.

Business applications commonly need to compute subtotals from subforms. As shown in Figure 6.23, some systems treat subforms as entirely separate forms, so you must first do the subtotal calculation on the subform, and then copy its value back to the main form.

Direct Manipulation of Graphical Objects

Can form usability be improved? In the last few years, the user interface to applications has been changing. The heavy use of graphics has led to an emphasis on **direct manipulation of objects**. Instead of typing in commands, the user can

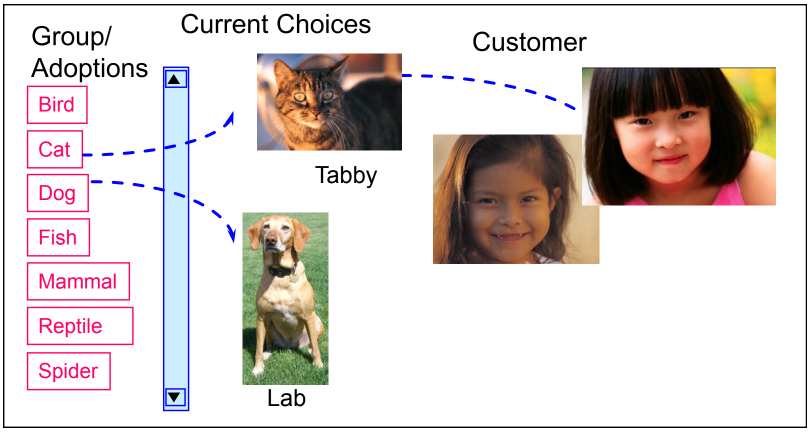


Figure 6.24

Direct manipulation of graphical objects at the Pet Store. Instead of entering an AnimalID into a box, you drag the picture of the animal to the customer to indicate a sale. Double-clicking on an item brings up more detail or related graphics screens.

drag an item from one location on the screen to another to indicate a change. Most people have seen this approach used with basic operating system commands. For example, in the days of command-line operating systems you had to type a command such as: COPY MYFILE.DOC A:MYFILE.DOC to copy a file. Today you click on the file icon and drag it to a disk drive icon.

More recently, direct manipulation has been expanded with the adoption of multi-touch screens. These devices provide options beyond the simple mouse-move commands. To date, most of the innovations have been in relatively simple actions such as the "pinch" move to contract or expand the size of an object. However, some opportunities exist to use this hardware for handling data. For instance, drilling down to see more detailed data or following referential links could be handled with a gesture. For ideas, see portions of the movie *Minority Report*. Ultimately, these gestures need to be standardized so they can be used on multiple devices and applications. Hopefully, standards groups will work on these concepts before some lawyer attempts to patent a gesture.

Sally's Pet Store Example

A graphical approach can make your applications easier to use. However, it requires changing the way you think about applications, and a good dose of creativity. Consider the Pet Store example. The basic forms designed earlier in this chapter were easy to create, and they will perform adequately. However, you could change the entire approach to the application.

Figure 6.24 shows a partial screen for the Pet Store example. Compare this form to the traditional data entry form shown in Figure 6.12. The traditional approach requires users to enter text into a box and perhaps select an item from a drop-down list. With the graphical approach the user sees photos of the individual animals or merchandise and drags them to the customer to indicate a sale. Double-clicking on an item provides more pictures or additional details. A similar approach would be used to special-order items, using **drag-and-drop** techniques to create search conditions such as category and price to narrow down the list of products.

Note that you cannot entirely eliminate data entry. At some point, you need to collect basic data on customers (name, address, phone number, etc.). This data could be entered on a traditional form that is activated when the clerk double-clicks the customer icon or photo. That is, the form in Figure 6.24 replaces the traditional sales form but does not replace the basic customer form. Of course, once the customer data is on file, it can be dragged back to this main form whenever the customer returns.

The Internet

The emphasis on graphics and a direct manipulation of objects can be particularly valuable for forms used with the Internet. For starters, most of the users will have little experience with databases and only limited knowledge of your company. Creating a graphical model of the company and its processes achieves two important objectives: (1) It makes the site easier to use because it matches the physical purchase methods users already know, and (2) it limits the actions of the users to those that you have defined.

Many Web forms use an approach similar to the direct objects, except that drag-and-drop is rarely implemented because of browser limitations. Most of the time, users browse or search a catalog of items. Selected items are added to a shopping cart. At checkout, the items are shown in a listing similar to a traditional sales form—but users generally must return to the catalog listing to edit the list to change or select new items. Perhaps as HTML5 gains popularity and developer experience, graphical ideas will expand into Web forms and applications.

One of the goals of the graphical approach is to hide the use of the database. Yes, all basic product information, figures, and sales data are stored in the database. The database system provides search capabilities and stores user selections. It also provides reports and data analysis for managers. However, users never need to know about the database itself. Users simply see an image of a store and its products. They manipulate the objects to learn more or to place orders. One difficulty of the Internet is that you are limited by the capabilities of the user's browser. Browsers handle most simple data entry controls, but rarely have the ability to perform drag-and-drop operations. Nonetheless, searching for graphical approaches can help you find ways to make the forms more intuitive and easier to use.

Complications and Limitations of a Graphical Approach

Several potential drawbacks exist to basing a form on the direct manipulation of graphical objects. The most important is that it can be an inefficient way to enter data. For example, you would not expect workers at a receiving dock to use a drag-and-drop form to record the receipt of several hundred boxes. A bar-code scanner would be considerably more efficient. Likewise, a quality control technician would prefer a simple keystroke (or voice) system so he or she could enter data without looking away from the task.

Even the Pet Store sales form is a debatable use of the drag-and-drop approach for in-store use. Think about the operations at a typical large pet store. Consider what would happen when dozens of customers bring shopping carts full of merchandise to the checkout counter. If a clerk has to use a drag-and-drop screen, the checkout process would take forever. Again, bar-code scanners would speed up the process. On the other hand, perhaps the operations of the store could be improved by eliminating the checkout clerk. Think about how the store would function if shoppers used the store's drag-and-drop Web site to select products, which were then delivered, or bagged and stacked for drive-through pickup. The difference in the value of the approach depends on the operations of the business and on who will be using the application.

A second difficulty with the graphics approach is that each application requires a considerable amount of custom programming. The traditional approach is reasonably straightforward. Common tools exist for entering data with forms made up of text boxes, combo boxes, and subforms. These tools can be used for virtually any database application. On the other hand, direct manipulation of objects requires that individual business objects be drawn on the screen and associated with data. Then each user action (double-click and drag-and-drop) has to be defined specifically for that application. In the future tools may be created to assist in this programming. However, today, a graphical approach requires considerably more programming effort than other approaches.

Building graphical database applications across the Internet carries similar problems. There are two primary limitations: transmission speed and limitations of software tools. However, a huge amount of money and effort is being directed toward the Web. Many firms in several industries are working on solutions to both limitations.

Database Design Revisited

What problems arise if a design always uses one-to-many relationships? It is helpful to see how database design decisions affect the layout of a form. The key element to remember is that one-to-many relationships are built on forms as either a subform or a linked form. The "many" side of the relationship is repeating, so the form needs some mechanism for collecting multiple rows of data. Remember when you design a database you often have to decide if a relationship is one-to-one or one-to-many. If you take the lazy way out and make everything one-to-one, users will yell because it will not be possible to record the important data. For example, in the standard sales form, think about what would happen if the design allowed only one item to be sold at a time, instead of many. You would no longer need the SaleItems table and its corresponding subform. Instead, the Sales form would contain a single spot to select one product being sold. In some fields (such as real estate), this approach is reasonable. But, what happens when someone wants to purchase five items at the same time? The clerk could turn away the customer (which is really bad). Or, the clerk could initiate five different Sales transactions—each with a single item. It would work, but the clerk would be unhappy, and the customer would decide you had no clue how to build an information system, or run a store. So, you created the SaleItems table and added the subform to the Sales form to handle multiple items (rows) per sale.

Now look at the other extreme. You might opt for flexibility in your design and build every relationship as one-to-many. Again, consider the sales example. Most organizations assume that any given sale is processed by one employee for one customer. To handle a wider variety of situations, you decide to model both of those as one-to-many relationships: (1) Any sale can be made to many customers, and (2) Any sale can involve multiple employees. The catch is that you now need to add two tables to the design (SaleCustomers and SaleEmployees). Figure 6.25

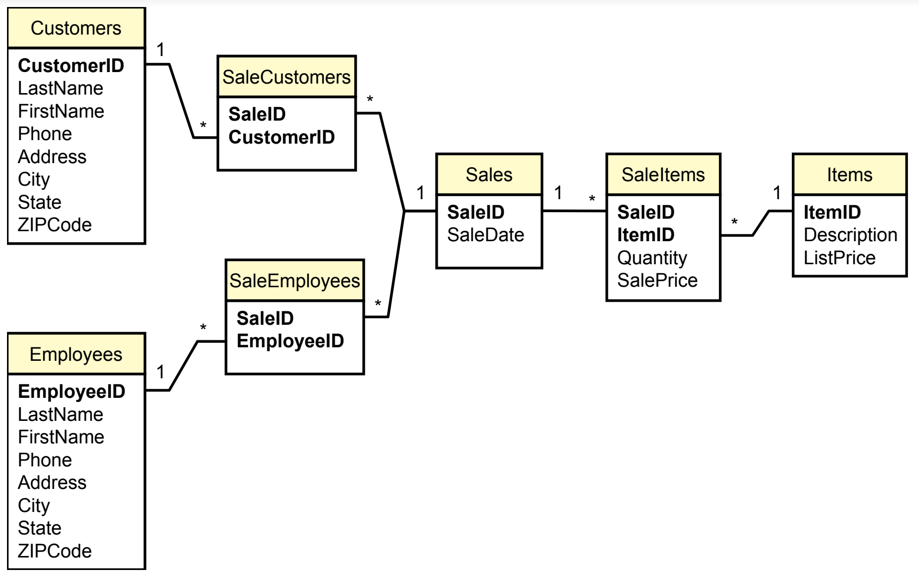


Figure 6.25

Database design variation. If there can be many customers and many employees per sale, the database design needs two new tables: SaleCustomers and SaleEmployees.

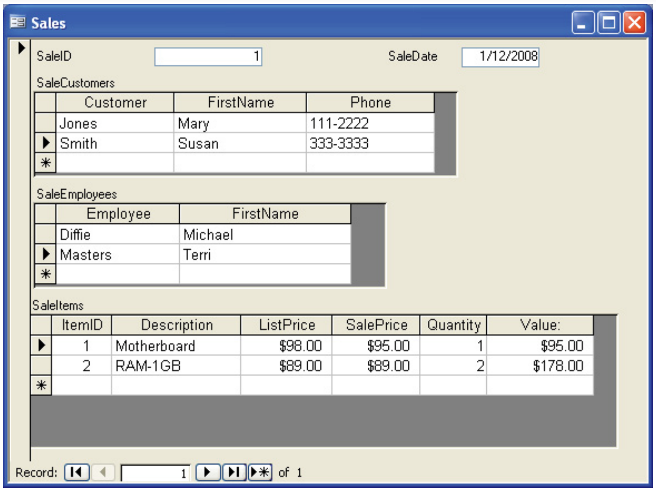
shows the two new tables. Observe that both SaleID and CustomerID (or EmployeeID) need to be part of the primary key.

More importantly, you need to add two repeating sections to the Sales form. Figure 6.26 shows a version of the new form. Notice the addition of the two scrolling regions for entering customers and employees. A traditional sales form would contain only a single entry for customer and for employee. If the organization often needs to record multiple customers and employees for each sale, there is nothing wrong with this version of the form. However, if the company almost always records only one entry for each, then this form is overkill. It takes up more screen space and is more difficult to understand. It will probably require more training to use—just to explain to clerks that most of the time, they will simply enter one customer and one employee.

The point is that you need to think about the usability of the forms when you design the database. You cannot just randomly choose one-to-one or one-to-many relationships. They must match the true needs of the organization.

Reports

What are the basic roles of reports? When you understand forms, reports are straightforward. Increasingly, the main difference between forms and reports is that forms are used to enter data and reports tend to have a set of fixed formats—with an emphasis on subtotals. Increasingly, reports are delivered electronically and sometimes have interactive elements, so the line between forms and reports is blurry. Some reports are still designed to be printed—such as receipts or invoices. However, report writers are increasingly Web based and managers use them to evaluate and summarize data using Web browsers. It is possible to create reports using traditional forms tools, but a report writer has two main strengths:



SaleID 1 SaleDate 1/12/2008

SaleCustomers

Customer	FirstName	Phone
Jones	Mary	111-2222
Smith	Susan	333-3333

SaleEmployees

Employee	FirstName
Diffie	Michael
Masters	Terri

SaleItems

ItemID	Description	ListPrice	SalePrice	Quantity	Value:
1	Motherboard	$98.00	$95.00	1	$95.00
2	RAM-1GB	$89.00	$89.00	2	$178.00

Figure 6.26

Database design variation. To handle many customers and many employees, two new subforms need to be added to the sales form. This form is more complex and more difficult to use than the standard sales form, so it should be avoided unless the users truly do need to record many customers and many employees.

(1) It can easily handle multiple pages of output (with consistent page headers and page numbering) and (2) It can combine both detailed and summary data. Chapter 5 illustrates how SQL queries can produce relatively complex results with the GROUP BY clause. However, a single SQL query can be used to display either detail rows of data or the summaries—not both. A good DBMS report writer also provides additional control over the output, such as printing negative values in red.

Report Design

As summarized in Figure 6.27, several issues are involved in designing reports. As in the development of forms, you and the users need to determine the content and layout. You must also identify the typical size of the report (number of pages and number of copies), along with noting how often it must be printed. Because of the physical steps involved, printing reports can be a time-consuming process. A report of a few dozen pages is no problem. However, when a report blooms into hundreds of pages with thousands of copies, you have to plan more carefully. First, you need a fast, heavy-duty printer. Then you need machines and people to assemble and distribute the report copies. You generally have to schedule time to use the printer for large reports.

Paper reports also present a different challenge to security. Paper reports require the use of more traditional security controls, such as written distribution lists, numbered copies, and control data. If security is an important issue in an organization, then these controls should be established when the report is designed.

Several physical and artistic aspects are involved in designing reports. The size of the page, the typeface used, and overall design of the page all must be determined. Newer DBMS report writers are relatively flexible, which is good and bad.

Report usage/user needs	Security controls
Report layout choices	Distribution list
Tabular	Unique numbering
Columns/subgroups	Concealed/nonprinted data
Charts/graphs	Secured printers
Paper sizes	Transmission limits
Printer constraints	Print queue controls
How often is it generated?	Output concerns
Events that trigger report	Typefaces
Size of the report	Readability
Number of copies	Size
Availability of color	User disabilities
	OCR needs

Figure 6.27

Fundamentals of report design. Determine content and layout with users. Estimate size and printing times. Identify security controls. Check typefaces and sizing for user readability.

The good part is that designers have greater control over the report. The bad part is that designers need to understand more about design—including the terminology.

Artistic design and a thorough treatment of design issues are beyond the scope of this book. If you are serious about design (for paper reports, forms, or Web pages), you should take a course in graphic design. In any case, it helps if you learn a few basic terms.

Terminology

Many of the basic terms come from typesetting and graphics design. The terms shown in Figure 6.28 will help you understand report writers and produce better reports. The first step is to choose the page layout, in terms of paper size; orientation (portrait versus landscape); and margins. The type of binding system will affect the margins, and you might have to leave an extra gutter margin to accommodate binding.

The next step is to choose the typeface and font size. In general, serif typefaces are easier to read, but sans serif faces have more white space, making them easier to read at larger and smaller sizes. Avoid ornamental typefaces except for covers and some headings. Columns of numbers are generally printed at a fixed width to keep columns aligned. Special fixed-width typefaces (e.g., Courier), in which all of the characters use exactly the same width, are especially appropriate if you need to align columns of nonnumeric data without the use of tab stops.

Font size is generally specified in terms of points. Most common printed material ranges from 10- to 12-point fonts. A useful rule of thumb is that a capitalized letter in a 72-point font is approximately 1 inch tall. Some report systems measure sizes and distance in picas. A pica is 1/6 of an inch, or the same height as a 12-point font (72/6 = 12).

If your reports include graphs and images, the terminology becomes more complex. Be aware that the quality of bitmap images depends on the resolution of the original image and the resolution of the output device. Common laser printers have a 1200-dots-per-inch (dpi) resolution. Typesetters typically achieve about

- Page layout
  - Landscape vs. portrait
  - Margins
  - Gutter
  - Trim
- Typefaces
  - Serif (Times New Roman)
  - Sans serif (Arial)
  - Ornamental (*Script*)
  - Fixed width (`Courier`)
- Font size
  - Common: 10-12 point
  - 72 points approximately 1 inch
  - Pica: 1/6 inch, 12 points

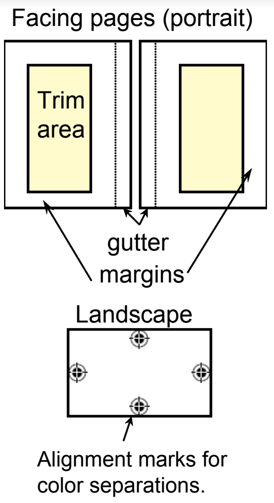


Figure 6.28

Basic publishing terminology. Understanding the basic design terms helps you design better reports and communicate with publishers and typesetters.

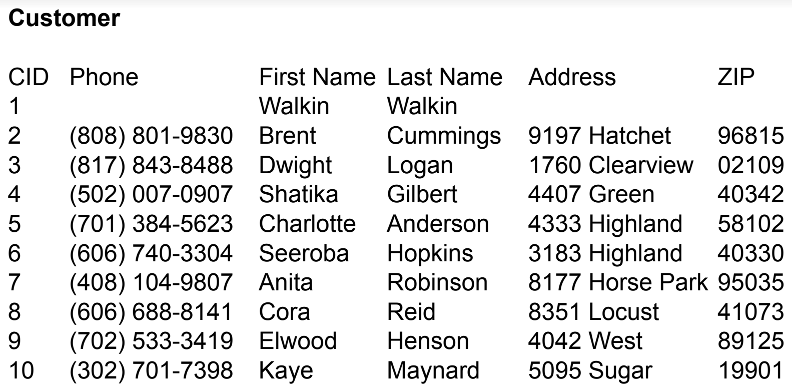
2,400-dpi resolution. An image that looks good on a 1200-dpi laser may be too small or too jagged on a 2,400-dpi typesetter.

If your reports are in color, you quickly encounter additional problems. In particular, colors on your screen may not be the same as on the printer. Similarly, a sample report printed on a color ink jet might look completely different when submitted to a typesetter. The Pantone® color standard is designed to minimize these problems by providing numbers for many standard colors. Advanced software also supports gamma correction that you can apply to your monitor so that colors displayed on your monitor will match those from the printer. The related issue you will encounter in color printing is the need to create color separations for all of your reports. For full-color submissions to print shops, each report page will need four separate color sheets. Denoted CMYK for each of the three primary colors—cyan (blue), magenta (red), and yellow—and the key color (black). In this case, each page will need high-resolution alignment marks so the colors can be reassembled properly.

One of the first elements of design that you must learn is to keep your reports simple and elegant. For instance, stick to one typeface and one or two font sizes on a page. Use plenty of white space to highlight columns and features. Look at the entire layout to observe where the eyes will be attracted and how they will move. Most important, since design style continually changes, examine newspaper and magazine layouts regularly for new ideas and patterns.

Basic Report Types

From the perspective of data layout, there are essentially three types of report designs: tabular, groups or subtotals, and labels. The choice you make depends on the type of data and use of the report. All report writers support these three basic formats. Many provide options to combine various elements.



Customer

CID	Phone	First Name	Last Name	Address	ZIP
1		Walkin	Walkin		
2	(808) 801-9830	Brent	Cummings	9197 Hatchet	96815
3	(817) 843-8488	Dwight	Logan	1760 Clearview	02109
4	(502) 007-0907	Shatika	Gilbert	4407 Green	40342
5	(701) 384-5623	Charlotte	Anderson	4333 Highland	58102
6	(606) 740-3304	Seeroba	Hopkins	3183 Highland	40330
7	(408) 104-9807	Anita	Robinson	8177 Horse Park	95035
8	(606) 688-8141	Cora	Reid	8351 Locust	41073
9	(702) 533-3419	Elwood	Henson	4042 West	89125
10	(302) 701-7398	Kaye	Maynard	5095 Sugar	19901

Figure 6.29

Tabular report layout. Tabular reports have few options but are good for detailed data listings. They are used for itemized listings of data.

Tabular and Label Reports

The tabular layout shown in Figure 6.29 is the simplest report design. It basically prints columns of data, much like the output of a query. The advantage over a simple query is that the tabular report can print page headings and page numbers on every page. You also have a little more control over font size and column width. Tabular reports are generally used for detail item listings, such as inventory reports. Note that the sort order becomes crucial, since these reports will be used to search for specific items. On the other hand, these reports do not contain much information for making decisions. In general, printed versions of these reports should no longer be needed. It is generally easier to use a form or report to lookup the specific items needed instead of carrying them around on paper.

As shown in Figure 6.30, labels are also straightforward. The essence of a label report is that all output for one row of data is printed in one “column” on the page. Then the next row is printed in the following column. The name label report comes from the use of preprinted or precut pages used for labels. These reports are sometimes named based on the number of physical columns. The example in Figure 6.30 has three labels across a page, so it is a three-up report. Before report writers, printing a label report was quite challenging, since the printer could only work from the top of the page. Hence, you had to write a program that printed the top line for three different rows of data, then return and print the second line, and so on. Today’s printers are more flexible, and report writers make the job easy.

Keep in mind that label reports can be useful for other tasks—whenever you want to group data for one row into separate locations on a page. For instance, by inserting blank rows and changing the label size, you might create a tic-tac-toe pattern of data. It could be an interesting effect for a cover page or advertising sheet, but avoid using such patterns for hundreds of pages of data.

Groups or Subtotals

The most common type of report is based on groups and computes subtotals. It also provides the most flexibility over the layout of items on the report. Common examples include printing a receipt or a bill. Many times the report will print several rows of data, like the order form shown in Figure 6.31. Each order for the

Dwight Parrish 9904 Plum Springs Road Worcester, MA 01613	Dwight Logan 1760 Clearview Street Boston, MA 0210	David Sims 6623 Glenview Drive Boston, MA 02216
Hershel Keen 8124 Industrial Drive Nashua, NH 03080	Reva Kidel 5594 Halltown Road Bangor, ME 04401	Dan Kennedy 3108 Troon Court Burlington, VT 05401
Sharon Sexton 2551 Elementary Drive Barre, VT 05641	Kelly Moore 6116 Clearview Street Middlebury, VT 05753	Cassy Tuck 7977 Fairways Drive Clifton, NJ 07015

Figure 6.30

Tabular report layout. Tabular reports have few options but are good for detailed data listings. They are used for itemized listings of data.

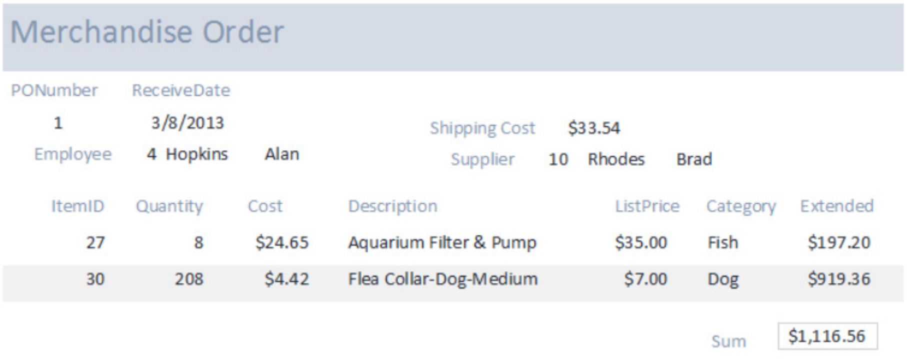
month is printed in one report, but the items are grouped together to show the individual order subtotals. Many people refer to these reports as control break reports.

Some Web-based reporting tools support a level of interactivity with group reports. For example, Microsoft SQL Server Reporting Services supports icons for each subtotal level that enable the user to roll-up the details and show just the subtotals or drill-down to see the details in one section only. The process is a simplified version of data cube browsers presented in Chapter 9.

The key to the subtotal report is to note that it includes both detail item listings (item ordered, quantity, cost, etc.), and group or total data (order date, customer, and order total). To create this report, you first build a query that contains the data that will be displayed. The example would probably include the Order, OrderItem, Merchandise, Customer, Employee, and Supplier tables. Be careful: If you want to see the detail, do not include a GROUP BY statement in the query. If you examine the raw data from this huge query, you will see a large number of rows and columns—many with repeating data. That is fine at this point, but not exactly

Figure 6.31

Group or subtotal report. Note that several orders are being printed. Each order is a group and has a detailed repeating section of items being ordered. The report can compute subtotals for each order and a total for the entire report.



Merchandise Order

PONumber ReceiveDate
1 3/8/2013
Employee 4 Hopkins Alan
Shipping Cost $33.54
Supplier 10 Rhodes Brad

ItemID	Quantity	Cost	Description	ListPrice	Category	Extended
27	8	$24.65	Aquarium Filter & Pump	$35.00	Fish	$197.20
30	208	$4.42	Flea Collar-Dog-Medium	$7.00	Dog	$919.36

Sum $1,116.56

Report Header
Page Header
Group Header1
    Group Header2
        …
        Detail
        …
    Group Footer2
Group Footer1
Page Footer
Report Footer

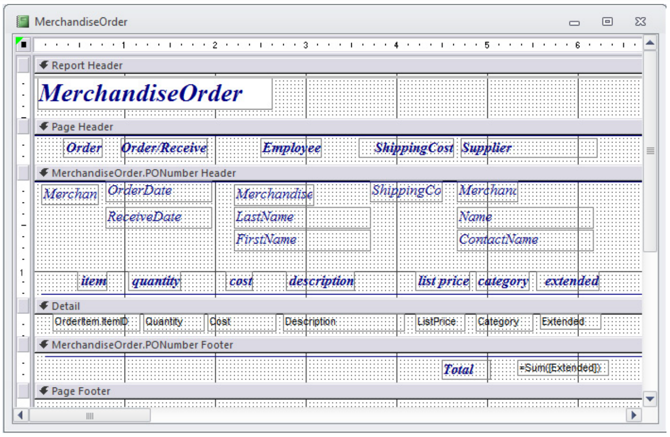


Figure 6.32

Report layout for groups. Note the basic report elements (report header, etc.). Also notice that the page layout is set by the position and properties of the data controls. In the sample report for Merchandise Orders, only one group is defined (on the Order number).

what the user wants to see. The objective of the report is to clean up the display of the data.

To create a grouped report, examine the report design shown in Figure 6.32. This layout page shows the **group breaks** in the data and specifies the layout of each element on the page. Again, layout is set by the individual controls. The controls have properties that can be changed to alter the appearance of the data displayed by that control. For example, you can set basic typeface and font attributes.

The basic elements of a report are headers, footers, group breaks and detail areas. The **report header** contains data that is displayed only when the report is first printed, such as a cover page. Similarly, the **report footer** is used to display data at the end of the report, for example, summary statistics or graphs. The **page header** and **page footer** are displayed on every page that is printed, except for the report header and footer pages. Page headers and footers can be used to display column headings, page numbers, corporate logos, or security identifiers.

The report features that define this type of report are the groups. The example shown in Figure 6.32 has one group defined: MerchandiseOrder.PONumber. The report will break, or create a new group of data, for each PONumber in the query. Notice that each Order can contain many items ordered. The report design specifies that these rows will be sorted by the ItemID number (within each order). Each grouping can have a group header and a group footer. The group header displays data that applies to the entire order (e.g., Date, Customer, Employee, and Supplier). It also holds the column labels for the detail (repeating) section. The group footer displays the subtotal for each group.

The common uses of each report element are summarized in Figure 6.33. Note that all of the elements (except detail) can work in pairs—headers and footers. You are not required to use both. For instance, you might choose to display page numbers in a page header and delete the page footer to provide additional space on the page.

Report Section	Usage
Report header	Title pages that are printed one time for entire report.
Page header	Title lines or page notes that are printed at the top of every page.
Group header	Data for a group (e.g. Order) and headings for the detail section.
Detail	Innermost data.
Group footer	Subtotals for the group.
Page footer	Printed at the bottom of every page—page totals or page numbers and notes.
Report footer	Printed one time at the end of the report. Summary notes, overall totals, and graphs for entire data set.

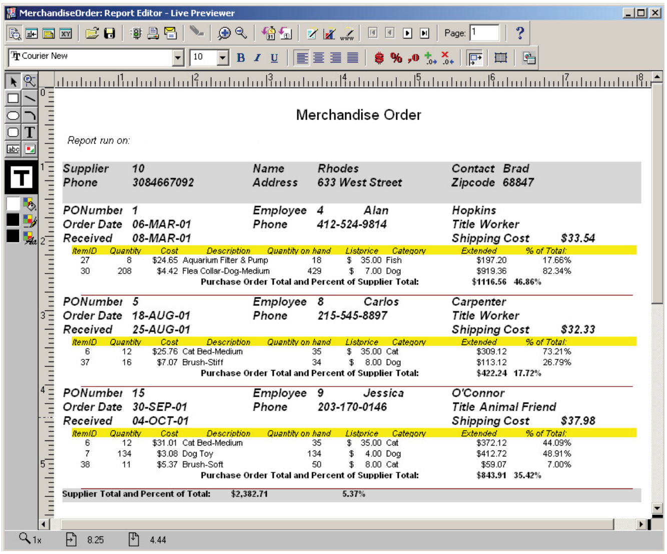
Figure 6.33

Common uses for report layout elements. Most elements are available in pairs, but you are free to delete any components you do not need.

For comparison, Figure 6.34 shows a similar report created using Oracle's 10g report writer. Notice that the overall structure is the same. Shading, boldface, and a highlight color have been used to improve the appearance of the report. However, it is still a little cramped, and could benefit from more white space. Ultimately, you would ask the users to focus the report on a specific item, and then improve the layout to highlight that item. In this version, the detail listing is highlighted,

Figure 6.34

Group report created and previewed with the report writer in Oracle. Shading is used to show the group headers and a highlight color draws attention to the detail heading.



```
                         Merchandise Order

Report run on:

Supplier     10              Name      Rhodes              Contact   Brad
Phone        3084667092      Address   633 West Street     Zipcode   68847

PONumber     1               Employee  4        Alan       Hopkins
Order Date   06-MAR-01       Phone     412-524-9814        Title Worker
Received     08-MAR-01                                     Shipping Cost      $33.54
 ItemID  Quantity   Cost   Description          Quantity on hand  Listprice  Category   Extended   % of Total:
   27       8     $24.65  Aquarium Filter & Pump        18        $  35.00  Fish        $197.20    17.66%
   30     208      $4.42  Flea Collar-Dog-Medium       429        $   7.00  Dog         $919.36    82.34%
                  Purchase Order Total and Percent of Supplier Total:                   $1116.56  46.86%

PONumber     5               Employee  8        Carlos     Carpenter
Order Date   18-AUG-01       Phone     215-545-8897        Title Worker
Received     25-AUG-01                                     Shipping Cost      $32.33
 ItemID  Quantity   Cost   Description          Quantity on hand  Listprice  Category   Extended   % of Total:
    6      12     $25.76  Cat Bed-Medium                35        $  35.00  Cat         $309.12    73.21%
   37      16      $7.07  Brush-Stiff                   34        $   8.00  Dog         $113.12    26.79%
                  Purchase Order Total and Percent of Supplier Total:                   $422.24   17.72%

PONumber     15              Employee  9        Jessica    O'Connor
Order Date   30-SEP-01       Phone     203-170-0146        Title Animal Friend
Received     04-OCT-01                                     Shipping Cost      $37.98
 ItemID  Quantity   Cost   Description          Quantity on hand  Listprice  Category   Extended   % of Total:
    6      12     $31.01  Cat Bed-Medium                35        $  35.00  Cat         $372.12    44.09%
    7     134      $3.08  Dog Toy                      134        $   4.00  Dog         $412.72    48.91%
   38      11      $5.37  Brush-Soft                    50        $   8.00  Cat          $59.07     7.00%
                  Purchase Order Total and Percent of Supplier Total:                   $843.91   35.42%

Supplier Total and Percent of Total:    $2,382.71          5.37%
```

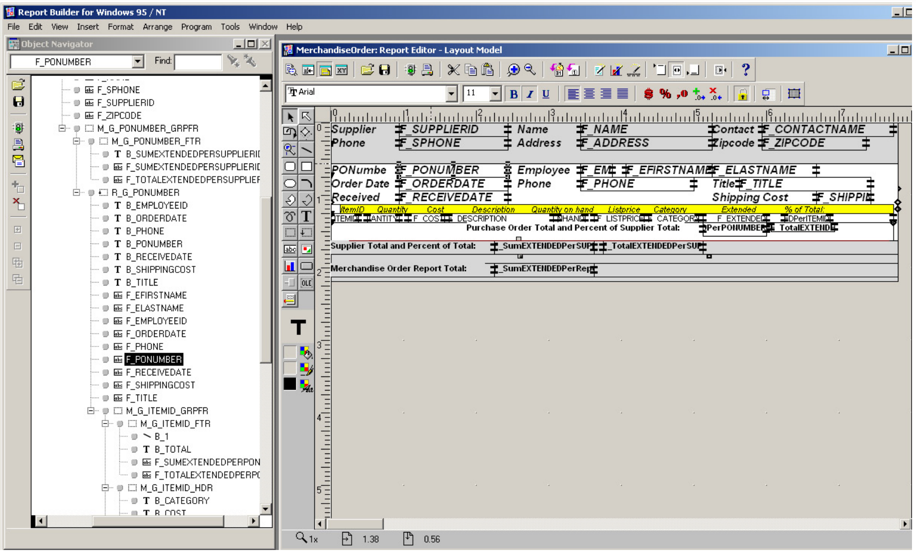

Report Builder for Windows 95 / NT
File Edit View Insert Format Arrange Program Tools Window Help
Object Navigator
F_PONUMBER
Find:
F_SPHONE
F_SUPPLIERID
F_ZIPCODE
M_G_PONUMBER_GRPFR
M_G_PONUMBER_FTR
B_SUMEXTENDEDPERSUPPLIERI
F_SUMEXTENDEDPERSUPPLIERI
F_TOTALEXTENDEDPERSUPPLIEF
R_G_PONUMBER
B_EMPLOYEEID
B_ORDERDATE
B_PHONE
B_PONUMBER
B_RECEIVEDATE
B_SHIPPINGCOST
B_TITLE
F_EFIRSTNAME
F_ELASTNAME
F_EMPLOYEEID
F_ORDERDATE
F_PHONE
F_PONUMBER
F_RECEIVEDATE
F_SHIPPINGCOST
F_TITLE
M_G_ITEMID_GRPFR
M_G_ITEMID_FTR
B_1
B_TOTAL
F_SUMEXTENDEDPERPON
F_TOTALEXTENDEDPERPC
M_G_ITEMID_HDR
B_CATEGORY
B_COST
MerchandiseOrder: Report Editor - Layout Model
Arial
11
Supplier F_SUPPLIERID Name F_NAME Contact F_CONTACTNAME
Phone F_SPHONE Address F_ADDRESS Zipcode F_ZIPCODE
PONumbe F_PONUMBER Employee F_EM F_EFIRSTNAME F_ELASTNAME
Order Date F_ORDERDATE Phone F_PHONE Title F_TITLE
Received F_RECEIVEDATE Shipping Cost F_SHIPPI
ItemID Quantity Cost Description Quantity on hand Listprice Category Extended % of Total:
Purchase Order Total and Percent of Supplier Total:
Supplier Total and Percent of Total: SumEXTENDEDPerSUP TotalEXTENDEDPerSUP
Merchandise Order Report Total: SumEXTENDEDPerRep
1x 1.38 0.56

Figure 6.35

Group report design in Oracle. Shading is used to show the group headers and a highlight color draws attention to the detail heading.

but if the total value of the order is more important to the users, you would need to reorganize this report.

Figure 6.35 shows the design view for the Oracle report. First, notice the overall structure is controlled by the sections, which are shaded in the design view. Second, observe that the individual data elements are displayed using text box controls. These controls are similar to the text boxes on a form, but only display data. You can set properties to change the format, font, or layout of the data. Finally, check out the hierarchical tree listing on the left side. It shows the complete structure of the report and makes it easy to find and select individual sections and controls.

Groups represent one-to-many relationships. For example, each order can have many items in the detail section. If there are several one-to-many (or many-to-many) relationships in the data, you might want to use multiple levels of groups. As illustrated in Figure 6.36, each group is nested inside another group, with the detail at the innermost level.

To create this report, you must build a query that contains every item that will be displayed. Begin by focusing on the detail level and then join additional tables until you have all the columns you need. You can use computed columns for minor computations such as Price * Quantity. Be careful to avoid aggregate functions (e.g., Sum) and avoid the use of GROUP BY statements. The only time you might include these two features is if your "detail" row is actually a subtotal (or average) itself.

Group reports are generally used for computations—particularly subtotals. In general, computations on one row of data should be performed with the query. On the other hand, aggregations (Sum, Average, etc.) are handled by the report writer. Report writers have different methods of defining the scope of the operation—that is, what data should be included in the total.

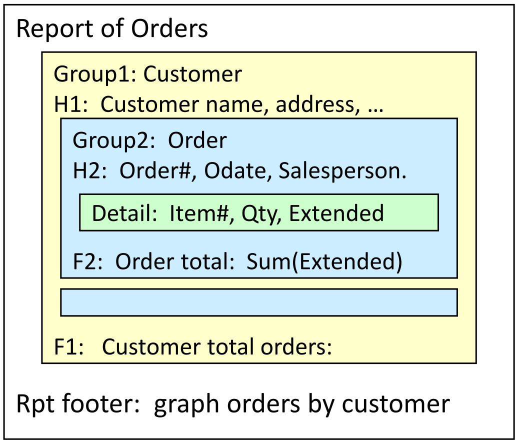

Report of Orders
Group1: Customer
H1: Customer name, address, …
Group2: Order
H2: Order#, Odate, Salesperson.
Detail: Item#, Qty, Extended
F2: Order total: Sum(Extended)
F1: Customer total orders:
Rpt footer: graph orders by customer

Figure 6.36

Nested groups. For example, each customer can place many orders, and each order has many detail lines. Two groups are used: (1) to show the total orders for each customer and (2) to show the total value of each order.

Charts

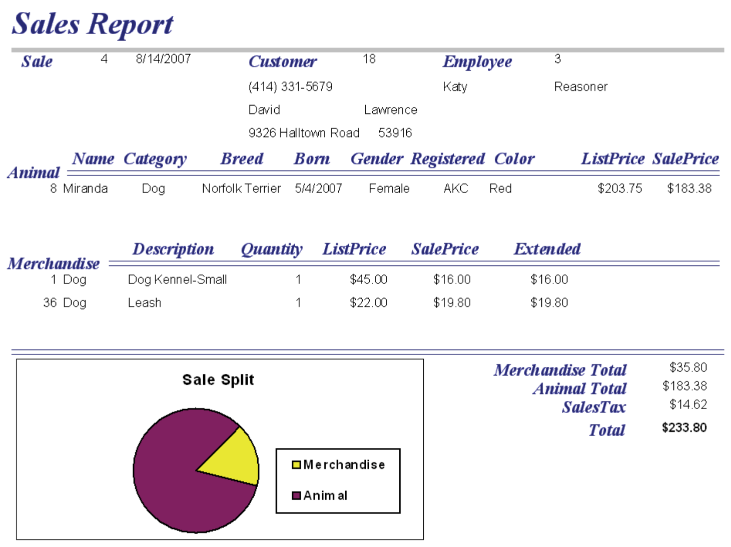
Reports are increasingly used for analysis and to identify patterns and trends in the data. As a result, users want charts to help them visualize the data. Charts on reports are similar to graphs on forms. The first step is to decide with the user what type of graph will best illustrate the data. The second step is to determine where the chart should be positioned within the report elements. If you are graphing detail items, then the graph belongs in the detail section, where it will be redrawn for every row of data. If it is a summary graph, it belongs in a group footer, or perhaps in the report footer if it summarizes data across the entire report.

Once you have determined the type and location of the graph, you build a query to collect the data. This query can be different from the query used to produce the overall report. In particular, when the graph is in a group footer, you might need to use aggregation functions in the query for the graph. Be sure to include a column that links the graph to the data in the report—even if that column will not be displayed on the graph. Figure 6.37 shows one sale on the Sales report for the Pet Store. The totals for the graph are computed by a separate query.

Summary

Forms must be designed to match the user's tasks and make your application easy to use. To meet this goal, you need to pay attention to design principles, operating system guidelines, and human limitations. Where possible, you should build the form to use direct manipulation of objects, such as dragging items from one location to another to signify shipment.

Forms are based on tables or queries. Each form has a single purpose and can store data in one table. More complex forms can be created by placing subforms onto the main form. Controls on the form are used to enter data into the tables, perform lookup functions, and manipulate data. Several standard controls are available for a Windows environment (e.g., text boxes, combo boxes, and option



Sales Report

Sale	4	8/14/2007	Customer	18	Employee	3
			(414) 331-5679		Katy	Reasoner
			David	Lawrence		
			9326 Halltown Road	53916		

Animal

	Name	Category	Breed	Born	Gender	Registered	Color	ListPrice	SalePrice
8	Miranda	Dog	Norfolk Terrier	5/4/2007	Female	AKC	Red	$203.75	$183.38

Merchandise

		Description	Quantity	ListPrice	SalePrice	Extended
1	Dog	Dog Kennel-Small	1	$45.00	$16.00	$16.00
36	Dog	Leash	1	$22.00	$19.80	$19.80

Merchandise Total	$35.80
Animal Total	$183.38
SalesTax	$14.62
Total	$233.80

Figure 6.37

Sample graph on the sales report. It illustrates the portion of the sale spent on animals versus merchandise. Note that the graph appears on the same level as the Sale table—not on the detail level and not on the report footer.

buttons). Additional controls can be purchased to handle more complex tasks, such as calendars for scheduling and three-dimensional imaging.

Reports are generally printed and differ from forms because reports are designed only to present data, not to collect it. There are several types of forms, but many business forms rely on subtotals or groupings to display different levels of data. For example, a sales report might be grouped by sales division or salesperson or both. You use a query to combine all data items needed for a report. There are two benefits to using a report writer: (1) It is a straightforward way to set data formats and alignment, and (2) The report can include detail listings as well as subtotals and totals.

A Developer's View

As Miranda noted, the database wizards can create basic forms for you. However, before you crank up the form wizard and generate hundreds of small forms, think about the tasks of the users and the overall design. Try to put the most important information on one central form with a few secondary forms to help. Strive for a clean, well-organized screen and use colors and graphics sparingly to enhance the appearance. You should also develop a design standard and layout for the application to ensure consistency. Just be sure to leave room for creativity. For your class project, you should begin creating the basic forms and reports.

Key Terms

accessibility
aesthetics
check box
clarity
command button
consistency
controls
data-bound controls
direct manipulation of objects
drag-and-drop
event
feedback
focus
group break
heads-down data entry
human factors design
input mask
label
option button
page footer
page header
report footer
report header
resource file
single-row form
startup form
subform form
tab order
tabular form
text box
Unicode
use case

Review Questions

1. Which human factors are important to consider when designing forms?
2. How can you make your applications accessible to a wider group of workers?
3. How are international issues handled on forms?
4. What are the primary form types?
5. What are the main controls you can use on forms?
6. Why are updateable queries an issue with forms?
7. What is the purpose of subforms?
8. What is the purpose of linked forms?
9. What are the main report types?
10. What are the primary sections of reports?

Exercises

1. ✓ Research and report on the steps needed to create a multi-language version of a form using Microsoft .NET (Web version). Or use Java if your instructor prefers.
2. Review the documentation for the DBMS you are using and identify the features that it provides to support accessibility for all users.
3. Review the documentation for the DBMS you are using and identify the main forms controls that it provides.
4. Review the documentation for the DBMS you are using and describe how reports can be accessed via the Web.

For the following questions, create the databases and build the forms for the exercises from chapters 2 and 3. Note: Make initial forms. They do not have to be perfect matches to the drawings.

5. Medical test results, Chapter 2, Exercise 1.
6. ✓ Household appliance sales, Chapter 2, Exercise 2.
7. Repair service, Chapter 2, Exercise 3.
8. Day spa, Chapter 2, Exercise 4.
9. Glove manufacturer, Chapter 2, Exercise 5.
10. Custom ear bud manufacturer, Chapter 2, Exercise 6.
11. Automobile maintenance, Chapter 2, Exercise 7.
12. Farmer's market sales, Chapter 3, Exercise 1.
13. Exercise records, Chapter 3, Exercise 2.
14. Basketball league, Chapter 3, Exercise 3.
15. Network tracking, Chapter 3, Exercise 4.
16. Custom cell phone cases, Chapter 3, Exercise 5.
17. Vintage clothing sales, Chapter 3, Exercise 6.
18. Voter contacts, Chapter 3, Exercise 7.

Sally's Pet Store

19. Create basic forms to handle administrative tasks such as editing Breed, Category, AdoptionGroup, and Supplier data.
20. Create a form to edit Merchandise information. Eventually, this table will contain thousands of entries so provide features to enable users to sort and filter the data by category and description.
21. Create a form that makes it easy for adoption groups to enter new animal information. The group should only be able to edit data for their own animals that have not yet been adopted. (Assume they will eventually log in, but initially just use a drop-down list to have them set the Adoption Group.)
22. Create a form with a drop-down list to select a year/month and then display the total merchandise sales for that month by category. Include a chart.
23. Create a form that enables managers to select a starting and ending date and then display a chart showing total merchandise purchases over that time period for each supplier.
24. Create a report showing merchandise sales by category over time (monthly). Include a line chart.
25. Create a report showing total sales of merchandise by month for each supplier compared to monthly purchases from those manufacturers. Include a chart.
26. Create a report with a chart showing a count of the number of sales by employee by month.
27. Create a report showing total merchandise purchases by category for customers who have adopted at least one cat and one dog. (Hint: First build a query to get the list of customers who have adopted a cat and a dog.)

Rolling Thunder Bicycles

28. Create a form with a drop-down list for year that then displays a list of the top 10 customers for the year in terms of total sales. Hint: Use parameters in the query to refer to the control values.
29. Create a form that enables a manager to choose an employee, starting date, and ending date and display a chart of sales by model type for those conditions. Hint: Use parameters in the query to refer to the control values.
30. Create a form that enables employees to select a component and see the current quantity in stock along with a chart of sales (installations) by year/month.
31. Create a form that enables a manager to enter a start date and ending date and display a chart of sales by model type by month for that date range.
32. Create a sales report that shows total sales by employee by month.
33. Create a report that enables a manager to specify a start and end date and show total sales organized by model type between those dates. Hint: Use parameters in the query.

34. Create a sales report that shows total sales by model type for each month, organized by year.

35. Create a report to show total purchases and total payments to manufacturers by year.

Corner Med

Corner Med

36. Create a form that lets a physician select a diagnosis code and see all patients (sorted from the most recent) with that diagnosis..

✓ 37. Create a report that prints a bill for a patient visit.

38. Create a form that enables a physician to enter keywords and then search for ICD10 diagnosis codes that match those keywords. Note: You probably have to stick with relatively simple search methods.

39. Create a report that displays a chart for a specified patient. It should include all visits in chronological order and lists symptoms and treatments at each visit.

40. Create a report that displays the Bills (visits) that have been overpaid. Hint: First create a query to compute the total amount owed by visit.

41. Create a report that lists the count of the number of patients by month under the primary ICD diagnosis categories. The major groupings are based on the following table.

ICD10 Code	Category
A00-B99	1. Infectious and Parasitic Diseases
C00-D48	2. Neoplasms
D50-D89	3. Diseases of the blood and blood-forming organs…
E00-E90	4. Endocrine, nutritional and metabolic diseases
F00-F99	5. Mental and behavioral disorders
G00-G99	6. Diseases of the nervous system
H00-H59	7. Diseases of the eye and adnexa
H60-H95	8. Diseases of the ear and mastoid process
I00-I99	9. Diseases of the circulatory system
J00-J99	10. Diseases of the respiratory system
K00-K93	11. Diseases of the digestive system
L00-L99	12. Diseases of the skin and subcutaneous tissue
M00-M99	13. Diseases of the musculoskeletal system and connective tissue
N00-N99	14. Diseases of the genitourinary system
O00-O99	15. Pregnancy, childbirth and puerperium
P00-P96	16. Certain conditions originating in the perinatal period
Q00-Q99	17. Congenital malformations, … and chromosomal abnormalities
R00-R99	18. Symptoms, signs and abnormal clinical and lab findings, not elsewhere
S00-T98	19. Injury, poisoning and certain other consequences of external causes
V01-Y98	20. External causes of morbidity and mortality
Z00-Z99	21. Factors influencing health status and contact with health services
U00-U99	22. Codes for special purposes

Web Site References

http://www.microsoft.com/enable	Accessibility guidelines.
http://www.unicode.org	Primary site for Unicode information.
http://www.sigchi.org/	Association for Computing Machinery—Special Interest Group: Computer and Human Interaction.
http://www.sigaccess.org/	Association for Computing Machinery—Special Interest Group: Computers and the Physically Handicapped.

Additional Reading

Cooper, A. *About Face: The Essentials of User Interface Design.* Foster City, CA: IDG Books, 1997. [A good discussion of various design issues.]

Ivory, M. and M. Hearst, The State of the Art in Automating Usability, *Communications of the ACM*, 33(4), December 2001, 470-516. [General discussion on evaluating system usability.]

Koletzke, P. *Oracle Developer Advanced Forms and Reports*, Berkeley: Osborne/McGraw-Hill, 2000.

O'Reilly, Inc, Ed, *The Oracle PL/SQL CD Bookshelf: 7 Best Selling Books on CD ROM*, Cambridge, MA: O'Reilly & Associates, 2000. [A collection of several useful Oracle reference books on CD-ROM.]

Raskin, J. *Humane Interface*, The: New Directions for Designing Interactive Systems, Reading, MA: Addison-Wesley, 2000. [The need for a new interface as explained by the creator of the Apple Macintosh project.]

Tsichritzis, D. Form Management, *Communications of the ACM*, 25(7), July 1982. [Basic concepts of database forms.]

Chapter 7

Database Integrity and Transactions

Chapter Outline

What You Will Learn in This Chapter

- Why would you need to use procedural code when SQL is so powerful?
- How are SQL commands integrated into more traditional programming structures?
- What capabilities exist in procedural code?
- How are business rules added to the database?
- How does a DBMS handle multiple transaction events?
- How do you prevent problems arising when two processes change the same data?
- What are the primary rules to ensure integrity of transactions?
- How are key values generated?
- How can procedural code track row-by-row through a query?
- What issues arise when maintaining totals in the database?

A Developer's View

Ariel: Well, is the application finished?

Miranda: No. The basic forms and reports are done. But I'm still running into some problems.

Ariel: I guess there is always more to do. What kinds of problems?

Miranda: Well, the numbers are sometimes wrong. It seems to happen when several people are working on the same data at the same time. And the application seems a little slow sometimes. And…

Ariel: Whoa. I get the picture. But these seem like common problems. Does the database system have any tools to help?

Miranda: I think so. I'm going to start by looking at some programming topics and data triggers. Then, I think indexes will help me with performance.

Getting Started

Procedural code (programming) is used to handle transactions and other operations that must be performed in a specific order. Currently, every DBMS has its own proprietary programming language. Although the features are similar, the syntax varies. So you need to learn how to write some fundamental programs in the DBMS you want to use. Procedural code is needed for tasks such as custom functions, transactions that require multiple changes, handling concurrency issues, and generating key values.

Introduction

Why would you need to use procedural code when SQL is so powerful? Business applications often exhibit several common problems. For example, multiple users might try to change the same data at the same time, or multiple changes need to be made together, or you need to generate new ID numbers for a table. These situations must be handled correctly to ensure the integrity of the data. SQL commands are powerful tools, but in many of these situations, you need the ability to execute multiple statements or to choose which command should be run. Database systems have evolved procedural languages to handle these situations.

Although there are diverse methods to implement procedural languages, it is helpful when the language is embedded into the query system. With this approach, all of the code and conditions remain within the database definition and constraints are enforced automatically for all applications. These conditions are often written as data triggers—code that is executed when some data element is modified.

The issues of transactions, concurrent access, and key generation appear in almost every business application. This chapter explains the issues involved and provides the common solutions. Performance is a tricky issue as databases expand into huge datasets. Complex queries across many large tables could take a long time to run. But, transaction-based applications need to process data quickly. Vendors have invested considerable money and time into improving performance.

One common solution is to create indexes on the tables. You need to understand the basic index technologies to make informed choices to improve your application's performance.

Two-Minute Chapter

SQL is powerful but sometimes it is necessary to use traditional procedural programming languages to accomplish tasks. Procedural code executes one operation at a time and includes loops and conditional statements. It is often used to examine one row of data at a time. The large DBMSs integrate procedural statements with SQL commands. Many also support writing code in external languages (such as C and Java) that can submit SQL statements to store and retrieve data.

Writing procedural code requires several steps. (1) Learning the overall functions and syntax of the commands. (2) Understanding where to place the code so that it is executed at the proper time. (3) Testing and debugging. (4) Learning when to use procedural code.

The challenge with Step 1 is that the SQL standard has begun defining procedural elements but most systems still rely on their proprietary commands. The overall structures are similar but the details are different, which are explained in the workbooks. Primary structures include the ability to define Functions, Conditions, and Loops. SQL commands are integrated with programming code by defining parameters (or variables) within the SQL command that hold values assigned from the code.

Step 2 is critical because most systems today are event-driven and code is executed in response to some defined event. Within a database, you typically attach code to common data triggers including data UPDATE, INSERT, and DELETE events. For example, when a row of data in a table is changed, the DBMS can execute your custom code to check various conditions. So you have to first think in terms of when your code should be executed.

Step 3 is important with any development method. DBMSs rarely provide additional support for testing, so it is critical for programmers to break things into small pieces and thoroughly test all of the pieces during development.

In terms of Step 4, several common business situations require the support of procedural code. Support for transactions is the most important: Several operations that must be performed (or failed) together. The classic example is transferring money from one bank account to another. Handling errors, including issues with concurrent access, is another common situation. Some systems (particularly Oracle) also require support for generating key values. Other situations arise when dealing with creating forms and making them more usable.

Procedural Languages

How are SQL commands integrated into more traditional programming structures? A **procedural language** is a traditional programming language such as C or Java, where you specify the sequence of a set of commands. Common SQL commands are not procedural because you tell the DBMS only what you want done, not how to do it. Although SQL commands are powerful, sometimes you need the more precise control of a procedural language. For example, you might want to specify that a group of commands must be executed in a particular order and all must be completed for the transaction to succeed. Or, you might want to execute some commands only if certain external conditions are true. In

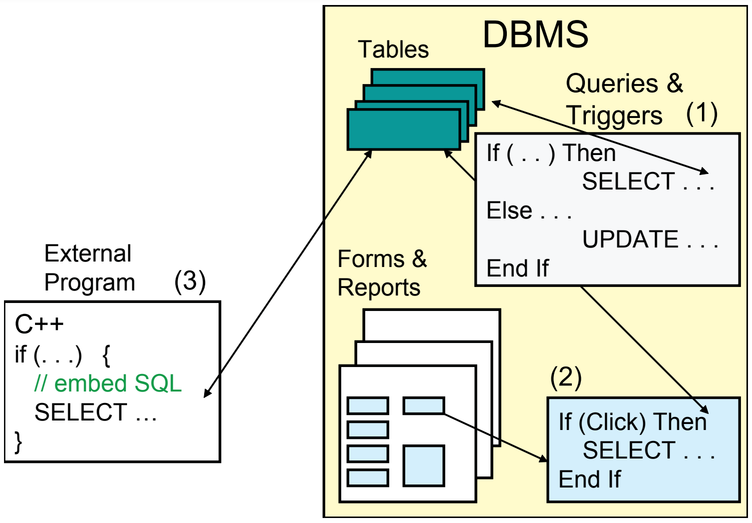


Figure 7.1

Location of procedural code. Code can usually be written in the query system, within a database form, or in an external program. When possible, code should be placed within the query system so that it cannot be by passed.

more complex cases, you might need to step through each row in a table to perform some difficult computation.

Many varieties of procedural languages exist, but they have elements in common. All of them have variables, conditional statements (if), loops, and subroutines. Each language has its own **syntax**, which includes details such as command and function names, statement terminators, assignment operators, and whether you use parentheses or square brackets for arrays. The syntax is important when you write code, but integrated editors help by prompting for various items and compilers will pinpoint most syntax errors.

This chapter focuses on the logic needed to handle common database operations. The main text is generally language neutral, so you can see how the ideas apply to any database situation. The workbooks provide specific examples using the syntax and structure of individual database systems.

Where Should Code Be Located?

One of the first major questions you face is where the code should be written, stored, and executed. Figure 7.1 shows that procedural code can be placed in three locations: (1) within the DBMS engine as queries or database triggers, (2) within forms and reports, or (3) in external programs. Large, commercial systems, such as Oracle, SQL Server, and DB2 have a procedural language embedded in the DBMS itself. You write code just as you would write any other query and can mix procedural commands with SQL statements. The SQL standard has slowly been adding procedural capabilities. But each vendor supports the concepts using a different syntax.

In general, code that relates directly to the data should be created as a **database trigger** inside the DBMS. Placing the code inside the DBMS means it is written only once and can be called automatically, regardless of how the data is accessed.

The DBMS will ensure that the code is always executed and not bypassed. Think about a security situation where you want to write a note to a log table every time someone changes an employee salary. If you rely on programmers to implement this code in their forms, they might forget to do it or even do it incorrectly. Additionally, someone could create an entirely new form or use a query to change the data directly, without executing the security code. Placing the code within the database provides a mechanism to ensure that it is run anytime the data is changed, regardless of how the modification is generated. In the SQL standard, procedural code stored within the database is called a **persistent stored module (PSM)**, and related procedures and functions can be stored in developer-defined modules. With the release of Office 2010, Microsoft added some rudimentary data macros that can be assigned to tables to handle these types of tasks.

Code within forms should concentrate on handling events or custom problems within the specific form. On the other hand, placing the code into a separate external file is a technique often used in n-tier client/server systems described in Chapter 11. It has the advantage of consolidating the business logic into one location. Separating the business logic from the DBMS makes it easier to replace the DBMS if desired. Database code in external software also arises on Web sites and other situations where data is exchanged with external devices, such as bar code scanners or other sensors.

User-Defined Functions

User-defined functions are a good illustration of procedural code. Occasionally you need a calculation that will be used by several different queries, reports or forms. Even if the computation is relatively simple, placing the code in one location makes it substantially easier to find and change later. You can define your own function name and perform almost any computation you need using procedural code. Figure 7.2 provides an example of a simple function to estimate item costs. In practice, this function would be more complex and include tables and queries, but keeping it simple focuses on the basic elements of a user-defined function.

A function is just a set of code designed to perform a defined task. Typically this function and task need to be called from multiple locations. Functions are passed values and perform computations on these parameters. A value is returned to the calling routine. You can also create procedures, which are different from

Figure 7.2

User-defined function. Placing the business logic in a central location makes it easy to modify later. The function can be used in code segments or SELECT statements.

```
CREATE FUNCTION EstimateCosts
        (ListPrice Currency, ItemCategory VarChar)
RETURNS Currency
BEGIN
        IF (ItemCategory = ‘Clothing’) THEN
                RETURN ListPrice * 0.5
        ELSE
                RETURN ListPrice * 0.75
        END IF
END
```

```
CREATE FUNCTION IncreaseSalary
        (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
BEGIN
        IF (Amt > 50000) THEN
                RETURN -1                   -- error flag
        END
        UPDATE Employee SET Salary = Salary + Amt
        WHERE EmployeeID = EmpID;
        RETURN Amt;
END
```

Figure 7.3

Function to update the database. The input parameters are used to specify values in the SQL statement. Additional computations can be performed and the parameters modified if needed.

functions in that they do not return a value. However, in almost all cases, you will want to use functions—if only to return error codes. A key feature is that you can include procedural statements such as "if" conditions to handle complex logic.

Figure 7.3 shows a function that uses input parameters to update the database. Almost all functions and procedures use parameters to pass in values to be used in calculations. You can also create local variables to modify the parameters and then use them in the SQL statement. Functions can be as complex as you need. The procedural language system contains the standard elements of any programming language: variables, conditions, loops, and subroutines.

The specific syntax of the module language and parameters depends on the DBMS. The versions shown here reflect the most recent SQL standard, which is only partially supported by DBMS vendors. Although Microsoft Access does not support the CREATE FUNCTION statement, you can build functions in VBA code modules.

Looking Up Data

Procedures and functions often need to be able to use data from tables or queries. Obtaining data from a single row is straightforward with the SELECT INTO statement. It behaves the same as a standard SELECT statement, but instead of displaying the values, it places them into local variables. However, you have to be careful to ensure that the SELECT statement returns only a single row of data. If you make a mistake in the WHERE condition and return multiple rows, it will generate an error.

Figure 7.4 shows how the SELECT INTO statement is used to retrieve a single value. The statement can be used to retrieve data from multiple columns. Just add another COLUMN INTO VARIABLE on the SELECT line and separate it with a comma from the existing line. Notice the difference between the overall objectives in Figures 7.3 and 7.4: The first hard-codes a maximum value (50000), whereas the new approach looks up the maximum raise in a table. This approach is better than using a fixed value because you can create a form that enables an administrator to change this value quickly. If you leave fixed numbers in your program code, a programmer would have to wade through all of the modules to find the magic number. In addition, anytime someone has to change program code,

```
CREATE FUNCTION IncreaseSalary
        (EmpID INTEGER, Amt CURRENCY)
RETURNS CURRENCY
DECLARE
        CURRENCY MaxAmount;
BEGIN
        SELECT MaxRaise INTO MaxAmount
        FROM CompanyLimits
        WHERE LimitName = 'Raise';

        IF (Amt > MaxAmount) THEN
                RETURN -1                   -- error flag
        END
        UPDATE Employee SET Salary = Salary + Amt
        WHERE EmployeeID = EmpID;
        RETURN Amt;
END
```

Figure 7.4

Looking up single data elements. The SELECT INTO statement can be used to return data from exactly one row in a table or query. The result is stored in a local variable (MaxAmount) that you can use in subsequent code or SQL statements.

there is a large risk that additional errors will be introduced. Whenever possible, you should place important values into a table and use the lookup process to get the current value when it is needed.

Programming Tools

What capabilities exist in procedural code? Ideally, you already know how to write program code in a separate language such as Basic, C#, Java, or C++. In most situations, you can use these tools to write any level of code you need and then embed database calls within that program. Typically, the database calls consist of SQL statements to insert or retrieve data. However, sometimes you will have to use the database language built into the DBMS. For instance, when you need to examine large amounts of data, it is usually faster to handle the data solely within the DBMS and return simpler results to other programs. Transferring data—even within the same computer—takes time and processing resources. The DBMS is already optimized for handling data internally.

The main concepts you need to know with any procedural language are: (1) Sequence, (2) Variables, (3) Conditions, (4) Loops, (5) Input and Output, and (6) Procedures and functions (subroutines). These are the building blocks or tools that are available to build programs.

One. The primary difference between SQL queries and programming languages is the concept of sequence. A procedural language executes one command line at a time and then moves to the next one. This process controls the order in which commands or steps are executed. In contrast, note that the SQL SELECT command provides minimal control over sequence. Rows are operated on in any order determined by the query optimizer. You can specify the sorting of the final result, but not the order in which rows are operated on. Hence, it is relatively easy to see situations where a procedural language is necessary, such as when two or more commands need to be executed in a specific sequence. A simple program might consist of two INSERT commands—where data is added to one table and then referenced by the second INSERT command.

Two. Variables are temporary locations in memory to hold data. They usually have a defined data type. Within a DBMS procedural language, the data types available match those used within tables, such as integer, float, and date. When code is written in more traditional languages such as Basic, C#, and Java, the database connector needs to transfer DBMS data types into local variable data types. This process is complicated when the database can hold Null values. External program code often needs special functions to translate data—watching for problems with Null values. One key to understanding variables is to recognize their scope. **Scope** refers to the context or location where a variable is defined. For instance, variables declared within a function only exist within that function. The values are hidden from code written in other functions.

Three. Conditions. The most common form is IF (condition) THEN … ELSE … END IF. Sometimes a CASE or ELSE IF block is available to test multiple values in one setting. The purpose is to define multiple code sections so that only one is executed depending on the value of the condition being tested. The action statements within the conditional element are indented to make them easier to read by separating them from the conditional logic.

Four. Loops. Loops define a block of code that is to be executed multiple times. The number of times can be fixed; determined by the amount of data such as the number of rows in a table; or determined dynamically within the loop. In a database environment, the most common use of loops is to define a SELECT query on a table to retrieve a set of rows—then execute the code for each row of data. This approach is used only when SQL cannot handle the problem. SQL is almost always faster at working with sets of rows, but sometimes, procedural code is needed when computations must be performed in a specific order.

Five. Input and Output. Code within the database typically needs to retrieve or store data in tables. SQL statements are used to handle these operations (SELECT, INSERT, UPDATE, and DELETE). The commands can be modified by adding parameters created from variables defined in the code. Code that is written on forms (or reports) can also access data entered onto the form by users.

Six. Procedures and Functions. These subroutines are used to split the code into manageable pieces that are easier to read and to debug. Procedures and functions contain code that can be called from multiple locations—so any code that needs to be used in more than one location should always be written as a function or procedure. But, even if the code is called only one time, it can be useful to write it as a separate function. Smaller functions are easier to debug and they reduce the complexity of the overall program. For example, perhaps you need to write a procedure that performs five different steps. Each step takes 10 lines of code to create. Instead of writing one procedure consisting of 50 lines of code, it is better to write a main procedure that calls five other procedures—each with the 10 lines of code.

Data Triggers

How are business rules added to the database? Data triggers are procedures that are executed when some event arises within the database. The code is written in the query system and is saved as a procedure or function within the database. By binding the code to the database tables, the DBMS ensures the code is always executed when changes are made to the data. The common events that can host

	INSERT	
BEFORE	DELETE	AFTER
	UPDATE	

Figure 7.5

Data triggers. You can set procedures to execute whenever one of these actions occurs. Row events can be triggered before or after the specified event occurs.

triggers are Update, Insert, and Delete, but some systems enable you to attach code to events related to users or the database instance. To understand the role of triggers, consider a procedure that is run whenever someone changes the Salary column in the Employee table. When the data is changed, your trigger procedure is fired to record the person who made the change. With the log, auditors can go back and see who made changes to this critical data. The salary example is a common use of data triggers, which is to add specific security or auditing features to the database. They can also be used to handle business events, such as monitoring when quantity on hand drops below some level and generating an e-mail message or an EDI order to a supplier.

Figure 7.5 lists the basic SQL commands that support triggers. The main data triggers on the rows and columns each have two attributes: BEFORE and AFTER. For example, you can specify a procedure for BEFORE UPDATE and a different procedure for AFTER UPDATE. The BEFORE UPDATE event is triggered when a user attempts to change data, but before the data is actually written to the database. The AFTER UPDATE trigger is fired once the data has been written. You choose the event based on what you want to do with your application. If you need to check data before it is written to the database, you need to use a BEFORE trigger. For instance, you might want to perform a complicated validation test before saving data. On the other hand, if you want to record when data was changed or need to alter a second piece of data, you can use an AFTER trigger.

Statement versus Row Triggers

The SQL standard defines two levels of triggers: (1) triggers may be assigned to the overall table or (2) they may be assigned to fire for each row of data being modified. Figure 7.6 shows the timing of the various triggers for an UPDATE command. Triggers created to the overall table are fired first (BEFORE UPDATE) or at the very end (AFTER UPDATE). Then individual row triggers are fired before or after each row being examined. For row-level triggers, you can also add conditions that examine the row data to decide if the trigger should be fired or ignored. For instance, you might add a row trigger in the Salary case that fires only for employees in a certain division. Note that this condition is completely separate from the original UPDATE WHERE statement. The trigger condition is used only to decide whether or not to fire the trigger.

Figure 7.7 shows a sample trigger that fires whenever a row is changed in the Employee table. Notice that it is a row-level trigger because of the FOR EACH ROW statement. The example also illustrates that triggers can examine and use the data stored in the target table before it is changed (OLD ROW) and after it has been changed (NEW ROW). In this situation, the original salary and new salary are both recorded to the log table. With this information, security managers and auditors can quickly query the log table to identify major changes to salary and

```
UPDATE Employee
SET Salary = Salary + 10000
WHERE EmployeeID=442
OR EmployeeID=558
```

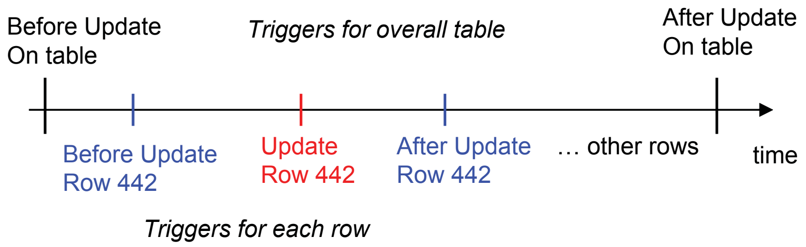


Figure 7.6

Update triggers can be assigned to the overall table and fire once for the entire command, or they can be assigned to fire for each row being updated.

then investigate further to ensure the changes were legitimate. You do have to be careful with the OLD and NEW data. For example, the NEW data has not yet been created in a BEFORE UPDATE trigger, so it cannot be accessed. Also, you cannot alter the OLD data within your trigger code.

Canceling Data Changes in Triggers

One of the uses of triggers is to examine changes in detail before they are written to the database. The BEFORE UPDATE and BEFORE INSERT triggers are often used to validate complex conditions. You also might want to provide more cautious checks before deleting data. In these cases, the structure of the trigger is straightforward. The key element is that you need a way to stop the original SQL statement from executing. The WHEN condition is used to examine the row that is scheduled to be deleted. As shown in Figure 7.8, the SIGNAL statement raises

Figure 7.7

Trigger to log the users who change an employee salary. The trigger fires any time the salary is updated, regardless of the method used to alter the data. It is a useful security tracing technique for sensitive data because it cannot be circumvented, except by the owner of the trigger.

```
CREATE TRIGGER LogSalaryChanges
AFTER UPDATE OF Salary ON Employee
REFERENCING   OLD ROW as oldrow
          NEW ROW AS newrow
FOR EACH ROW
          INSERT INTO SalaryChanges
          (EmpID, ChangeDate, User, OldValue, NewValue)
          VALUES
          (newrow.EmployeeID, CURRENT_TIMESTAMP,
          CURRENT_USER, oldrow.Salary, newrow.Salary);
```

```
CREATE TRIGGER TestDeletePresident
BEFORE DELETE ON Employee
REFERENCING OLD ROW AS oldrow
FOR EACH ROW
        WHEN (oldrow.Title = ‘President’)
                SIGNAL CANNOT_DELETE_PRESIDENT;
```

Figure 7.8

Canceling the underlying SQL command. This trigger examines the data for the employee row being deleted. The company always wants to keep data on any employee with the president title. The WHEN condition evaluates each row. The SIGNAL statement raises an error to prevent the underlying delete from executing.

an error condition that prevents the row from actually being deleted. The actual signal condition (CANNOT_DELETE_PRESIDENT) can be almost anything, but it must be defined as a constant in the overall module. Note that most database system vendors have not yet adopted the SIGNAL keyword, so the actual syntax you need will depend on the system (and version) that you are using. The workbooks give the actual cancel method and syntax needed for each specific DBMS. For instance, Oracle uses the function: Raise_Application_Error, whereas Microsoft SQL Server uses Raiserror.

In general, you should try to avoid using triggers for simple check conditions. Instead, use the standard SQL conditions (e.g., PRIMARY KEY, FOREIGN KEY, and CHECK) because they are more efficient and are less likely to cause additional problems. But sometimes you need to create complex conditions that are difficult to handle with simple conditions.

Cascading Triggers

A serious complication with triggers is that a database can have many triggers on each table. **Cascading triggers** arise when a change that fires a trigger on one table causes a change in a second table, that triggers a change in a third table, and so on. Figure 7.9 shows a common inventory situation. When an item is sold, a new row is added to the SaleItem table that contains the quantity sold. Because the item has been sold, the quantity on hand is updated in the Inventory table. A trigger on the Inventory table then checks to see if the QOH is below the reorder point. If it is, a new order is generated and sent electronically to a supplier, resulting in inserts on the Order and OrderItem tables.

There is nothing inherently wrong with cascading triggers. However, long chains of updates can slow down the system. They also make it difficult to debug the system and find problems. In the example, you might be looking at a problem in the OrderItem table, but it could have been caused by an error in the trigger code for the SaleItem table. The longer the chain, the more challenging it is to identify the source of problems.

A more difficult problem can potentially arise with cascading triggers. What happens when the chain loops on itself? Figure 7.10 shows an example of the problem. A company has embedded several rules about the methods of paying employees. When the salary reaches a certain level, the employee is eligible for bonuses. When the employee has already received substantial bonuses, the bonus amount is limited and the employee is granted additional stock options. If the lev-

Tables	Triggers and Timing

Sale(SaleID, SaleDate, …)
SaleItems(SaleID, ItemID, Quantity, …)	

	AFTER INSERT ON SaleItems
	   UPDATE Inventory
	   SET QOH = QOH – newrow.Quantity

Inventory(ItemID, QOH, …)	

	AFTER UPDATE ON Inventory
	   WHEN newrow.QOH < newrow.Reorder
	      INSERT {new Order}
	      INSERT {new OrderItem}

Order(OrderID, OrderDate, …)
OrderItem(OrderID, ItemID, Quantity, …)	

Figure 7.9

Cascading triggers. With triggers defined on multiple tables, a change in one table (SaleItem) can cascade into changes in other tables. Here, when an item is sold, quantity on hand is updated. If QOH is below the reorder point, a new order is generated and sent.

el of stock options is substantial, the original salary is reduced. But that takes the system back to the beginning, and the salary change could trigger another round of updates. Depending on the computations, this loop could diverge so that the numbers get larger and larger (or increasingly negative), and the computations never end. For this reason, the SQL standard is defined to forbid trigger loops. Systems that follow the standard are supposed to monitor the entire chain of updates, and if it encounters a loop, it should cancel changes and issue a warning. Even if the system is supposed to identify these loops, you should always check the system yourself to make sure that these problems will not arise. Obviously, the system is easier to check if there are only a limited number of triggers. If you can list the triggers in the order shown here, it is fairly easy to see the loop. However, systems rarely provide this option. Instead, you have to look through all of the database triggers and draw your own charts.

INSTEAD OF Triggers

Some database systems support the INSTEAD OF option as an even stronger type of trigger. A standard trigger runs your code in addition to performing the underlying function (DELETE, INSERT, or UPDATE). The INSTEAD OF option completely replaces the underlying command with your code. So, even if the change should be written to the database, you will have to write the additional SQL statements to take the appropriate action. Although this process seems more complicated, it is a useful trick for making queries updateable. Recall that a query that joins multiple tables generally is not updateable; data cannot be added to the query because the system does not always know which table gets the new row. To solve the problem, you can add an INSTEAD OF trigger to the query. Then, changes that are needed can be written to the individual tables with separate SQL statements

	Tables	Triggers and Timing
1	**Employee**(EID, Salary)	
		AFTER UPDATE IF newrow.Salary > 100000 THEN Add **BonusPaid** END
2	**BonusPaid**(EID, BonusDate, Amount)	
		AFTER UPDATE or INSERT IF newrow.Bonus > 50000 THEN Reduce Bonus Add **StockOptions** END
3	**StockOptions**(EID, OptionDate, Amount, SalaryAdj)	
		AFTER UPDATE Or INSERT IF newrow.Amount > 100000 THEN Reduce **Employee** Salary END
4		*Return to Step 1*

Figure 7.10

Trigger loop. Consider what happens when cascading triggers create a loop, where one trigger returns to alter a table that generated the original change. This loop would set up iterations that might converge or diverge. Even if the loop converges, it will eat up considerable resources.

Trigger Summary

Your first look at database triggers might seem overwhelming. Any table can contain trigger code before and after three different events. You can even write multiple triggers for each event. Do you really need to write database triggers? How do you determine which event to use? The first answer is that you should be conservative in using triggers. Use them to establish critical business rules and monitoring that need to be centralized. Database triggers are convenient and powerful, making it easy to ensure that relatively complex tasks are handled correctly. However, they are difficult to debug and explain to other developers.

The answer to the second question is trickier. You first need to understand the detailed nature of the business rule. Choose the database trigger that provides the most direct application of the rule. For example, if you need a rule related to changing inventory levels, add the trigger to the Items table; not the SaleItems table. When in doubt, write the rule in several locations and test each version. One of the main indicators of success is when your rule fires exactly one time. If the rule does not fire during a test run, it is probably too far away from the desired table. If it fires repeatedly for one business operation, the rule is at too detailed of a level (such as on the SaleItems table instead of the Sale table).

Steps	Savings Balance	Checking Balance
0. Start	5,340.92	1,424.27
1. Subtract 1,000	4,340.92	1,424.27
2. Add 1,000	4,340.92	2,424.27
Problem arises if transaction is not completed		
1. Subtract 1,000	4,340.92	1,424.27
2. Machine crashes		1,000 is gone

Figure 7.11

Transactions involve multiple changes to the database. To transfer money from a savings account to a checking account, the system must subtract money from savings and add it to the checking balance. If the machine crashes after subtracting the money but before adding it to checking, the money will be lost.

Transactions

How does a DBMS handle multiple transaction events? When building applications, it is tempting to believe that components will always work and that problems will never occur. Tempting, but wrong. Even if your code is correct, problems can develop. You might face a power failure, a hardware crash, or perhaps someone accidentally unplugs a cable. You can minimize some of these problems by implementing backup and recovery procedures, storing duplicate data to different drives, and installing an uninterruptible power supply (UPS). Nevertheless, no matter how hard you try, failures happen.

A Transaction Example

An error that occurs at the wrong time can have serious consequences. In particular, many business operations require multiple changes to the database. A **transaction** is defined as a set of changes that must all be made together. Consider the example in Figure 7.11. You are working on a system for a bank. A customer goes to an online banking application and instructs it to transfer $1,000 from savings to a checking account. This simple transaction requires two steps: (1) subtracting the money from the savings account balance and (2) adding the money to the checking account balance. The code to create this transaction will require two updates to the database. For example, there will be two SQL statements: one UPDATE command to decrease the balance in savings and a second UPDATE command to increase the balance in the checking account.

You have to consider what would happen if a machine crashed in between these two operations. The money has already been subtracted from the savings account, but it will not be added to the checking account. It is lost. You might consider performing the addition to checking first, but then the customer ends up with extra money, and the bank loses. The point is that both changes must be made successfully. The other option is that both operations can fail—leaving the customer and the bank at the starting point. If you have a choice, you want all operations to succeed, but keep in mind that total failure is better than partial success in these cases.

```
CREATE FUNCTION TransferMoney(Amount Currency,
            AccountFrom Number,AccountTo Number)
      RETURNS NUMBER
curBalance Currency;
BEGIN
      DECLARE HANDLER FOR SQLEXCEPTION
      BEGIN
            ROLLBACK;
            Return -2;                -- flag for completion error
      END;
      START TRANSACTION;    -- optional
      SELECT CurrentBalance INTO curBalance
      FROM Accounts WHERE (AccountID = AccountFrom);
      IF (curBalance < Amount) THEN
            RETURN -1;        -- flag for insufficient funds
      END IF
      UPDATE Accounts
      SET CurrentBalance = CurrentBalance – Amount
      WHERE AccountID = AccountFrom;
      UPDATE Accounts
      SET CurrentBalance = CurrentBalance + Amount
      WHERE AccountID = AccountTo;
      COMMIT;
      RETURN 0;                       -- flag for success
END;
```

Figure 7.12

Transaction to transfer money. If the system crashes before the end of the transactions (Commit), none of the changes are written to the database. On restart, the changes may all be rolled back, or the transaction restarted.

Starting and Ending Transactions

How do you know that both operations are part of the same transaction? It is a business rule—or the definition of a transfer of funds. The real problem is: How does the computer know that both operations must be completed together? As the application developer, you must tell the computer system which operations belong to a transaction. To do that you need to create procedural code and mark the start and the end of all transactions inside your code. When the computer sees the starting mark, it starts writing all the changes to a log file. When it reaches the end mark, it makes the actual changes to the data tables. If something goes wrong before the changes are complete, when the DBMS restarts, it examines the log file and completes any transactions that were incomplete. From a developer's perspective, the nice part is that the DBMS handles the problem automatically. All you have to do is mark the start and the end of the transaction.

Transactions illustrate the need for procedural languages. As shown in Figure 7.12, the multiple UPDATE statements need to be stored in a module function or procedure. In this example, the two UPDATE statements must be completed together or fail together. The START TRANSACTION statement is optional (in the SQL standard) but highlights the beginning of the transaction. If both updates complete successfully, the COMMIT statement executes, which tells the DBMS

```
START TRANSACTION;
SELECT …
UPDATE …
SAVEPOINT StartOptional;
UPDATE …
UPDATE …
If error THEN
          ROLLBACK TO SAVEPOINT StartOptional;
END IF
COMMIT;
```

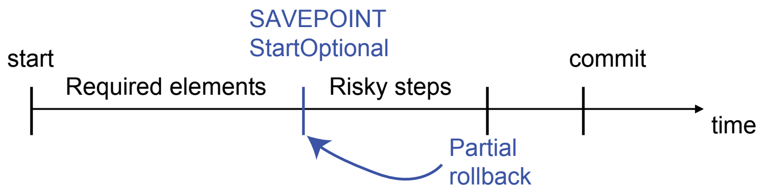


Figure 7.13

SAVEPOINT. A SAVEPOINT enables you to rollback to an intermediate point in the procedure. You can set multiple SAVEPOINTS and choose how far back you want to rollback the changes.

to save all of the changes. If an unexpected error arises, the ROLLBACK statement executes so none of the changes are saved. Most systems handle the transaction requirement by writing all changes to an intermediate log file. If something goes wrong with the transaction, the system can recover the log file and rollback or complete the transaction.

Notice that the START TRANSACTION line comes before the initial SELECT statement. This might seem unnecessary, since it appears that only the UPDATE commands need to be within the transaction. There is a syntax reason for placing this statement first: Any SELECT statement automatically initiates a new transaction. However, as will be explained in the section on concurrency, there is a good reason for starting the transaction before this SELECT statement. Think about things that can go wrong if another process tries to modify the data retrieved by the SELECT statement, before this transaction is finished.

SAVEPOINT

Sometimes, you need intermediate points in a transaction. Some steps are more critical than others. You might have some optional changes that would be useful to save, but if they fail, you still need to ensure that the critical updates are committed. The SAVEPOINT technique divides transaction procedures into multiple pieces. You can roll back a transaction to the beginning, or to a specific SAVEPOINT. Figure 7.13 illustrates the process and shows the syntax to set a SAVEPOINT and rollback to it. As indicated, it can be used to mark a set of risky steps that you would like to include in the update but are not required to use. Consequently, if the updates fail for the risky section, you can discard those changes and still keep the required elements that were defined at the beginning of the transaction. Generally, you could accomplish the same thing by using multiple COMMIT statements, but sometimes the optional code might include a calculation that you want to include in the final result. Without the SAVEPOINT option, you might have to write the final value more than once.

Receive Payment	Balance	Place New Order
1. Read balance 800	800	
2. Subtract Pmt. -200		
		3. Read balance 800
4. Save balance 600	600	
		5. Add order 150
	950	6. Write balance 950

Figure 7.14

Concurrent access. If two processes try to change the same data at the same time, the result will be wrong. In this example the changes made when the payment is received are overwritten when a new order is placed at the same time.

Multiple Users and Concurrent Access

How do you prevent problems arising when two processes change the same data? One of the most important features of a database is the ability to share data with many users or different processes. This concept is crucial in any modern business application: Many people need to use the application at the same time. However, it does create a potential problem with database integrity: What happens when two people try to change the same data at the same time? This situation is known as **concurrent access**. Consider the example of an Internet order system shown in Figure 7.14. The company records basic customer data and tracks charges and receipts from customers. Customers can have an outstanding balance, which is money they currently owe. In the example, Jones owes the company $800. When Jones makes a payment, a clerk receives the payment and checks for the current balance ($800). The clerk enters the amount paid ($200), and the computer subtracts to find the new balance due ($600). This new value is written to the customer table, replacing the old value. So far, no problem. A similar process occurs if Jones makes a new purchase. As long as these two events take place at different times, there is no problem.

However, what happens if the two transactions do occur together? Consider the following intermingling: (1) The payments clerk receives the payment, and the computer retrieves the current amount owed by Jones ($800). (2) The clerk enters the $200 payment. Before the transaction can be completed, Jones places a new order on the Internet for $150 of new merchandise. (3) The Web server also reads the current balance owed ($800) and adds the new purchases. Now, before this transaction can be completed, the first one finishes. (4) The payments clerk's computer determines that Jones now owes $600 and saves the balance due. (5) Finally, the Web server adds the new purchases to the balance due. (6) The order computer saves the new amount due ($950). Customer Jones is going to be justifiably upset when the next bill is sent. What happened to the $200 payment? The answer is that it was overwritten (and lost) when the new order change was mixed in with the receipt of the payment.

Receive Payment	Balance	Place New Order
1. Read balance 800	800	
2. Subtract Pmt. -200		
		3. Read balance 800
4. Save balance 600	600	Error: Blocked
		3. Read balance 600
		4. Add order 150
	950	5. Write balance 750

Figure 7.15

Serialization. The first process locks the data so that the second process cannot even read it. Concurrent changes are prevented by forcing each process to wait for the earlier ones to be completed.

Optimistic Locks

Two common methods exist to solve the problem of concurrent changes (optimistic and pessimistic). Today, with fast computer speeds the DBMS can process transactions quickly so there is a lower probability of concurrency problems. An **optimistic lock** begins with the assumption that collisions are rare and unlikely to arise. If they do arise, it is easier to handle the situation at that time. Handling problems is straightforward and takes less DBMS overhead. Particularly in distributed database environments, it is often easier and faster to use optimistic locking.

The key to understanding optimistic locks is to realize that they are not really locks; the DBMS lets your program read any piece of data needed. When your program attempts to change the data, the DBMS rereads the database and compares the currently stored value to the one it gave you earlier. If there is a difference between the two values, it signifies a concurrency problem because someone else changed the data before you were able to finish your task. The DBMS then raises an error and expects your program to deal with it. In summary, optimistic locking can improve performance, but it requires you to deal with potential collisions. Figure 7.15 outlines the basic process. The key to the process lies in modifying the UPDATE command by adding a WHERE clause similar to: WHERE Amount = oldAmount. The “oldAmount” value is the original value stored in a variable when the transaction begins.

The preferred solution to collisions using optimistic locks is to rollback any changes you have already made, and restart your code to read the current value from the database, re-compute your changes, and write the new value to the database. Consider the example of the orders in Figure 7.16. The function first reads the current value of the balance into memory. After completing some other tasks (slow code), it attempts the UPDATE command, with one twist. It specifies that the UPDATE command only applies to the row with the given Account Number **and with the original Amount value**. If the value was changed by a second transaction, this UPDATE command will not alter any rows. The error test following the UPDATE command will recognize if the changes were successful or not. If successful, the routine is done and it exits. If the changes failed, you have

```
CREATE FUNCTION ReceivePayment (
        AccountID NUMBER, Amount Currency) RETURNS NUMBER
BEGIN
        DECLARE HANDLER FOR SQLEXCEPTION
        BEGIN
                ROLLBACK;
                RETURN -2;
        END
        SET TRANSACTION SERIALIZABLE, READ WRITE;
        UPDATE Accounts
        SET AccountBalance = AccountBalance - Amount
        WHERE AccountNumber = AccountID;
        COMMIT;
        RETURN 0;
END
```

Figure 7.16

Transaction to transfer money. If the system crashes before the end of the transactions (Commit), none of the changes are written to the database. On restart, the changes may all be rolled back, or the transaction restarted.

complete control over what to do. In this case, it makes sense to go back and pick up the newly revised Amount and try again. To be safe, you should add a counter to the number of retries. If the count reaches too large of a number, this routine should simply give up and produce an error code indicating that it is not possible to update the data at this time.

One catch with the UPDATE command is that you have to be careful with Null values. Recall from queries that a condition of the form Amount = Null will not work correctly. Instead, you have to write Amount Is Null. Consequently, if the original value might be missing, the comparison test is more complicated:

((Amount = oldAmount) OR (Amount IS Null AND oldAmount IS Null))

One of the strengths of the optimistic approach is that it works with any DBMS, even if multiple distributed databases are involved in the transactions. However, it does require that programmers write and validate the proper code for every single update. Consequently, it makes sense to create a code library that contains a generic version of the UPDATE command that can be called for almost any transaction.

The other powerful feature of this approach is that the program code can contain relatively sophisticated analysis to automatically handle common update problems. The other optiona of a pessimistic lock usually just blocks or delays a transaction which forces users to slow down or solve problems themselves. On the other hand, the optimistic lock realized that it simply had to get the new balance and use it to compute the final amount. No intervention and almost no delay were involved.

Today it is possible to reduce the collisions and concurrent access issues. Focus on using the DBMS to handle all updates. Avoid computing values in code or on forms. Consider Web-based forms which are notoriously slow. The form shows customer account data to a clerk. The clerk enters a value for a payment receipt. If this value is added to the current balance on the form or on the Web server, it runs the risk of a collision when the total is written back to the DBMS. This col-

Process 1	Data A	Data B	Process 2
1. Lock Data A			
	Locked By 1		2. Lock Data B
3. Wait for Data B		Locked By 2	
			4. Wait for Data A

Figure 7.17

Deadlock. Process 1 has locked Data A and is waiting for Data B. Process 2 has locked Data B and is waiting for Data A. To solve the problem, one of the processes has to back down and release its lock.

lision can be avoided by computing the total within the DBMS using the update statement:

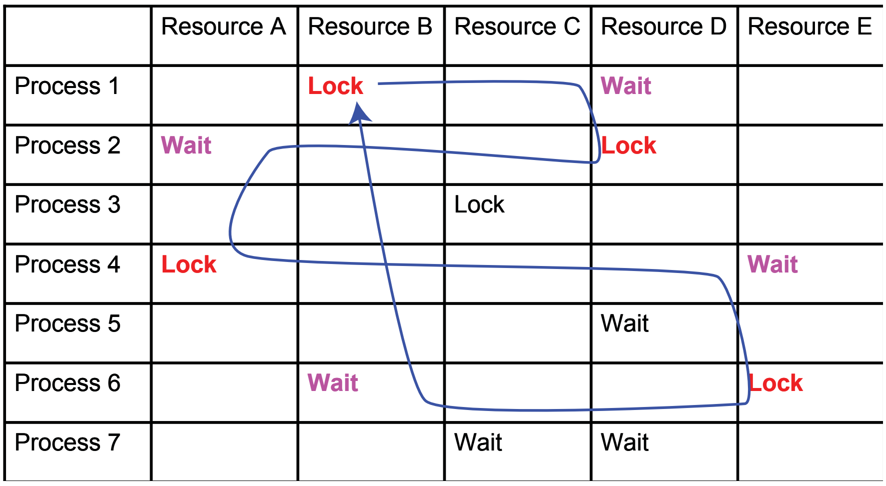
```
UPDATE Customer
SET Balance = Balance + NewValue
WHERE CustomerID=@CustomerID;
```

The DBMS simply adds the new value to whatever total currently exists in the table. Your code does not need to test for concurrency issues. Of course, a DBMS running parallel processors (and multithreading) would have to internally monitor concurrency issues when running multiple update commands at the same time. But that work is handled by the DBMS vendor.

The other way to minimize concurrency issues is to avoid storing any totals. Transaction changes are simply written to a table along with time stamps. Totals are computed from this log table whenever they are needed. However, in some

Figure 7.18

Lock manager. A global lock manager tracks all locked resources and associated processes. If it detects a cycle, then a deadlock exists, and the lock manager instructs processes to release locks until the problem is solved.



	Resource A	Resource B	Resource C	Resource D	Resource E
Process 1		Lock		Wait	
Process 2	Wait			Lock	
Process 3			Lock		
Process 4	Lock				Wait
Process 5				Wait	
Process 6		Wait			Lock
Process 7			Wait	Wait	

situations you still want to monitor concurrency. For instance, you do not want two people to buy the last seat on an airplane.

Pessimistic Locks: Serialization

A second solution to the problem of concurrent access is to prevent collisions by forcing transactions to be completely isolated. As shown in Figure 7.17, the **serialization** process forces transactions to run separately so that a second process cannot even read the data being modified by the first process. The first process requests a lock on the balance. Any process that attempts to read that data before the lock is released will receive an error message. A key feature in this approach is the ability of the DBMS to set row-level locks to minimize interference with other processes. Some early systems used table-level locks, so no one could read the data while one balance was being updated!

The method of invoking this type of lock mechanism depends heavily on the DBMS. SQL 99 defined a standard method of specifying the transaction lock, but it has not been widely implemented yet. Figure 7.18 shows the basic logic, but keep in mind that the syntax will be different for each DBMS. The main step is to specify the **isolation level** to SERIALIZABLE in the SET TRANSACTION statement. The DBMS then knows to lock each data element you will be using so that other transactions will be prevented from reading the data until the first changes have been committed. However, it is important that all of the transaction procedures contain error-handling code. Otherwise, when the second transaction (RecordPurchase is almost identical to this one) runs, it will crash and display a cryptic error message when it tries to update or read the data.

The concept of serialization is logical, and it emphasizes the importance of forcing each transaction to complete separately. However, it is based on the technique of a **pessimistic lock**—where each transaction assumes that concurrent interference will always occur. Every time the transaction runs, it places locks on all of the resources that will be needed. This technique slows down the processing and can result in another serious problem described in the following section.

Multiuser Databases: Concurrent Access and Deadlock

Concurrent access is a problem that arises when two processes attempt to alter the same data at the same time. When the two processes intermingle, generally one of the transactions is lost and the data becomes incorrect. For most database operations the DBMS handles the problem automatically. For example, if two users open forms and try to modify the same data, the DBMS will provide appropriate warnings and prevent the second user from making changes until the first one is

Figure 7.19

Optimistic locking process. The steps assume that concurrency problems will not arise. If another transaction does change the data before this transaction finishes, the code receives an error message and must restart.

1. Read the balance.
2. Add the new order value.
3. Write the new balance.
4. Check for errors.
5. If errors exist, return to step 1.

```
CREATE FUNCTION ReceivePayment (
        AccountID NUMBER, Amount Currency) RETURNS NUMBER
oldAmount Currency;
testEnd Boolean = FALSE;
BEGIN
        DO UNTIL testEnd = TRUE
        BEGIN
                SELECT Amount INTO oldAmount
                WHERE AccountNumber = AccountID;
                …
                UPDATE Accounts
                SET AccountBalance = AccountBalance - Amount
                WHERE AccountNumber = AccountID
                AND Amount = oldAmount;
                COMMIT;
                IF SQLCODE = 0  And nrows > 0 THEN
                        testEnd = TRUE;
                        RETURN 0;
                END IF
                -- keep a counter to avoid infinite loops
        END
END
```

Figure 7.20

Optimistic concurrency with SQL. Keep the starting value within memory and then only do the update if that value is unchanged. If another transaction changed the data before this one completes, go back and get the new value and start over.

finished. Similarly, two SQL operations (e.g., UPDATE) will not be allowed to change the same data at the same time.

Even if you write program code, the DBMS will not allow two processes to change the same data at the same time. However, your code has to understand that sometimes a change to the data will not be allowed. This condition is often handled as an error.

The solution to the concurrency problem is to force changes to each piece of data to occur one at a time. If two processes attempt to make a change, the second one is stopped and must wait until the first process finishes. The catch is that this forced delay can cause a second problem: deadlock. **Deadlock** arises when two (or more) processes have placed locks on data and are waiting for the other's data. An example is presented in Figure 7.19. Process 1 has locked data item A. Process 2 has locked item B. Unfortunately, Process 1 is waiting for B to become free, and Process 2 is waiting for A to be released. Unless something changes, it could be a long wait.

Two common solutions exist for the deadlock problem. First, when a process receives a message that it must wait for a resource, the process should wait for a random length of time, try again, release all existing locks, and start over if it still cannot obtain the resource. This method works because of the random wait. Of the two deadlocked processes, one of them will try first, give up, and release all locks with a ROLLBACK statement. The release clears the way for the other process to complete its tasks. This solution is popular because it is relatively easy to program. However, it has the drawback of causing the computer to spend a lot of time waiting—particularly when there are many active processes, leading to many collisions.

A better solution is for the DBMS to establish a global lock manager as shown in Figure 7.20. A lock manager monitors every lock and request for a lock (wait). If the lock manager detects a potential deadlock, it will tell some of the processes to release their locks, allow the other processes to proceed, and then restart the other processes. It is a more efficient solution, because processes do not spend any time waiting. On the other hand, this solution can be implemented only within the DBMS itself. The lock manager must be able to monitor every process and its locks.

For typical database operations with forms and queries, the DBMS handles concurrent access and deadlock resolution automatically. When you write code to change data, the DBMS still tries to handle the situation automatically. However, the DBMS may rely on you to back out your transaction. Some systems may simply generate an error when the second process attempts to access the data, and it is your responsibility to catch the error and handle the problem.

This section focuses on terms used in computer science and the SQL standards. They are not critical for beginning students.

ACID Transactions

What are the primary rules to ensure integrity of transactions? The concept of integrity is fundamental to databases. One of the strengths of the database approach is that the DBMS has tools to handle the common problems. In terms of transactions, many of these concepts can be summarized in the acronym ACID. Figure 7.21 shows the meaning of the term. **Atomicity** represents the central issue that all parts of a transaction must succeed or fail together. **Consistency** means that all data in the database ultimately must be consistent. Even though there might be temporary inconsistencies while a transaction is being processed, in the end, the database must be returned to a consistent state. This status should be able to be tested with application-defined code. For example, referential integrity must be maintained after a transaction is completed. **Isolation** means that concurrent access problems are prevented. Changes by one transaction do not result in errors in other transactions. Note that transactions are rarely completely isolated:

Figure 7.21

ACID transactions. The acronym highlights four of the main integrity features required of transactions.

- **Atomicity**: All changes succeed or fail together.
- **Consistency**: All data remain internally consistent (when committed) and can be validated by application checks.
- **Isolation**: The system gives each transaction the perception that it is running in isolation. There are no concurrent access issues.
- **Durability**: When a transaction is committed, all changes are permanently saved even if there is a hardware or system failure.

	ItemID	QOH	Price
→	**111**	**5**	**15**
	113	6	7
	117	12	30
→	**118**	**4**	**12**
	119	7	22
→	**120**	**8**	**17**
→	*121*	*7*	*16*
→	*122*	*3*	*14*

```
SELECT SUM(QOH)
FROM Inventory
WHERE Price Between 10 And 20

Result: 5 + 4 + 8 = 17
```

```
INSERT INTO Inventory
VALUES (121, 7, 16)
INSERT INTO Inventory
VALUES (122, 3, 14)
```

```
SELECT SUM(QOH)
FROM Inventory
WHERE Price Between 10 And 20

Result: 5 + 4 + 8 + 7 + 3 = 27
```

Figure 7.22

Phantom rows. The first SELECT statement will select only three rows of data. When the second transaction runs, additional rows will match the criteria, so that the second time the query runs, it will return a different result, because it includes the phantom rows.

they might encounter pessimistic or optimistic locking messages that need to be handled. **Durability** indicates that committed transactions are lasting. Once the transaction commits a change, it stays changed. This concept is critical in the face of hardware and software failures and is more difficult to maintain in a distributed database environment. Most systems ensure durability by writing changes to a log file. Then, even if a hardware failure interrupts an update, the changes will be finished when the system is restarted. Importantly, once the COMMIT statement is accepted, the DBMS cannot rollback the changes.

With SQL 99, the START TRANSACTION and SET TRANSACTION commands can be used to set the isolation level. In increasing isolation order, the four choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. These levels are supposed to be used to prevent different types of concurrency problems, but rarely is there a need for the intermediate levels, so many systems provide only the first and last.

The READ UNCOMMITTED level provides almost no isolation. It enables your routine to read data that another transaction has altered but not yet committed. This problem is sometimes called dirty read because the value you receive might be rolled back and the value ultimately may be inaccurate. If you select this level, SQL will not allow your transaction to update any data, because it might spread a false number throughout the database. The READ COMMITTED level is similar to optimistic concurrency. It will prevent your transaction from reading uncommitted data, but the data might still be changed or deleted by another transaction before the first transaction completes.

The REPEATABLE READ level prevents specific data you are using from being changed or deleted, but does not resolve the problem of phantom data. As shown in Figure 7.22, consider a transaction that computes the sum of quantity on hand if the price of an item falls within a specified range. Now, a second transaction is started before the first one completes. This command inserts rows of data (or alters the prices). These new rows are phantom rows that are not included in the first query because they did not exist when the query began. After the first two queries have finished, if you repeat the first query, the phantom rows will be committed and you will see new results.

Are phantom rows bad? In many ways, no; they simply arise because a database has constantly changing data. You (and managers) must always remember that the results of a query are accurate only at a specific point in time. On the other hand, if you are writing procedural code, you might be surprised by the results when your queries do not finish in the order you expected—particularly if the DBMS is running on a multiprocessor system. In these situations, you might have to add semaphores or repeat queries to ensue your code follows a specific sequence. Alternatively, you can specify a higher level of isolation.

The SERIALIZABLE isolation level prevents the phantom row problem by ensuring that all transactions behave as if they were run in sequence. However, keep in mind that this result is usually accomplished through the use of locks, so it requires database resources, and it does not guarantee that your transaction will be able to finish on the first try. You still need error handling to catch and resolve the problem when your transaction is blocked by another one.

Key Generation

How are key values generated? As you know by now, the relational database relies heavily on primary keys, which must be unique. It can be difficult in business to guarantee that these keys are always created correctly. Hence, most relational databases have a mechanism to generate numeric keys that are unique. Although these methods work reasonably well for simple projects, you will eventually learn that generated key values present some challenges that must be handled with programming. Also, bear in mind that each DBMS uses a different mechanism to generate keys.

The main problem you encounter with generated keys is when you want to add a row to one table and then insert the matching key value into a second table. For example, when you add a new Customer, the system generates a CustomerID, which you need to insert into the Order table. Figure 7.23 shows the basic problem: the CustomerID key generated to create the new customer must be kept by the transaction procedure so that the key can be inserted into the Order table. The diverse ways of handling the number creation make the problem more difficult.

Logically, generated keys could be created through two primary methods: (1) by an automatic method when a new row is added to a table, and (2) by a separate

Figure 7.23

Generated keys. Creating an order for a new customer requires generating a CustomerID key that is used in the Customer table and must be stored so it can be used in the Order table.

1. Generate key for CustomerID. 2. INSERT row into Customer.	Customer Table CustomerID, Name, …
3. Generate key for OrderID. 4. INSERT row into Order, using new OrderID and CustomerID.	Order Table OrderID, CustomerID, …

1. INSERT row into Customer.
2. Get the key value that was generated.
3. Verify the key value is correct.
4. INSERT row into Order.

Figure 7.24

Auto-generated keys. The process seems relatively easy when the DBMS automatically generates keys. However, what happens at step 2 if two transactions generate a new key value on the same table at almost the same time?

key generation routine. The advantage of the first method is that the process of adding a row to the initial (Customer) table is relatively simple. The drawback is that it is tricky to make sure you get the correct generated key to use in a second table. The second method solves the second problem, but makes it more difficult to create keys and requires programmers to ensure that the process is followed for every table and insertion operation.

As shown in Figure 7.24, if the DBMS automatically generates key values for each table, the code seems relatively simple. Microsoft Access and SQL Server use this approach. The complication is that problems arise when two transactions generate new key values on the same table at almost the same time. Or, when one transaction triggers inserts into multiple tables. You need to be careful that your code retrieves the correct key value. With some systems, it is difficult to verify the value is correct. You might have to use a SELECT INTO statement to retrieve the customer data and double-check the name and phone number.

Because of the difficulties in obtaining an auto-generated key value, the second approach of calling a key generation routine has some benefits. This approach is primarily used by Oracle. Figure 7.25 shows the basic steps needed to create an order for a new customer. Notice that there is no uncertainty about the key value generated. The generation routine ensures that values are unique—even if two transactions request values at the same time. The drawback to this approach is that it is not automatic. However, it is straightforward to write trigger code for the main table (Customer) to generate a new ID for use whenever an INSERT is performed on the table.

Database Cursors

How can procedural code track row-by-row through a query? To this point, all of the procedures and functions have dealt with either DML statements or single-row SELECT statements. These statements either do not return values or they return only one row of data. This restriction simplifies the program logic and

Figure 7.25

Key-generation routine. The steps are not difficult, but programmers must add them for every table and every routine that inserts data.

1. Generate a key for CustomerID
2. INSERT row into Customer
3. Generate a key for OrderID
4. INSERT row into Order

```
DECLARE cursor1 CURSOR FOR
        SELECT AccountBalance
        FROM Customer;
sumAccount, balance Currency;
SQLSTATE Char(5);
BEGIN
        sumAccount = 0;
        OPEN cursor1;
        WHILE (SQLSTATE = '00000')
        BEGIN
                FETCH cursor1 INTO balance;
                IF (SQLSTATE = '00000') THEN
                        sumAccount = sumAccount + balance;
                END IF
        END
        CLOSE cursor1;
        -- display the sumAccount or do a calculation
END
```

Figure 7.26

SQL cursor structure. DECLARE, OPEN, FETCH, and CLOSE are the main statements in the SQL standard.

makes it easier to learn the foundations of SQL procedures. However, some applications will require more sophisticated queries: SELECT statements that return multiple rows of data.

Remember that SQL commands operate on sets of data—multiple rows at one time. What if you want more precise control? Perhaps you need to examine one row at a time to perform a complex calculation, compare some data from an external device, or display the row to the user and get a response. Or perhaps you need to compare one row of data to a second row. For example, you might want to subtract values across two rows. It is difficult to accomplish these tasks with standard SQL commands. As noted in Chapter 9, newer versions of SQL are adding features to perform even these tasks with straight SQL commands. However, you will still find times where you want to track through query results one row at a time.

Cursor Basics

SQL has a process that enables you to track through a set of data one row at a time. You create a **database cursor** that defines a SELECT statement and then points to one row at a time. A loop statement enables you to move the cursor to the next row and repeat your code to examine each row returned by the query. You can also move the cursor back to previous rows, but this process requires more overhead and is rarely needed.

Figure 7.26 shows the basic structure of a procedure to create a cursor and loop through the Customer table to calculate the total amount of money owed. Of course, this particular calculation can be done easier and faster with a simple SELECT statement. The goal here is to show the main structure of the code needed to implement a database cursor. The DECLARE CURSOR statement defines the SELECT statement that retrieves the rows to be examined. Although the example

```
DECLARE cursor2 SCROLL CURSOR FOR
SELECT …
OPEN cursor2;
FETCH LAST FROM cursor2 INTO …
Loop…
        FETCH PRIOR FROM cursor2 INTO …
End loop
CLOSE cursor2;
```

Figure 7.27

FETCH options. A scrollable cursor can move in either direction. This code moves to the last row and then moves backward through the table. Other FETCH options include FIRST, ABSOLUTE, and RELATIVE.

uses only one column, you can use any common SELECT statement including multiple columns, WHERE conditions, and ORDER BY lines. You must OPEN the cursor to use it, and eventually should CLOSE the cursor to free up database resources. When a cursor is first opened, it points to a location immediately before the first row of data. The FETCH statement retrieves one row of data and places the columns of data for that row into program variables. A loop is necessary to track through each row that matches the selection conditions.

Scrollable Cursors

By default, the FETCH command picks up the next row. If the FETCH command pushes the cursor past the end of the dataset, an error condition is created. You can use the WHENEVER statement to catch the specific error, or you can examine the SQLSTATE variable to see if an error was generated with the last SQL statement. A string value of five zeros indicates that the last command was successful.

Several options are available for the FETCH command to move the cursor to a different row. The common options are NEXT, PRIOR, FIRST, and LAST. These retrieve the indicated row. Figure 7.27 outlines the cursor procedure that begins at the last row and moves up to the first row. Note that you must declare the cursor as scrollable with the SCROLL keyword. Of course, it would be more efficient to simply sort the data in reverse order and then move forward; but the objective is

Figure 7.28

Transaction concurrency in cursor code. Your cursor code has tracked down through the data to Carl. It then tries to go back to the prior row with FETCH PRIOR. But, if another transaction has inserted a new row (Bob) in the meantime, your code will pick up that one instead of the original (Alice).

Original Data		Cursor	Modified Data		Insert
Name	Sales		Name	Sales	
Alice	444,321	1. Read Alice	Alice	444,321	
Carl	254,998	2. Read Carl	Bob	333,229	3. Bob inserted by
Donna	652,004		Carl	254,998	second process
Ed	411,736	4. Move Prior	Donna	652,004	
		but get Bob	Ed	411,736	
		instead of Alice			

Year	Sales	Gain
2000	151,039	
2001	179,332	
2002	195,453	
2003	221,883	
2004	223,748	

Figure 7.29

Sales analysis table. A standard SELECT query can compute and save the sales total by year. You now need to write a cursor-based procedure to compute the sales gain from the prior year.

to show that you can move in either direction. Additional FETCH scroll options include the ability to move to the first row (FETCH FIRST) and to jump to a specific row in the dataset. For example, FETCH ABSOLUTE 5 will retrieve the fifth row in the dataset. Since you rarely know the exact row number to retrieve, the relative scroll option is more useful. For instance, FETCH RELATIVE -3 skips back three rows from the current position.

The ability to move backward in the list of rows highlights another transaction concurrency issue. What happens if you work your way down a set of rows and issue the FETCH PRIOR command? Most of the time, you would simply retrieve the row before the current one. But what happens if another transaction inserts a new row immediately before the FETCH PRIOR command is executed? Figure 7.28 shows the problem. Your code has tracked down to Carl, but a second process has inserted Bob into your list. The FETCH PRIOR command will return data for Bob instead of the data for Alice that you expected to see. The SQL standard solution to this problem is to make the dataset insensitive to other changes. You simply add a keyword to the cursor declaration (DECLARE cursor3 INSENSITIVE CURSOR FOR …). Effectively, the DBMS copies the results of the query into a temporary table that is not affected by other commands. Although this approach will work, it can be an expensive use of database resources. Instead, be sure to ask yourself why you need to move backward. In most cases, you will find that it is unnecessary. For example, if you want to calculate differences by subtracting the value on the current row from the value on the prior row, simply store the "prior" value in memory, then fetch the next row and perform the subtraction. There is no need to move backwards and risk getting the wrong value.

You might notice that there is no procedure to find a row within the retrieved dataset and move the cursor to that row (such as a SEEK command). Although some systems provide this feature, it is rarely needed. Instead, you should create the WHERE condition to only retrieve exactly the rows you want.

Changing or Deleting Data with Cursors

A common situation that a cursor-based application encounters is the need to change or delete the data at the current row. For example, Figure 7.29 shows a table created to hold sales data for analysis. A standard SELECT command with a GROUP BY clause can compute the sales totals by year. You need to write a cursor-based procedure to compute the increase (or decrease) in sales for each year.

```
DECLARE cursor1 CURSOR FOR
SELECT Year, Sales, Gain
FROM SalesTotal
ORDER BY Year
FOR UPDATE OF Gain;
priorSales, curYear, curSales, curGain
BEGIN
        priorSales = 0;
        OPEN cursor1;
        Loop:
                FETCH cursor1 INTO curYear, curSales, curGain
                UPDATE SalesTotal
                SET Gain = Sales – priorSales
                WHERE CURRENT OF cursor1;
                priorSales = curSales;
        Until end of rows
        CLOSE cursor1;
        COMMIT;
END
```

Figure 7.30

Cursor code for update. The FOR UPDATE option in the declaration enables the Gain column to be changed. The WHERE CURRENT OF statement specifies the row pointed to by the cursor.

Figure 7.31

Parameterized cursor query. Your code sets the value of maxPrice through user input or calculation or another query. When this cursor is opened, the value is applied to the SELECT statement and only the matching rows are returned.

```
DECLARE cursor2 CURSOR FOR
SELECT ItemID, Description, Price
FROM Inventory
WHERE Price < :maxPrice;
maxPrice Currency;
BEGIN
        maxPrice = …      -- from user or other query
        OPEN cursor2;     -- runs query with current value
        Loop:
                -- Do something with the rows retrieved
        Until end of rows
        CLOSE cursor2;
END
```

The catch is that you need to store this computed value back into the table. To do that, you need to specify that the cursor is updateable, and then write an UPDATE statement that stores the calculation in the row currently pointed to by the cursor. Figure 7.30 shows the main code needed to perform the calculations.

Notice that the cursor declaration states that only the Gain column is updateable. This option protects the database slightly. If you make a mistake or someone else modifies your code later, the DBMS will allow only the Gain column to be changed. An attempt to change the Year or Sales column will generate an error. The other important element is the *WHERE CURRENT OF cursor1* statement. This condition states that the row currently fetched, or pointed to by the cursor, is the one to be changed. The UPDATE statement will apply only to this row. An almost identical statement can be used to delete the current row (DELETE FROM SalesTable WHERE CURRENT OF cursor1).

Cursors with Parameters

Occasionally, you need a more dynamic query, where you want to pick the specific rows based on some variable within your procedure. For example, a user might enter a price, or your program compute a price based on some other query. Then, you want to retrieve only the rows that are less than the specified price and perform some computation on those rows. You can enter local variables as parameters in the cursor query. Figure 7.31 shows the basic elements of the parameterized cursor. You enter the name of a variable within the cursor's SELECT statement. Within the procedure, you assign a value to this variable. The value might be computed from other variables, input by the user, or even retrieved from a different cursor or query. When the parameterized cursor is opened, the current value is substituted into the query, so that it returns only the rows that match the request. Parameterized queries in the cursor provide powerful tools to dynamically evaluate data automatically in response to other changes.

Be aware that each DBMS uses a different notation to indicate parameters. The standard uses a colon in front of the variable name (:MyVar). SQL Server uses an "at" sign (@MyVar). Oracle does not use any characters in front of the parameter variable, but requires a colon in assignment statements (MyVar := 100). Microsoft Access does not require any special notation. The benefit to marking parameters is that it makes them easier to spot when reading code written by others. When you work in systems without the notation, you might want to adopt a policy of naming parameters and variables to make them easier to recognize (such as v_MyVar).

Figure 7.32

Processing inventory changes. When an item is sold, the quantity sold is entered into the SaleItem table. This value has to be subtracted from the QuantityOnHand in the Merchandise table.

SaleItem Table	Event Code	Merchandise
<u>SaleID</u> <u>ItemID</u> Quantity SalePrice	1. Item is sold by adding row to SaleItem. 2. Quantity is subtracted from QuantityOnHand.	<u>ItemID</u> Description QuantityOnHand ListPrice Category

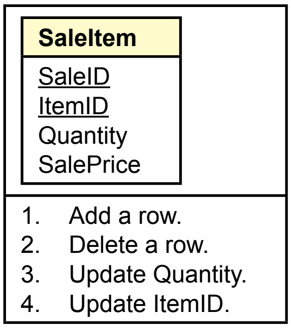


Figure 7.33

SaleItem events. Driven by business operations, four major events can arise in the SaleItem table. The QuantityOnHand must be altered in the Merchandise table for each of these events.

Merchandise Inventory at Sally's Pet Store

What issues arise when maintaining totals in the database? ? To understand the value of procedural code, it helps to look at an example. Handling inventory updates is often a tricky procedure in business database applications. In many situations, employees need to know the quantity on hand for a particular item. An employee may be looking at items to reorder, or a manager might want to know which items are overstocked and have not been selling fast enough. Two basic methods exist to determine the quantity on hand in a database system. First, you could write a procedure that computes the current total on hand whenever it is needed. The routine would add every purchase and subtract every sale of the item to reach the current inventory level. In a large application, this process might be slow. The second approach is to keep a running total of the quantity on hand in the inventory table. This value must then be updated whenever an item is purchased or sold. This second process provides the total very quickly, but faces the drawback of some slightly complicated programming. Keep in mind that both methods also need an adjustment mechanism for "inventory shrink," to use the accountant's euphemistic term for inventory items that have disappeared.

Looking at the Merchandise table from Sally's Pet Store, shown in Figure 7.32, you will notice that it contains a column for QuantityOnHand, so the plan is to use the second inventory approach and keep an updated total for each item. Ultimately, you will need three sets of procedures: One to handle item purchases, one for item sales, and one to adjust for inventory shrinkage identified from physically counting the stock. The adjustment procedure is straightforward, but you have to work on the user interface to make it easy to use. The purchase and sale processes are similar to each other, so the discussion here will examine only the sale of an item.

Whenever something changes in the SaleItem table, the total in the Merchandise table has to be adjusted. Figure 7.33 shows the four basic changes that can arise in the SaleItem table. For instance, when an item is sold, a new row is added to the SaleItem table keyed by the SaleID and ItemID. The row includes the quantity of the item being purchased, such as 10 cans of dog food. This quantity is used to adjust the QuantityOnHand in the Merchandise table. These events might

```
CREATE TRIGGER NewSaleItem
AFTER INSERT ON SaleItem
REFERENCING NEW ROW AS newrow
FOR EACH ROW
        UPDATE Merchandise
        SET QuantityOnHand = QuantityOnHand – newrow.Quantity
        WHERE ItemID = newrow.ItemID;
```

Figure 7.34

New Sale trigger. Inserting a new row triggers the event to subtract the newly entered quantity sold from the quantity on hand.

not be immediately obvious, so consider the following business actions that drive them.

1. A new sale results in adding a row to the SaleItem table, so QuantityOnHand must be decreased by the quantity sold.
2. A clerical error or a customer changing his or her mind could result in the cancellation of a sale or of an item, so a row is removed from the SaleItem table. Any quantity that was already subtracted from the QuantityOnHand must be restored to the total.
3. An item could be returned, or the clerk might change the Quantity because of an error. The quantity adjustment must be applied to the QuantityOnHand total.
4. An item might have been entered incorrectly, so the clerk changes the ItemID. The QuantityOnHand for the original ItemID has to be restored, and the QuantityOnHand for the new ItemID has to be reduced.

You can use database triggers to make the process easier by writing code for each specific event. If you are working with a DBMS without database triggers, the corresponding code has to be written into the forms; this process is similar, but you need to validate each form to make sure it has the necessary code.

The first situation of adding a new row is straightforward. Figure 7.34 shows the logic needed for the database trigger. Only one UPDATE statement is needed: subtract the newly entered Quantity from the QuantityOnHand in the Merchandise table. If you are responsible for reviewing or fixing code in an existing application, you should find that this event is usually handled correctly. The problem is that many developers forget about the other events.

Figure 7.35

Delete Row trigger. This trigger reverses the original subtraction by adding the Quantity back in.

```
CREATE TRIGGER DeleteSaleItem
AFTER DELETE ON SaleItem
REFERENCING OLD ROW AS oldrow
FOR EACH ROW
        UPDATE Merchandise
        SET QuantityOnHand = QuantityOnHand + oldrow.Quantity
        WHERE ItemID = oldrow.ItemID;
```

SaleItem	Clerk	Event Code	Merchandise
SaleID 101 ItemID 15 Quantity 10 Quantity 8	1. Enter new sale item, enter Quantity of 10. 3. Change Quantity to 8.	 2. Subtract Quantity 10 from QOH. 4. Subtract Quantity 8 from QOH.	ItemID 15 QOH 50 QOH 40 QOH 32
Solution that Corrects for Change			
SaleID 101 ItemID 15 Quantity 10 Quantity 8	1. Enter new sale item, enter Quantity of 10. 3. Change Quantity to 8.	 2. Subtract Quantity 10 from QOH. 4. Add original Quantity 10 back and subtract Quantity 8 from QOH.	ItemID 15 QOH 50 QOH 40 QOH 42

Figure 7.36

Errors arise if you do not handle changes in quantity. If Quantity is changed, you must add back the old value and then subtract the new value. The top steps show the error in QOH if you do not handle changes.

The second event of handling deleted rows is no more difficult than the code for inserting a row. Figure 7.35 shows the new trigger that is needed. Deleting a row from SaleItem indicates that the item was not really sold. Consequently, the trigger reverses the effect of the sale by adding the Quantity back to the QuantityOnHand.

As shown in Figure 7.36, the situation for changing data is more complex. You need to think about what it means when the Quantity value is changed. Say that the QuantityOnHand for the specified item begins at 50 units. Then, a SaleItem row was inserted with a Quantity of 10. The insert trigger fired and subtracted those 10 units, leaving the QuantityOnHand at 40 units. The clerk now changes the Quantity from 10 to 8. Since 2 fewer units were sold, the QuantityOnHand needs to be adjusted.

Figure 7.37

Update Quantity trigger. If Quantity is changed, you must add back the old value and then subtract the new value.

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING   OLD ROW AS oldrow
          NEW ROW AS newrow
FOR EACH ROW
          UPDATE Merchandise
          SET QuantityOnHand = QuantityOnHand
                    + oldrow.Quantity – newrow.Quantity
          WHERE ItemID = oldrow.ItemID;
```

```
CREATE TRIGGER UpdateSaleItem
AFTER UPDATE ON SaleItem
REFERENCING   OLD ROW AS oldrow
              NEW ROW AS newrow
FOR EACH ROW
BEGIN
    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand + oldRow.Quantity
    WHERE ItemID = oldrow.ItemID;

    UPDATE Merchandise
    SET QuantityOnHand = QuantityOnHand – newRow.Quantity
    WHERE ItemID = newrow.ItemID;
    COMMIT;
END
```

Figure 7.38

Final update trigger. If the ItemID is changed, you must restore the total for the original item and subtract the new quantity from the new ItemID.

As shown in Figure 7.37, the easiest way to understand the adjustment code is to think of it as adding the original 10 units back and then subtracting the new Quantity of 8 units. The net result will leave QuantityOnHand at 42 units. Notice that you need access to the old row value (10). All trigger-based systems have a way to obtain this value. If you have to build the inventory code on a form, it is slightly more complicated to obtain this value; but it can be done.

The fourth change to the code is more difficult to portray. What happens if a clerk changes the ItemID value? Ultimately, you have to restore the QuantityOnHand for the original ItemID, then subtract it for the new ItemID. The first complication is that database triggers might not have separate events for each column being changed. So you have to integrate the changes due to the ItemID into the previous code written to handle Quantity changes. Again, you need to think about the individual steps. Start with a QuantityOnHand of 50 for ItemID 1, then enter a sale of 10 items. The Insert trigger reduces QuantityOnHand to 40 units. Now the clerk changes the ItemID from 1 to 11. That means that no units of ItemID 1 were actually sold, so the 10 units have to be added back to its QuantityOnHand. Additionally, the 10 units have to be subtracted from the QuantityOnHand for ItemID 11. As shown in Figure 7.38, this trigger requires two separate UPDATE statements. Notice that the WHERE clause in the first statement uses the oldrow.ItemID and the second one uses the newrow.ItemID. Also, look more closely at the two SET statements. The first one adds the oldRow.Quantity, the second one subtracts the newRow.Quantity. Why is this difference important? First, it is possible that the clerk changed the Quantity along with the ItemID, and you need to make sure the old Quantity is used for the old ItemID. Second, and more importantly, this trigger also handles the simple change in Quantity, even if the ItemID is not changed. Assume the ItemID is set at 1 and is not changed. Start with a QuantityOnHand of 50 units, and an initial Quantity sold of 10, leaving a current QuantityOnHand of 40 units. Read through the code to see how it works if only the Quantity is changed from 10 to 8 units. First, the old Quantity (10) is added back to the QuantityOnHand. Second, the new Quantity (8) is subtracted, leaving 42 units on hand. This process is the same as that shown in Figure 7.37, but it is accomplished in two steps instead of one.

The same code must be written for the purchase table (OrderItem) with the same logic. However, for business reasons, you might want to wait to update the QuantityOnHand until the items actually arrive. If you do decide to wait, your primary initial trigger is not on the OrderItem INSERT event, but on the UPDATE event on the MerchandiseOrder table. Have the trigger look for an entry in the ReceiveDate column, and then do the QuantityOnHand updates.

Summary

Although SQL commands are powerful, you sometimes need a procedural language to gain detailed control over updates or to connect to other devices or applications. Depending on the DBMS, procedural code can exist within modules, within forms, or in external applications. Database triggers are an important application of procedural code. These procedures are triggered or exectued in response to some database event, such as inserting, updating, or deleting data. Triggers can be used to enforce complex conditions or to execute business rules. For instance, a trigger might be attached to QuantityOnHand within an Inventory table to automatically notify a supplier when the value falls below a certain level. Cascading triggers arise when a change in one table fires a trigger that causes changes in additional tables, that might trigger even more events. Long cascades can be difficult to debug and use substantial server resources.

Transactions are critical applications in most business operations. They represent a collection of changes that must succeed or fail together. Setting start and ending points for transactions is an important step in application development to protect the integrity of the data. Concurrent access where multiple users attempt to modify the same data at the same time is another substantial threat to database integrity. Pessimistic locks have often been used to protect data through serialization so that only one transaction can see data at a time. However, multiple locks eat up resources and can lead to deadlock issues. Optimistic locks assume that collisions are unlikely, but code must be added to handle the situations when they do arise. The ACID acronym (atomicity, consistency, isolation, and durability) is a useful way to remember the main features desired of a DBMS to protect transaction integrity.

Generating keys is an important step in many relational databases, since it is difficult to trust humans to create unique identifiers. Two common methods are used to generate keys: (1) automatically create them when a row is added to a table, or (2) provide a separate function that generates keys on demand. Both methods create complications. The automatically generated keys are difficult to obtain and use in secondary tables. The generation functions require programmers to write code for every table and every insertion procedure.

Database cursors provide a method for procedural code to retrieve multiple rows of data from a query to step through the rows one at a time. The cursor points to one current row that can be examined, modified, or deleted by your code. Scrollable cursors move forward or backward through the rows, but whenever possible, you should try to move only in one direction. With updateable cursors, code can change or delete the data in the current row. With a parameterized query, code can dynamically choose the rows to be retrieved in response to other conditions.

A Developer's View

Miranda learned that even a good DBMS often requires programming to handle some complex issues. In developing your application, you should examine all of the business processes and identify transaction elements. Also, be sure that your UPDATE and DELETE procedures can handle concurrency issues. Remember that a professional application anticipates and handles errors gracefully. Write data triggers or module code to automate basic processes and perform all needed calculations. Write additional cursor-based code if needed to perform advanced calculations.

Key Terms

atomicity
cascading triggers
concurrent access
consistency
database cursor
deadlock
durability
isolation
isolation level
optimistic lock
persistent stored module (PSM)
pessimistic lock
procedural language
scope
serialization
syntax
transaction
trigger

Review Questions

1. Why would you need a procedural language when SQL is available?
2. What is the purpose of data triggers?
3. What is the purpose of form events?
4. What is a transaction and why do they have to be defined by developers?
5. How do you start and finish a transaction?
6. How is pessimistic locking different from optimistic locks?
7. What code do you need to add to handle conflicts with optimistic locks?
8. What is an ACID transaction?
9. What are the most common methods used to generate keys?
10. How do you obtain the most recently generated key in the DBMS you are using?
11. What is a database cursor and why is it important?
12. What is the program logic to using a database cursor to alter data?

Exercises

1. Create a small database with tables for Customers and Employees. In addition to name and phone number, each table should hold a date column for when the person first started (as either a customer or hire date). Write a function that returns a percentage discount that uses a phone number to decide if the buyer is a customer or employee. Customers for less than one year get no discount, 1-3 years (2%), 4-7 years (4%), 8 or more years (5%). Employees for less than one year get no discount, 1-2 years (5%), 3-5 years (7%), 6 or more years (10%).

2. Create a database table of Employees that includes the maximum number of vacation days and number of sick days allowed each year.

```
Employees(EmployeeID, LastName, FirstName, Phone,
VacationDays, SickDays, DateHired, Dateborn)
```

Create a second table with keys for EmployeeID and Year that has values for number of vacation days and sick days taken that year.

```
    EmployeeDays(EmployeeID, EYear, NVacation, NSick)
```

   Write a function that has input parameters for Year, EmployeeID, number of days off, and whether they should be recorded as sick or vacation days. If the employee exceeds the number of allotted sick days, assign the days as vacation time instead. Excess vacation days do not get counted as sick days.

3. Using the same two tables as the prior exercise (Employees and EmployeeDays), write a database trigger that prevents anyone from entering a value for vacation days taken that exceeds the maximum allowed.

4. Create a table that lists item category and the level of tax on that category. For example, food (0 percent), clothing (3 percent), entertainment (10 percent). Write a function with category and price as parameters. Compute and return the appropriate tax. Normally, you would use an SQL statement for this computation, but if the tax table is provided on a separate system, you might need to write code.

5. Create a data trigger that writes a row in a new table whenever employee salary is changed. Store the date changed, the employee, the old salary and the new value.

6. Create a data trigger that will prevent anyone from increasing an employee salary by more than 75 percent.

7. Create a data trigger (or form code if triggers are not available) that adjusts inventory quantity on hand whenever an item is sold. You need a SaleItem and Item table.

8. A Web site sells custom components for cell phones. The site often offers daily deals which consist of "packages" of related items for a specific phone. Table: PackageItems(PackageID, ItemID, SalePrice) For example, one deal might contain a case, screen protector, and color-matched earphones. Each item is listed separately in the Items table of the database which includes

the Quantity On Hand value. Table: Items(ItemID, Category, Description, ListPrice, QOH) Write a transaction function to safely handle the sale of one package that updates all of the QOH values as part of a transaction.

9. Using the basic Items table that contains a QOH column, create a form that lets users edit the data directly. (Normally, you would use a Sale form but keep it simple for now.) Using default settings, determine what happens if two people change the same data at the same time. Adjust the settings to check for optimistic and pessimistic locking if they are available. Hint: You might want to create two separate forms connected to the same table for testing purposes.

10. Assume you are building a database for a Web-based form where a manager loads and displays all of the employee data for editing. At the end of the session, the changes made to the data are sent back to the database. Write the SQL command to safely update the table using optimistic concurrency. Assume you have an array that holds (a) the original values read from the database and (b) the new/changed values.

11. Create a table for LoanPayments(LoanID, PaymentNo, DateDue, Amount). Write a function that is called whenever a new loan is created, to load the payments table with the scheduled payments and amount due.

12. Given the following table, write a cursor-based procedure to loop through the table and compute the percent change from the prior month and store that value in the current row.

SalesMonth	Sales	PercentChange
01	25,123	
02	24,331	
03	32,992	
04	37,102	
05	42,474	
06	46,551	

13. Using the table in the previous exercise, write a cursor-based procedure to compute the average monthly sales (without using the SQL AVG statement).

Sally's Pet Store

14. Where would you put the code (which Event) in each of the following situations? Note if you are using Access or SQL. You do not have to create the code for this exercise.

    A. Notify a purchasing manager whenever inventory drops below a specified amount.
    B. Compute the Sales Tax owed on a Sale.
    C. Notify a supplier when an order is received.
    D. Notify adoption groups of the total amount of donations they received for the day.
    E. Validate a new employee's Taxpayer ID with an online company.

15. Write a function to compute the average purchase cost of an item over the prior year and provide a warning if the ListPrice of the merchandise is lower than that value.
16. Write a function to insert a new Customer and return the generated key value. Inputs to the function include the LastName, FirstName, and Phone number.
17. Create a table to hold totals of merchandise sales by month and a percentage increase in sales from the prior month. Write a (SQL) function to compute the monthly totals and transfer them into the table. Add code to compute the percentage changes.
18. Write the code to increase quantity on hand when an item is purchased—specifically when the receive date is set. Be sure to handle it as a transaction, since quantity on hand can also be affected by sales.
19. The Pet Store is thinking about purchasing scanners to use at checkout. These scanners will pick up the ItemID of each merchandise item scanned. Assume that this data will trigger an event when an item is scanned. Write a function that can be called by this event. This function should create a new sale, and store the data for the items sold. You can emulate the scanner trigger by creating a form with a control to select an ItemID and a button to fire the trigger.

Rolling Thunder Bicycles

20. Create a function to compute the great circle route (shortest) distance between two geographic locations.
21. Where would you put the code (which Event) in each of the following situations? Specify if you are using Access or SQL. You do not have to create the code for this exercise.
    A. Send an e-mail message to a customer when a bicycle is shipped.
    B. Send an e-mail message to a supplier to order more components when quantity on hand drops below a preset level.
    C. Notify a manager when an employee is involved with purchases of more than $50,000 in a month.
    D. Notify (e-mail) a manager if the daily sales value of bicycles exceeds a preset level (both high and low) in terms of percentage change from the prior year.
    E. Notify a purchasing manager of all items that were ordered within the last month but not yet received.
22. Create a table to log changes to Employee salaries (SalaryChange(ChangeID, ChangeDate, EmployeeID, OldSalary, NewSalary, User). Write trigger code on the Employee table to record any changes to the salary into the log table.
23. Create a function that estimates the time to build a new bicycle. It should use the average number of days for the same model type but adjust the days by the number of orders of all bikes made in the past 14 days.

24. Create a form or a function that lets the finance manager safely record payments to manufacturers.
25. Write a function to update the BalanceDue column in the Customer table while avoiding concurrency issues. The function needs input parameters for CustomerID and ChangeAmount which can be positive or negative.
26. Create a query to compute sales by month for each model type. Create a temporary table to hold that data and to hold the percentage change. Write a program that executes the query, placing the data into the table. Then cursor-based code computes the percentage change in sales. The function should return the new balance value.
27. Write a procedure to add an interest charge to customer accounts with a balance due. Make sure to handle concurrency/locking problems.
28. Write a program to automatically generate a new purchase order when quantity on hand falls below a specified level. Add the ReorderPoint column to the Component table and enter sample data.

Corner Med

29. Where would you put the code (which Event) in each of the following situations? Specify if you are using Access or SQL. You do not have to create the code for this exercise.
    A. Two physicians sign up for vacation on the same days.
    B. E-mail notices sent to the director physician whenever a patient is diagnosed with a set list of codes/diseases (particularly some contagious diseases).
    C. A warning message sent to the physician and business manager whenever the AmountCharged for a Visit Procedure is below 50 percent of the base cost.
    D. An e-mail sent to the business manager whenever the amount paid by the insurance company plus the amount paid by the patient differs from the total amount charged for a visit.
    E. A warning notice sent to the physician when the Systolic pressure for a visit is greater than 140 and the patient is prescribed a drug from a certain list.
30. Write a function that reduces the amount charged for a procedure for a specific patient (VisitProcedureID) and reduces the patient amount owed/ paid.
31. Create a table to hold revenue earned per week, using a date format of yyyy-ww. Include a column to hold percentage change from the prior week. Write a query to compute the totals and a routine to compute and store the percentage change.
32. To facilitate loading data from the company’s older system, write a function that creates a new patient record given LastName, FirstName, Gender, DateOfBirth as input parameters, and creates a new visit record for that patient for a VisitDate parameter. The function should return the newly generated VisitID.

33. Write a database trigger to record the date, user, and patient name any time a patient row is deleted.

34. Change the tables so that patients can make multiple payments. Include the date, amount of payment, and visit. Write a function to return the total amount paid by a patient for a given VisitID. Briefly explain why this method is better than the current tables.

Web Site References

http://www.sigplan.org/	Association for Computing Machinery—Special Interest Group on Programming Languages (advanced).
http://support.microsoft.com/kb/115986 http://speckyboy.com/2012/05/13/six-common-web-programming-mistakes-and-how-to-avoid-them/	Avoiding common database programming mistakes.

Additional Reading

Baralis, E. and J.Widom, An Algebraic Approach to Static Analysis of Active Database Rules, *ACM Transactions on Database Systems (TODS),* 25(3) September 2000, 269-332. [Issues in database triggers and sequencing, but plenty of algebra.]

Ben-Gan, I., L. Kollar, and D. Sarka, *Inside Microsoft SQL Server 2005: T-SQL Querying*, Microsoft Press: 2006. [Discussion and examples of advanced topics for SQL Server.]

Gray, Jim and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, San Francisco: Morgan Kaufmann Publishers, 1993. [A classic reference on all aspects of transaction processing.]

ISO/IEC 14834:1996, *Information Technology—Distributed Transaction Processing—The XA Specification, 1996*. [A discussion of the common method of handling transactions across multiple systems.]

Sanders, R. and J. Perna, *DB2 Universal Database SQL Developer's Guide*, Burr Ridge, IL: McGraw-Hill, 1999. [Using embedded SQL with IBM's DB2 database.]

Urman, S., R. Hardman, and M. McLaughlin, *Oracle Database 10g PL/SQL Programming, Oracle Press*: 2005. [One of many references providing an introduction to SQL Server programming.]

Vossen, G., G. Weikum and J. Gray, *Fundamentals of Transactional Information Systems : Theory, Algorithms, and Practice of Concurrency Control and Recovery*, San Mateo, CA: Morgan Kaufmann, 2001. [Detailed programmer's perspective of transaction details.]

Chapter 8

Application Development

Chapter Outline

What You Will Learn in This Chapter

- What features need to be included in finished applications?
- How do you create a consistent application design?
- How are forms and reports integrated and organized?
- How can users gain easy access to standard operations across the application?
- How can a computer application be modified for people with disabilities?
- How do you create custom help files?
- What does your application do when something goes wrong?
- How do you know your application works correctly?
- How will your application be installed?

A Developer's View

Miranda: Finally. I think I see the end of this project.

Ariel: That's terrific. What's left?

Miranda: Well, everyone is happy with the forms and reports. All I have to do now is tie them together into an application. I have a few details to add to make the forms a little easier to use. The salespeople complained about having to enter customer numbers twice, and they say the order lists are too long. They want to pick from a list of orders just for the given customer.

Ariel: That's it? Let's celebrate!

Miranda: Well, not quite yet. I also have to write some help files. Then I have to create a set of installation disks so they can install the system on all the computers.

Ariel: Sounds like a lot of details. Will it take long?

Miranda: I don't think so. But it will make the application more attractive and easier to use, so I really need to finish the details.

Getting Started

Applications are more than just forms and reports. You need to ensure all forms and reports have the same look and feel. You build an integrated application by linking everything with menus. You also need to add help files, add error handling, and test the entire application. Part of the testing includes ensuring that the application is accessible to all potential users. You also have to build administrative tools and deploy the application.

Introduction

What features need to be included in finished applications? As a database developer, it is your responsibility to create systems that help users in their jobs. You accomplish this task by building an **application** to perform a specific task. The task is defined by the user, and your application needs to be easy to use. The goal of the application is to collect data and provide information to help users make decisions. You define tables to hold the data, but you never want users to see the underlying tables. Instead, you create forms to collect data and reports to help users visualize and analyze the data.

An application is more than a collection of forms and reports. It is a set of forms and reports that work together. More specifically, an application has (1) an internal consistency to the user interface, (2) a structure or layout that supports the flow of user tasks, (3) menus to make it easy to find things, (4) a help system to provide documentation and assistance, (5) error handling to catch anticipated problems and protect the user, and (6) a method to deploy the application. At this stage in the design, you also have to perform considerable testing and evaluation of the application.

The first two items (consistency and structure) are the defining elements of an application. The others are features that are added at this stage to make the system more reliable and easier to use. The look and feel of the forms is the major

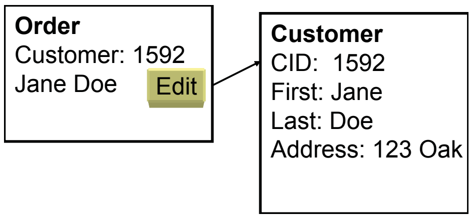


Figure 8.1

Application structure. Forms are connected so users can click a button or link to get more detail, open other forms, or display reports.

element of consistency and design. Every form in an application should have the same look and same approach to entering data. Otherwise users will get confused and become frustrated because your application makes it harder to perform their jobs. The topic of structure entails connecting forms and reports together. Users entering data on one form should be able to click a button, or double-click an entry to see more detail. Figure 8.1 shows a simple example. Users entering data on the Order form will probably want to open the Customer form to edit the data about the customer. As a developer, you have to learn what connections are needed on each form. These connections help create the structure of the application.

Two-Minute Chapter

Applications begin with forms and reports but require consistency, structure, and supporting details including menus, custom help topics, and the ability to handle errors. Unless you have been exceedingly careful from the start, be prepared to spend time to rebuild all forms and reports to ensure they have the same "look and feel."

Applications need to have consistent colors, fonts, layouts, menus, and usability features. When possible, build templates that serve as the foundation for forms and reports. These templates usually set the page structure as well as the styles to be used in each component. Applications might need different templates and styles for different devices—particularly if mobile phones are going to be used with smaller screens. To improve accessibility, try to use system fonts and colors to support user control. Whenever possible, support multiple input methods, including keyboards.

The structure of an application is important—particularly where components are stored and run, as well as how pieces connect together. Most applications require some type of menu or toolbar along with buttons and links to help connect forms and reports and make it easier to find various elements and explore the application capabilities.

Applications also need custom help pages to answer user questions—particularly by providing the ability to search for keywords. Help topics are usually written as individual HTML topic pages. Keywords can be added to those pages along with using heading levels to indicate the table of contents or structure. Windows help files are created by compiling the HTML files into a single CHM file using the HTML help compiler from Microsoft or (expensive) third-party compilers. Context-sensitive help is provided by assigning a number to every page using the topics.h file, then entering the help file name and topic number into the application properties; such as the form properties in Microsoft Access.

Applications also need extensive testing, including module, integration, stress/performance, usability, and security testing. Special attention should be paid to check for SQL Injection attacks. At a minimum, all user inputs should be cleaned and SQL queries should use parameters instead of string concatenation. Even with extensive testing, errors can still arise, so the application should have error handling code that can automatically recover from errors whenever possible. It should also have the ability to log errors so developers can examine the log to look for common issues and find improvements for the next versions.

Applications also need a deployment method. Server-based systems are straightforward and might be deployed using basic script files. Client-based applications should be packaged and installable from a simple start command.

Design Consistency

How do you create a consistent application design? This question is particularly important to answer when several developers are working on the same project. The application design consists of several levels. Some are easier to configure and observe. At the most basic level, all forms should use the same color scheme. Likewise, reports should use the same fonts and follow a similar layout. But, design also includes usability issues such as the page layout and selecting items from a list instead of memorizing ID numbers. Consistency in design also applies to the links between forms and the use of menus and toolbars. The primary issues discussed in this section are related to design and usability.

Page Design Templates

Look through a book, magazine, or well-designed application or Web site and you will see that all of the pages have a consistent appearance. Only a few fonts are used, headings are aligned the same, margins match, and colors blend and match across all of the pages. This consistency does not happen by accident. Graphics designers first create a template and then apply the template to all of the pages. A **template** defines the style features of a page. The capabilities depend on the system and tools available, but you can usually define the overall page layout, fonts, and colors.

Figure 8.2 illustrates the effect of a form template. Take a look through the forms you have created for your assignments and projects to this point. Do they look like the first form? When you look at a collection, do they all look the same, or are they all different? Eventually, developers learn to pay attention to detail and consistency. Even so, it is easy to make mistakes. A template makes it easier for everyone to be consistent and to reduce errors. Ultimately, someone still has to examine every form and report to double-check consistency, but with templates, you can come very close on the first pass.

Each development tool has different methods to create and apply templates. A few systems do not support templates at all, and some have predefined templates that you cannot change. The most common approach to templates is to apply the template at design time. If the template is changed later, there is no way to push the changes onto all of the forms based on the template. In other words: Be sure your template is complete and accurate before you create the forms and reports. If you have already created complex forms, it is difficult to apply a template. In some cases, you can create a new blank form based on the template and copy the controls from the original form. In extreme cases, you might have to rebuild the entire form from scratch if you need to apply a template later.

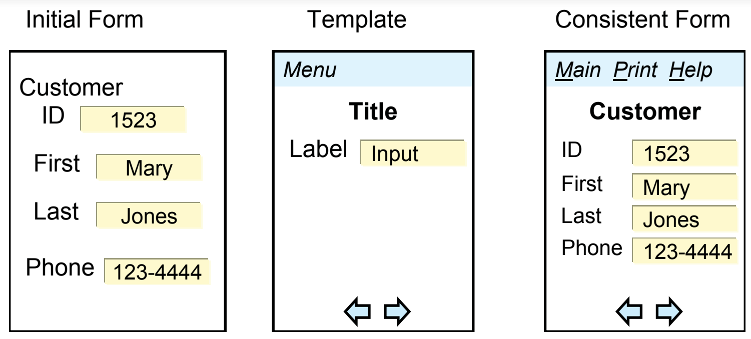


Figure 8.2

Template. The template defines the structure of the form and the attributes of the various elements. It can also include common features such as menus and navigation buttons.

Application design is something that needs to be established early in the development process. Even if your system does not support templates, you can define a **style sheet** that defines the overall page layout, the common elements to include, and the styles of each major element (titles, labels, text, and so on). Each developer is responsible for following the guidelines on the style sheet. This process is harder to use than a template, but it works for every system.

Templates are not the final answer to design questions. For instance, some forms or reports may need to add other features, or change margins to make something fit on a page. Fortunately, once the template is applied, you can override a setting and coerce the page to get the layout you want. However, you need to be careful with this power. As a developer, you have to make decisions. It is best to maintain consistency, but ultimately, you have to keep the users happy. If a manager insists on squeezing an extra column onto a report, you will have to reduce the margins or change the font size. On large projects, you should establish a coordinator who can be consulted when you need to override a template specification.

Usability

Beyond layout, fonts, and colors, you need to establish a consistent set of usability standards for an application. For instance, consider the issue of foreign keys such as using CustomerID in the SalesOrder table. When a clerk is entering data into the SalesOrder table, it would be painful to require the clerk to memorize the CustomerID values. Instead, the form will have some type of drop-down list or list-of-values option that enables the clerk to pick the appropriate customer from a list. In terms of your application, you need to be sure that all foreign key references use the same approach to solve the problem. Consider the situation shown in Figure 8.3, where the Sales form uses a drop-down list to pick customers based on phone numbers. Imagine how annoying it would be to use a pop-up box to search for customers by name on the Receipts form.

When the same users are going to work with multiple forms, the forms need to use a consistent data-entry method. In general it is better to be consistent across all of the forms. In a few cases, one group of users might insist on a unique lookup approach that more closely matches its needs, but these variations should be discussed and approved separately.

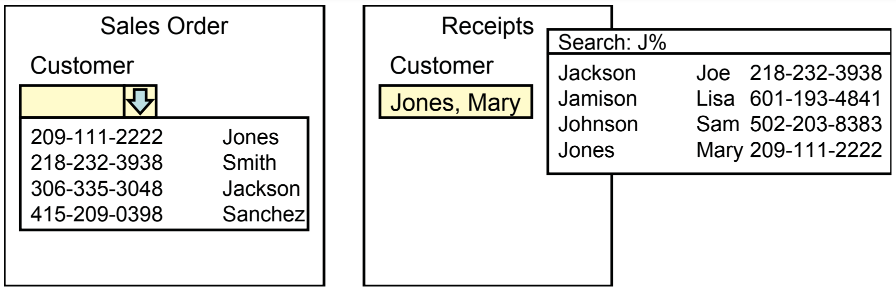


Figure 8.3

Consistent usability. Both forms require the selection of a customer since you cannot expect people to memorize ID numbers. It would be annoying to select customer by phone number in a drop down list on the Sales form but by a pop-up list organized by name on the Receipts form.

The idea of consistency also applies to the tab order, choice of related data to display on a form, subform layout, and similar topics. For example, when the user selects a customer, you might choose to display additional information, such as phone number, on the form. As much as possible, this same data should be displayed on all forms involving customer lookups.

Fonts and Customization

Selecting fonts and color schemes for an application is always a challenge. You face conflicting goals. On the one hand, you want to choose a pleasing design that displays all of the relevant information in one place. On the other hand, you need to give users control over the displays. Why do users need control? Users often configure their systems to support the way they work or to compensate for vision issues.

Users can configure the Windows environment through various settings in the Control Panel—including font sizes, color schemes, and regional settings such as date displays. Your application needs to support these settings. You provide this support by choosing system-defined fonts and colors. Windows development tools, such as Visual Studio, provide font settings for choosing system fonts. Use these choices instead of picking a fixed typeface and font size. Likewise, you should choose the Windows palette colors instead of forcing a fixed color. When your application runs, it will pick up the currently-defined fonts and colors. If the user changes those values, your application will adapt and use those colors. Yes, in some cases, the user might pick strange color combinations, but the decision belongs to the user—not to you.

Web applications are somewhat trickier—and the capabilities are heavily dependent on the development tools you use. At the moment, most systems do not provide user control over color schemes. Some default schemes are commonly used (e.g., white background, black text, and blue highlights), and you should follow these schemes when possible. On the other hand, font sizes are more flexible, and more interesting. Take a look at your browser options and you will find an option to control the font size (try Page/Text Size). However, if your Web form specifies fonts in terms of points, this option will not work for the users. Also, if you specify page layout sizes (e.g., tables) in terms of fixed measures such as pix-

Device	Size (Diagonal in.)	Pixels
Desktop	24	1920 x 1200
Laptop	15	1440 x 800 1920 x 1080
Apple iPad	11	2048 x 1536
Google Nexus 10	10	2560 x 1600
Apple iPhone 3S	3.5	480 x 320
Apple iPhone 5	4	1136 x 640
Samsung S4	5	1920 x 1080

Figure 8.4

Sample device resolution. Fewer pixels means less information can be displayed. Smaller size means text and images might be too small to read, or less information can be displayed at readable fonts sizes.

els, changes in text size will not work very well as the font changes but the container remains the same size. Consequently, you have to use font and size settings based on relative terms instead of absolute point values. A relatively new way to define sizes is to use ems, where one em is defined as the width of the letter M in the current font. With relative sizes, the font and layout values will be rescaled automatically when the user changes the font size.

Mobile Devices

Increasingly, users want to access data from anywhere—using mobile devices connected to internal applications or Web sites. Providing users with greater access to applications and data is good. However, cell phones and other portable devices generally have smaller screens. Many wireless plans also have limitations on the data transfer speed or monthly caps (or high prices) on the amount of data transfer. Figure 8.4 shows approximate sizes and pixel counts for a handful of devices. Keep in mind that technology continues to evolve, so you need to research current values when you build an application. The main key is that the cell phones (Apple and Samsung) are considerably smaller and use fewer pixels. Although starting with the Apple iPhone 4 released in 2010, resolutions have improved—no one will be able to read the text if it is drawn pixel-for-pixel. Instead, the newer phones and tablets use those pixels to display better-formed characters—shown at readable size but with better resolution and clarity. Compare an iPhone screen at 3 x 2 inches with a desktop monitor at about 13 x 8 inches. Physically, the desktop screen can hold 16 iPhone screens. The pixel count works to about the same ratio with the 3S, but not the version 5. The point is that even if your application uses only half of the desktop monitor, only a small fraction of that page could be displayed on a mobile screen at one time.

Pages that work on larger screens can become unusable at smaller sizes—either the fonts are too small to read, or the user has to scroll vertically and horizontally to see the entire form. Unfortunately, there is no good answer to creating pages for different sizes of screens. Often, it is necessary to create two versions of the application—one for “regular” screens and one for smaller mobile screens. Creating different versions is also useful because mobile devices often have slower (or

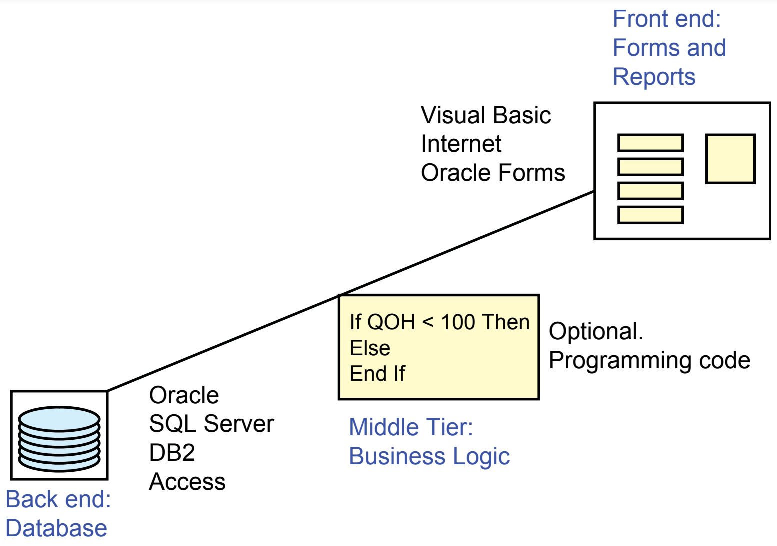


Figure 8.5

Application structure. It is common to use different tools for the back-end database and front-end forms. The use of the middle tier to handle business rules depends on the size of the application and the management details of the firm.

more expensive) network connections, so you have to be more cautious in using graphics and limiting the amount of data sent to the page. But developing two versions of an application takes more time--hopefully less than twice as much—because you already have the tables, functions, and calculations. And you have to plan for more people to handle maintenance and upgrades.

Application Structure

How are forms and reports integrated and organized? The overall structure is an important feature of any application. The structure or layout defines how the user will deal with the application. Most database applications will use forms and reports as individual components. The first step in designing the structure or architecture of the application is to design each form. The objective of application structure is to organize all of the forms and reports to produce a complete application. In some applications, this purpose can be achieved with a central startup form, which contains buttons to direct users to the appropriate form. More commonly, you will also need to add interconnection buttons on individual forms. For example, a user entering data on an order form might want to look up additional information on the customer form.

Today, it is common to separate a database application into two or three major sections. As shown in Figure 8.5, the **front end** consists of the forms and reports that the user sees. The **back end** consists of the database tables. Sometimes a **middle tier** is added that consists of program code to define and enforce business rules. With a network, this separation can be physical, and each component can run on separate computers. Even if all of the elements will run on a single machine, it often makes sense to separate them logically. This separation enables you to change each part independently. It also makes it easier to choose different tools for each purpose. For example, you could write the front end forms and reports with Visual Studio, store the data in an Oracle database, and write custom code to evaluate business rules. If the business changes, you could transfer the business rules to an enterprise resource planning system, or change the back-end DBMS with only minor changes on the front end.

Choosing the overall architecture of the application is the first step to designing the application. The choice of tools and structure will depend on the organization's needs and capabilities. Some companies have a formal process for designing and approving applications. This process is important when applications need to work together and fit into the overall structure of the company's information system.

Designing Applications

The first step in designing the application structure is to identify the various users and outline the tasks that will be performed with the application. The application must reflect the needs and working habits of the user. If several users have different needs, the application can be divided into sections for each group. A central startup form can be used to identify the user and direct him or her to the appropriate section.

This segmentation reduces complexity for the users and simplifies their tasks. However, it has two potential drawbacks. First, if the application has too many layers, users will have trouble finding the forms and reports they need. Second, poor organization confuses users and requires additional support and training. In other words, you must find an application structure that provides the functionality each user needs but is still easy to understand. The inherent conflict in these goals is what makes it so difficult to design a good application structure.

Even experienced programmers rarely design a "perfect" application the first time. In most cases you need to develop several ideas and test them. You can build **prototypes** by creating sample forms and including command buttons to tie them together. These prototypes can be given to users to test. You then incorporate user suggestions and modify the prototypes. By testing different structures, you can quickly learn which technique will work best.

In building a complex application structure, it is best to start with the core concepts. Once you have tested them with users, you can add features. Each revision constitutes a new application version. Keep track of the version number; record the date, the reason for the change, and the changes that were made. Most commercial software vendors follow this development process. No one tries to visualize a complete, massive application and create it up front. Instead, developers start with a basic concept and build a system that works and implements the fundamental concepts. Then developers expand the capabilities by adding new features.

The two most important aspects in this type of development are (1) getting the overall structure correct up front and (2) using a flexible design that is easy to modify later. For example, it is critical that your data tables be normalized, because normalized tables can be easily expanded later to provide new features.

The Startup Form

Designing an overall structure and appearance often requires artistic sensibility as well as logic and research. Each application is different and can require a unique approach. Yet, over time, designers have learned that some common elements can be used in many applications. The main menu or **startup form** is an element that many developers like to use. However, it is not really the ultimate answer to every problem.

- Directory for the application.
- Identify users.
- Startup and shutdown code.
  - Preload forms in background.
  - Initiate transaction and security logs.
  - Establish network connections.
- Copyright and usage notes.

Figure 8.6

Uses of startup forms. As the initial form for an application, the main menu can be used to control tasks that apply to the complete session.

The main purposes of the startup form are shown in Figure 8.6. The startup form is generally the first form of the application. It provides a centralized directory to the rest of the application. It often contains an image or picture and usually consists entirely of command buttons. Clicking a button brings the user to another menu form or to a specific form or report.

Because the startup form is the starting point for the application, it is a good place to identify the user. If possible, you should identify the user from the network login data. Otherwise, you will have to maintain a separate login for each user. The two primary reasons for identifying each user are (1) to maintain appropriate security controls and (2) to customize the application for each user group.

The startup form can be customized through layout and the use of color. For example, options primarily intended for different managers (marketing, finance, etc.) can be displayed in different colors. If additional customization is needed, individual options can be made invisible and disabled so that users see only buttons that are designed for their use. This approach simplifies the screen layout and reduces confusion. However, it is less useful if managers need to share their tasks.

Keep in mind that some applications will work better without a startup form. Think about the applications you use on a daily basis: word processing, spreadsheets, and so on. None of these require startup forms. Instead, they jump right to the primary user task and rely on menus to provide access to the functions. You could use this same approach for users who have a limited view of the application—such as front line clerks. Once a clerk logs in, the application jumps immediately to the primary data-entry form. The point is that you should not try to force a startup form into every application. Look at the jobs and talk with the users to find the best way to organize the forms and reports for each task.

Sally's Pet Store: Application Organization

In many ways the startup form is a table of contents into the application. It presents the organization of the application. Before building the startup form, you must decide how the application will be organized. That is, you must learn which forms are most important to users, how they will switch between forms, and how often they use each form.

Although there are many useful ways to organize any application, consider two different approaches to the Pet Store application. The first approach is shown in Figure 8.7. At first glance this approach seems reasonable. Items are ordered, then received and then sold to customers. Hence the store managers might want to start with orders and enter data by following each item from purchase through sale.

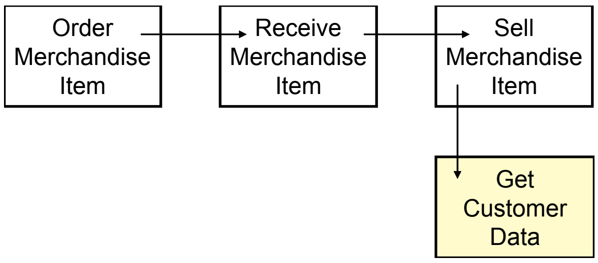


Figure 8.7

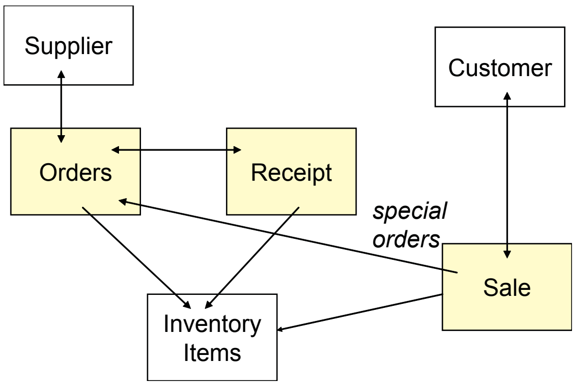
Poor organization of the Pet Store application. The links are at the wrong level (item instead of order). Managers rarely need to track individual items from order to receipt to sale.

Although this approach might sound reasonable at first, it has several flaws. First, managers rarely want to track individual items. Perhaps they want to follow individual animals, but rarely merchandise. Second, when an order is placed, the item has not been received yet, so there is no point in linking an order to the receipt of the shipment. More important, there is no way to connect individual items to a sale. For example, you might know that a customer bought three cans of a particular dog food, but there is no way to tell exactly which cans. Hence, managers rarely need a link from receipt of shipments to individual sales.

An improved approach appears in Figure 8.8. First, notice that it has more links—including bidirectional links. For example, when a shipment arrives, workers need to pull up the matching order to see whether the proper items were delivered. Hence an Orders button is placed on the Shipping Receipt form. Once in a while, a manager might want to check on the shipment of a particular order, so the link is bidirectional. Notice that Sales are connected to Orders and Receipts—but only through the Inventory items. Inventory QOH can be displayed directly on the Sale form. The Sale form also has a connection to Orders—to create special

Figure 8.8

Improved organization for the Pet Store. The lines represent links from one form to a second form. The links are usually created through buttons placed on the form.



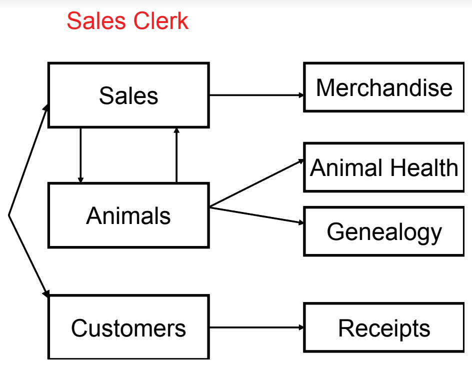


Figure 8.9

Collaboration diagram for sales clerk section. Diagramming the forms and reports used by an actor (employee) makes it easier to identify the overall structure and menu for this role.

orders. If an item is out of stock, a salesperson might want to check on recent orders to see when the item might arrive. The designer should talk with users to determine how often this situation arises and how it should be handled on the form.

Eventually the Pet Store application will contain many forms and reports. Most of them are linked to a startup form. Many of them are linked to each other. Buttons or events on one form lead the user to a related form. Some of the forms are simple and affect one table, but most display data from several related tables. Each individual form represents specific business events and tasks. Figure 8.9 shows the primary forms used by the sales clerk role. This diagram is a simplified version of a UML collaboration diagram. The main point here is that it identifies the initial forms needed by this group of users. Consequently, you should put links to Sales, Animals, and Customers on the startup for this user. You will need to create a similar diagram for the other roles at the Pet Store, such as purchasing and management.

Figure 8.10 shows one version of a startup form for Sally's Pet Store. The buttons on this form match the primary tasks identified for the groups of users. The buttons are color coded to highlight the three groups. You could go further and set the visibility of the buttons based on the category of the user. When each person logs in, the form displays the buttons or forms most relevant to that person's tasks. With some additional coding, you could write the form so each user could select a set of buttons to personalize the main screen. However, customization is usually easier on tool bars instead of the main menu. In general, you should avoid putting too many buttons on the main menu. An old rule of thumb states that the average person can handle seven, plus or minus three, items at a time, so four to ten buttons on a form is a good target. Obviously, a complex application ultimately has many more than 10 forms and reports. With large applications, you can extend the startup form to additional forms or submenus. You can also add drop-down menus to make it easier to find commonly used items.

Remember that you also need to connect forms to other forms. Depending on the user interface you choose, you might add buttons to forms or use double-clicks to trigger the second form to open. These links are commonly created with foreign key relationships, such as adding an Edit button to the SalesOrder form to open

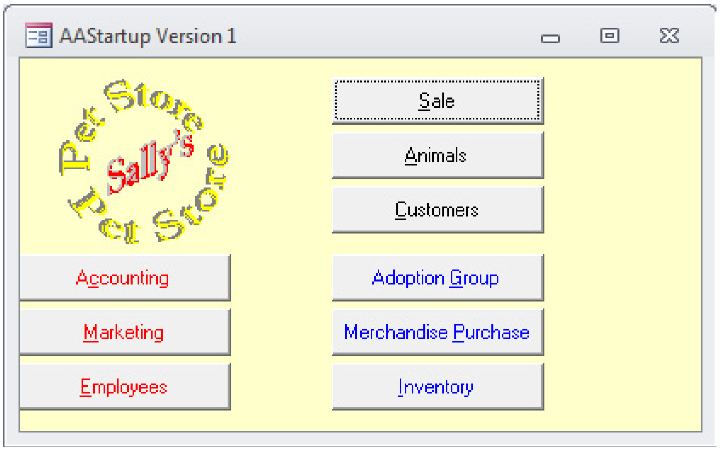


Figure 8.10

Sample startup form. The buttons match the user's tasks.

the Customer form. Similarly, you can add links to print or preview reports from various forms, such as printing a sales receipt from the SalesOrder form. The specific links depend on the needs of the users.

Administrative Tasks

When you build the application, you also need to think about the administrative tasks that will need to be performed. **Administrative tasks** consist of jobs that need to be performed to keep the application running, such as updating data in lookup tables, backing up the database, and assigning users to groups. Depending on the DBMS, some of these tasks are handled outside of your application. If your system needs routine maintenance or tasks performed on a specified schedule, you should incorporate a set of administrative forms to automate the tasks. If nothing else, collecting the tasks into one location makes them easier to handle and increases the probability that they will actually be done. Tasks that require external steps could at least be documented within the application. Ultimately, you can hide the administrative tasks from the common users and use the security system to make them accessible to a few administrators

Administrative forms are particularly important for Web-based applications. You will find that it is convenient to handle administrative tasks using a Web browser so the administrators can support the application from almost any location. This step is particularly critical when the application will be hosted by an external Internet service provider. On the other hand, it sometimes takes more work to create the administrative pages than it does to build the original application.

Menus and Toolbars

How can users gain easy access to standard operations across the application? Contemporary applications have several features that are designed to standardize the look and feel of applications and to make your applications relatively easy to use. **Menus** or **toolbars** and the **Help system** are common elements in most applications. Menus and toolbars are similar to each other and often created using the same techniques. A toolbar is a collection of items that perform some action

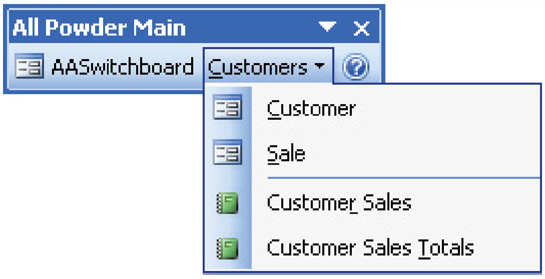


Figure 8.11

Sample menu. Note the hierarchical structure. Also, the underlined letter represents the access key, which can be activated from the keyboard. You can add shortcut keys (e.g., Ctrl+D), to activate a choice without going through the menu.

when clicked. The items can be icons or text. Text items can be opened to provide drop-down lists of additional items. This submenu makes it easier to organize the many choices. A toolbar that primarily consists of text items is often referred to as a menu, but the distinction is minor. Most systems enable you to create multiple toolbars or menus. Generally, you can modify them on the fly in response to user actions. With some systems, it is relatively easy to enable the toolbars so users can configure their own icons and selections on a custom toolbar.

A main menu is generally the same across the application. Hence, the menu centralizes choices that can be activated at any time. Menus are also useful for visually challenged workers and those who prefer to use the keyboard instead of a pointing device (mouse), because choices can be activated with the keyboard. Toolbars usually consist of a set of icons or buttons that perform common tasks. Some applications enable users to customize the toolbar with specific buttons and users often reposition toolbars.

Most menus are hierarchical: that is, detailed choices are presented under a few keywords. The Windows interface standard specifies that menus should be displayed at the top of the application. However, users may want to move menus to a different location. Most applications use similar commands on their menus. For example, the menu in Figure 8.11 contains top-level links for the main startup form, customer information, and help. Ultimately, entries would be added to cover the other main objects such as suppliers and animals. Whenever you create text items on a menu, you should define an access key so that users can select the entry directly from the keyboard. In a Windows environment, items are activated with the Alt key combination, such as Alt+C to open the main customers menu.

Purpose of the Menu

You might consider using the basic DBMS menu within your application. Then users will have full control over the database. In most cases, however, you will be better off building a custom menu for your application. A custom menu has several benefits. First, it can limit user actions. For instnace, if users do not need to delete data, the menu should not have the delete commands. You still have to set the appropriate security conditions to prevent them from using other methods to delete data, but removing a command from the menu helps to restrict user choices. A second advantage of a custom menu is that it simplifies the user interface. If en-

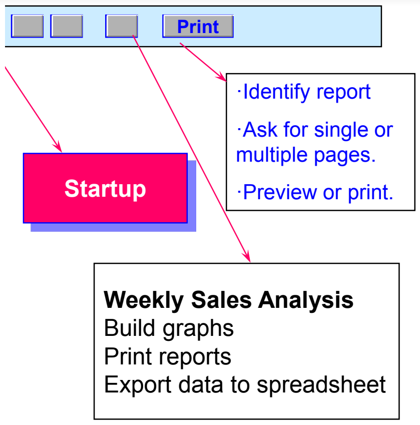


Figure 8.12

Sample toolbar. Toolbars can contain buttons and menus. Buttons generally display icons. When the pointer moves over them, a tooltip is displayed that briefly describes the button. When the button is clicked, an action is performed or a menu is displayed.

try-level users need only four or five commands, display only those options on the menu to make them easier to find. Third, you can add special functions to a custom menu. For example, you might add a special Help command to send e-mail to your support desk. Fourth, menu choices can be activated by keystrokes. Hence touch typists and visually challenged workers can use your application without looking at the screen.

Toolbars

Custom menus are usually implemented on toolbars. A toolbar contains a collection of buttons and menu items. When the user clicks a toolbar button, a predefined operation is executed. A toolbar can contain traditional buttons and textual menus. Most toolbars are **dockable**, which means that users can drag them to any place on the application window. Web-based forms rarely support dockable toolbars, but you can put menu options in a separate browser window or frame.

The purpose of a toolbar is to provide single-click access to complex actions or to commands that are used frequently. For example, many toolbars have an icon to immediately save the current work. As shown in Figure 8.12, you can put virtually any icon and any command on a toolbar. You can set different toolbars and menus for each form. You can even have multiple toolbars. For example, one toolbar might contain commands that apply to the entire application. Then special toolbars can be added as each form is opened.

Creating Menus and Toolbars

To support standardization and to simplify creating menus, most application development environments have a menu-generation feature. The exact steps depend on the system you are using; however, three basic procedures are used to create a menu: (1) Choose the layout or structure, (2) Give each option a name and an access key, and (3) Define the action to be taken when each option is selected. The main steps for creating a toolbar are similar except that you often create small graphical icons instead of text (step 2). When you create an icon, never assume that users will recognize an icon or understand what it represents. Most systems enable you to define a tooltip for each option. When the user moves the pointer over the icon, the **tooltip**, or short comment, is displayed. Every toolbar button must have a tooltip.

Creating toolbars and menus is straightforward with recent application development systems. You can customize an existing toolbar by adding or deleting options. Similarly, you can create a new toolbar. Button icons and menus can be dragged to the toolbar. The main step is to set the properties of each item. Menu names should be short and descriptive. You should also try to follow the standard names used in commercial software. To specify the access key, precede the key letter with an ampersand. For example, the *&File* text will appear as <u>F</u>ile, and the Alt plus F keys will activate that option. Shortcut keys (e.g., Ctrl + D) can be specified in the property settings of the detail menu item or the button command.

Most systems enable you to create multiple toolbars and then activate or deactivate toolbars for different users or in different areas within the application. You generally have to create a couple lines of code to activate or deactivate a specific toolbar.

Accessibility

How can a computer application be modified for people with disabilities? **Accessibility** is an important question, and it is also required for any application purchased by the U.S. Federal government. With the widespread adoption of graphical user interfaces in the 1990s, many people with disabilities encountered problems using the new applications. People with good vision might see value in dragging an object from one location to another, but many operations that rely on vision are unusable by other users. Certainly users with vision challenges will have problems, but it can also be difficult for other people to control a mouse or touch screen with enough detail to select and move items on a screen.

The most common methods to improving accessibility are:

1. Support multiple input methods (keyboard as well as mouse).
2. Do not put text into graphics, and use the Alt text tag to describe all images.
3. Use default and scalable fonts, do not use fixed sizes.
4. Select user-chosen colors instead of picking your own. In Windows,use defined values such as System.ControlText.
5. If you must pick your own colors on Web sites, use a style sheet and stick with high-contrast colors.

The goal is to ensure that your forms and applications accept multiple input methods. In particular, users should be able to navigate the application by using just the keyboard. Menu and toolbar selections should include keystroke options. Typically, these selections are made by using the *Alt* key along with other mnemonic keys. Short-cuts to specific actions are usually defined by *Ctrl* key combinations. The *Tab* key should move the selection point within a form to different fields with *Back-Tab* (Shift-Tab) moving in reverse. Of course, all of these keys need to be defined—preferably in one location with a list that can be read and memorized by users.

Along the same lines, use system-defined fonts and colors within your application. Avoid hard-coding a font size (such as 11 points) or color. With existing operating systems, users can define font sizes and colors that work best for them. When you hard-code sizes and colors in your application, your choices override those of the user. They might look good to you, but they could be invisible or highly annoying to the users. Remember that many people (as much as ten percent of the U.S. male population) can be red-green color blind. Most systems enable you to select colors based on the system-defined palette. For example, choose System.Text instead of “Black” and your application will pick up the values defined by the user.

In addition to input issues, your application needs to be conservative with graphics and images. Vision-impaired users often rely on screen readers to pick up the text from the page and read it aloud. In general, the screen readers cannot read text or interpret figures or directions written into images. When images are needed, be sure to enter text in the ALT tag that specifies in text what the image represents or critical information that it contains.

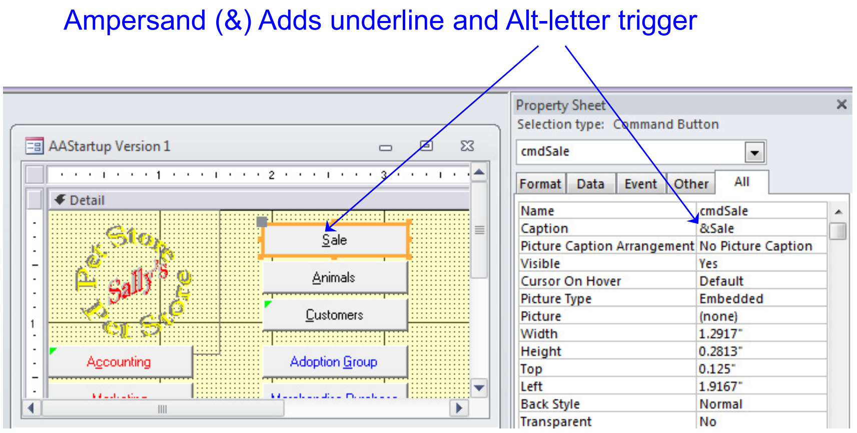
Most development systems today include tools to provide these standard features. However, typically you need to activate them and assign the keystrokes to them. As shown in Figure 8.13, menu and toolbars are often activated by adding an ampersand (&) in front of the hot key for that item. For instance an entry of *&Help* would generally be displayed with an underscore under the H as <u>H</u>elp. The inclusion of the ampersand tells the menu system to watch for the Alt-H combination to trigger that selection. All of the necessary tools are built into the platform, but as a developer, you must enter the ampersand every time you define a menu, button, or toolbar option.

These secondary input methods are also useful for people without vision problems. Because they are triggered from the keyboard, they can improve data entry speed for almost everyone. For example, practice with Word and Excel, using keyboard combinations to select menu items and you will find that many tasks become easier and faster because you do not have to move your hands from the keyboard to move the mouse.

One of the more interesting sources of ideas for accessibility is the U.S. government. The U.S. government has been required to implement accessibility options for several years. The ruling is known as “Section508” from the number of

Figure 8.13

Specify Alt-letter combination with ampersand. Every button, menu, and toolbar option should have a keyboard definition. Many systems use the ampersand (&) method before the key letter.



Property Sheet
Selection type: Command Button
cmdSale

Name	cmdSale
Caption	&Sale
Picture Caption Arrangement	No Picture Caption
Visible	Yes
Cursor On Hover	Default
Picture Type	Embedded
Picture	(none)
Width	1.2917"
Height	0.2813"
Top	0.125"
Left	1.9167"
Back Style	Normal
Transparent	No

the original statement. Government agencies have built an official Web site to discuss the topic: http://www.section508.gov/. A commercial site has similar notes: http://www.ada508.com.

In 2013, some discussion in the Federal government suggested that Congress might apply the same rules to commercial Web sites. The Americans with Disabilities Act (ADA) was written to require physical stores to provide access to everyone. So a few people have suggested that those with disabilities should also have equal access to any online site. It is not clear if the legislation has enough support to pass. It is also not clear that such a requirement is necessary. Presumably, some sites will find it useful to provide accessibility features. Is it truly necessary that all sites provide the same features? At the cost of opening up all Web sites to potential lawsuits. (The ADA has resulted in several nuisance lawsuits in the physical world.) But, some of the basic tenets of accessibility are straightforward and can be helpful to many users. Many of the suggestions for Web-based forms can be implemented on each page, such as including text descriptions of all functional images.

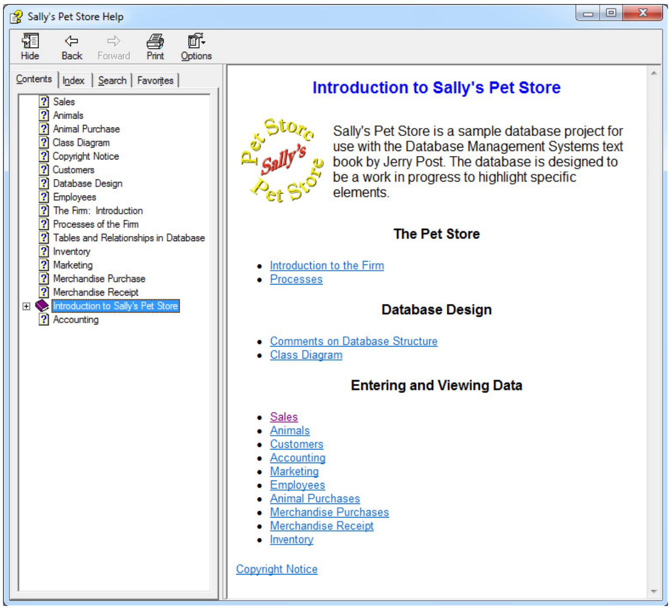
Custom Help

How do you create custom Help files? Online Help systems have grown to replace paper manuals. The goal is to provide the background information and the specific instructions that a user might need to effectively use the application system. Help files can contain text-based descriptions, figures, and hypertext links to related topics. As much as possible, the help messages should be **context sensitive**. The users should be presented with information that is designed to help with the specific task they are working on at the time. For instance, a user might want a definition of some term or more details about what actions can be performed on a specific page. Yet the Help system must also have an extensive search engine so that users can find information on any topic. Figure 8.14 illustrates a sample page from a Help system.

Microsoft has embedded a Help system within Windows for several years. This product has progressed through several versions. Most software developers use this system so users get a consistent Help system across all products. The Windows Help system displays the files, handles the links, table of contents, indexes, and searches almost automatically. As a developer, you can concentrate on creating the files that contain the basic information and the necessary links. Then the help compiler converts your data into a special file that the Windows Help system can display and search. Once you learn the basic elements of creating a page, the hard part is writing the hundreds of pages needed for a complete Help system. Most directors of large development projects hire workers just to write the Help files.

Help files built for the Windows system can be used with any application that runs on the Windows operating system. However, these compiled files do not work on the Web. If you are building an Internet-based application, you generally need to create a separate set of Help pages. The Oracle system provides its own Help compiler that you can use instead of the Windows system.

Most help systems today use HTML-based pages to display the text. Consequently, the help pages can also be used as support files for Web-based applications. However, the Windows-based systems always require some form of customization, so it is never as simple as just copying files.



Introduction to Sally's Pet Store

Sally's Pet Store is a sample database project for use with the Database Management Systems text book by Jerry Post. The database is designed to be a work in progress to highlight specific elements.

The Pet Store

- Introduction to the Firm
- Processes

Database Design

- Comments on Database Structure
- Class Diagram

Entering and Viewing Data

- Sales
- Animals
- Customers
- Accounting
- Marketing
- Employees
- Animal Purchases
- Merchandise Purchases
- Merchandise Receipt
- Inventory

Copyright Notice

Figure 8.14

Sample help screen. The Windows help system handles all of the display and searches. You just have to write the HTML topic pages and specify keywords.

Creating a Help File for Windows

The first and most important step in creating a Help file is to understand what information a user will need. Then you must write individual pages that explain the purpose of the system and how to use it. As with any communication project, you must first understand your audience. What types of people will use the application? What is their reading level? How much experience and training do they have with computers in general? Do they understand the business operations? The goal is to provide concise help information in a format that users can quickly understand. Usability studies show that most users do not want to use the Help system—they prefer systems that are easy to use. When users turn to the Help system, they are generally in a hurry and want a simple answer to a specific question.

Once you understand the needs of the users, you can write the individual Help pages. Five basic components are used to create a Help system: (1) text messages, (2) images, (3) hypertext links between topics, (4) keywords that describe each page, and (5) a topic name and a number for each page.

Microsoft currently supports two Help systems and is developing a third. But the company has experimented with several versions and it is not clear if newer versions are going to be carried forward. The original system generated HLP files. The second and most common one generates CHM files. A new system was designed for use with Visual Studio 2010. Its files are ZIP archives with a suffix of mshc. It is an improvement over Microsoft Help 2, which was used for Visual Studio 2003/2005 and Office 2007 (and 2010); but has been discontinued. The newer Microsoft Help Viewer (mshc) version has some useful features, but it might be a while before it is more widely implemented.

```
<object type=”application/x-oleobject”
classid=”clsid:1e2a7bd0-dab9-11d0-b93a-00c04fc99f9e”>
   <param name=”Keyword” value=”Contents”>
   < param name=”Keyword” value=”Introduction”>
   < param name=”Keyword” value=”Sally’s Pet Store”>
   < param name=”Keyword” value=”Management”>
</object>
<html><head>
<title>Sally’s Pet Store Introduction</title>
<link rel=”stylesheet” type=”text/css” href=”PetHelpStyle.css” />
</head><body>
<h1>Introduction to Sally’s Pet Store</h1>
<table><tr>
<td><img src=’PetStoreLogo2.gif’ border=’0’></td>
<td>Sally’s Pet Store is a sample database project for use with the
Database Management Systems textbook by Jerry Post. The database
is designed to be a work in progress to highlight specific elements.</td>
</tr></table>
<h2>The Pet Store</h2>
<ul>
<li><a href=’FirmIntroduction.html’>Introduction to the Firm</a></li>
<li><a href=’FirmProcesses.html’>Processes</a></li>
</ul>
</body></html>
```

Figure 8.15

Partial sample Help page. Create each topic as a separate Web page using HTML. The anchor <a> tag links to other pages. The <img> tag loads images. Use style sheets to set fonts and design. Use a table or a style to control layout. Place keywords for the page in the <object> tag.

The discussion in this section focuses on the CHM approach because it uses **hypertext markup language (HTML)** files—which are relatively easy to convert to Web based help. The newer Help Viewer system also relies on HTML files (technically well-formed XHTML pages), so the base concept is the same. Writing help files in HTML is relatively easy and many good tools exist for creating Web pages. However, be careful with the tools: Some of them, such as Microsoft Word, create complex code that might not work well with the Help compiler. You want to use an HTML editor that produces basic HTML code without relying on XML or JavaScript.

From a design perspective, it is crucial that you first design a style for your Help system and define that style using a cascading style sheet. A **style sheet** sets the typeface, font size, colors, and margins. The power of a style sheet is that you define all of the layout options in one place. Each page linked to the style sheet picks up those styles. So when you want to change the entire layout of your Help file, you make a few changes to the style sheet and every page uses that style.

Every topic is created as a separate HTML page. Users will be shown one page of material at a time. Try to keep topics short so they fit on one screen. Each Help page will contain links to other topics. Figure 8.15 shows part of a basic Help topic. Each page should have a title (marked with the <title> tag). Pages generally have links to other topics (using the HTML standard <a href> tag). Images can be in one of two formats: joint photographic experts group (JPEG) and graphics interchange file (GIF). Most Help images will be line-art drawings and should be in the GIF format. Most graphics packages can create and store files in these formats. When you save the file, use only letters and numbers in the filename—do not include spaces. Because you will eventually have hundreds of pages, it is a good idea to keep a separate list of the pages along with a short description of the topic and when it was last modified.

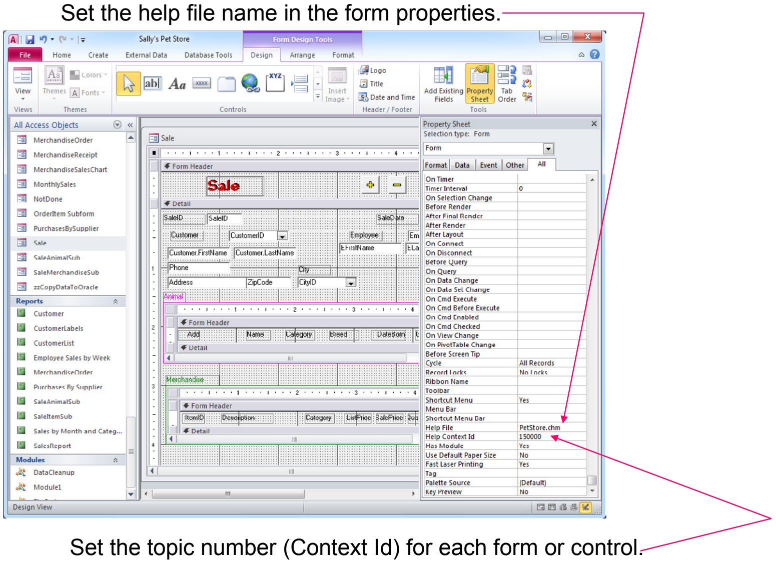
Keywords are an important part of every Help page. They are used to create an index for the user. An index lists the keywords alphabetically, when a user double-clicks a word, the corresponding Help page is displayed. The best way to create keywords is to enter them on each topic page. The easiest method is to copy the code from Figure 8.15 with the <object> tags and then change the keywords within that list for each page. Each keyword is listed with a separate <param> tag. If you want multiple levels, you can use a comma to list the hierarchy. For example, the three entries: (1) Sales; (2) Sales, Merchandise; and (3) Sales, Animal will create an index entry of Sales, followed by two indented lines for Animal and Merchandise.

Context-Sensitive Help

Consider an example of using help. Users working on the Sales form in your application do not want to wade through several Help pages or try to think of search terms. Instead, when they press the Help key, they expect to see information on that particular form. At a minimum, you need to create different Help pages for each form in your application. But now you need some method in your database

Figure 8.16

Setting context-sensitive help. In every form, enter the name of the Help file in the Help File property. Then enter the topic number for that form in the Help Context ID property. Every control or subform can also have a different Help topic—just enter the corresponding topic number.



Property Sheet
Selection type: Form

On Timer	
Timer Interval	0
On Selection Change	
Before Render	
After Final Render	
After Render	
After Layout	
On Connect	
On Disconnect	
Before Query	
On Query	
On Data Change	
On Data Set Change	
On Cmd Execute	
On Cmd Before Execute	
On Cmd Enabled	
On Cmd Checked	
On View Change	
On PivotTable Change	
Before Screen Tip	
Cycle	All Records
Record Locks	No Locks
Ribbon Name	
Toolbar	
Shortcut Menu	Yes
Menu Bar	
Shortcut Menu Bar	
Help File	PetStore.chm
Help Context Id	150000
Has Module	Yes
Use Default Paper Size	No
Fast Laser Printing	Yes
Tag	
Palette Source	(Default)
Key Preview	No

```
#define PetStoreIntro              100
#define Accounting               10000
#define Animal                   20000
#define AnimalDonation           30000
#define ClassDiagram             40000
#define Copyright                50000
#define Customer                 60000
#define DatabaseDesign           70000
#define Employee                 80000
#define FirmIntroduction         90000
#define FirmProcesses           100000
#define Inventory               110000
#define Marketing               120000
#define MerchandisePurchases    130000
#define MerchandiseReceipt      140000
#define Sale                    150000
```

Figure 8.17

Map file. Applications refer to topics by number, but the help system uses the filename. The map file (Topics.h) is a simple text file that assigns a number to each page.

application to specify which Help page should be displayed for each form. As shown in Figure 8.16, each form has properties for Help File and Help Context ID. Oracle and Visual Basic forms have similar properties. You enter the name of the file (e.g., PetStore.chm) in the Help File property. The Help Context ID requires a number. This number is a long integer and can range from 1 to more than 2 billion.

It is crucial to note that applications require a topic number, but your Help file refers to pages by their filename—not by numbers. To get these two systems to match, you must assign a unique number to every topic page. With HTML Help, you create a separate text file (usually called Topics.h) that maps this relationship. A sample file is shown in Figure 8.17. You can choose any number, but it is easier to remember them if you assign the numbers in groups. Also, with 2 billion numbers available, you can leave large gaps between the group numbers. For example, it is better to number by hundred thousands or millions instead of by ones (1, 2, 3, and so on). A useful technique is to assign numbers by business object (e.g., all Customer Help files might be numbered from 1,000,000 to 2,000,000). Once you have created the file, use the HtmlHelp API Information button (left side, fourth from the top) to tell the Help Workshop to include the file. Now, go through every form in your application and specify the file name and topic number for that form. Avoid changing the topic numbers in the Help file; they are hard to find in your application.

After you have created all of the files, you need to run the HTML Help compiler to combine everything into a single CHM file. A version of this tool can be downloaded free from Microsoft. Search for the htmlhelp.exe file. However, it is a relatively limited tool that can be cumbersome to use. Most development teams purchase a commercial product to gain more features including support for multiple writers. Several commercial tools exist at varying prices, and they generally include features such as support for multiple file types and version control.

```
<?xml version=”1.0” encoding=”utf-8”?>
<html xmlns=”http://www.w3.org/1999/xhtml”>
<head>
   <title>My  Page Title</title>
   <meta name=”Microsoft.Help.TopicLocale” content=”en-us” />
   <meta name=”Microsoft.Help.TopicVersion” content=”100” />
   <meta name=”Microsoft.Help.Id” content=”fadf1f04-77dd-43fb-81f6-72e5ae0bfc3d” />
   <meta name=“SelfBranded content=“true” />
   <meta name=“Microsoft.Help.Locale” content=“en-us” />
   <meta name=“Microsoft.Help.Package” content=“My_Help_Package_Pets_en-us_1” />
   <meta name=“Microsoft.Help.F1” content=“PetStore” />
   <meta name=“Microsoft.TocParent” content=“-1” />
   <meta name=“Microsoft.Help.Category” content=“Petstore::Introduction” />
   <meta name=“Microsoft.Help.ContentType” content=“Concepts” />
   <meta name=“Microsoft.Help.Keywords” content=“Introduction” />
   <meta name=“Microsoft.Help.Keywords” content=“Pet Store” />
   <meta name=“Description” content=“Basic description goes here…” />
   <meta name=“Microsoft.Help.tocOrder” content=“1” />
</head>
<body class=“primary-mtps-offline-document”>
   <div class=“topic”>
      <div class=“majorTitle”>This is the Page Title</div>
      <p>Sally’s Pet Store …. </p>
      <p><a href=“ms-xhelp:///?Id= fadf1f04-77dd-43fb-81f6-433e3ae08ac22”>My Link</a>

   </div>
</body>
</html>
```

Figure 8.18

Sample HTML Help 3. Meta tags within the file are used to define the basic features such as title, ID, table of contents location, and key words. Links use an ms-xhelp format to specify the ID of the link page.

Windows Help 3/Help Viewer

Microsoft might be changing the Windows Help system. Keep in mind that the company has tried at least at least two other times to create a new help system. The current version, loosely known as Help Viewer or Help 3 is an improvement over Help 2. Currently, the tool is only used to create help files that work within the Microsoft Visual Studio tool. However, there is a chance that the tool could be applied to other products in the future.

In terms of writing Help files for Help Viewer, the process is similar to the existing HTML Help: Begin by writing each topic in a separate HTML file. One important catch is that the HTML file actually needs to be XHTML—which is a more precise version of HTML that is compatible with XML. The headers are slightly different, and all tags must be complete. For example, a paragraph must have both a beginning and ending tag: *<p>My paragraph</p>*.

The other big difference is that all metadata is stored in the same file. There are no separate files for topics, keywords, table of content lists, or keywords. Everything is marked in the page using special tags. Figure 8.18 shows the basic format of a simple XHMTL help page. Note the use of meta tags to specify the items

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
 <title>An optional title.</title>
</head>
<body class="vendor-book">
 <div class="details">
  <span class="vendor">Pet Store</span>
  <span class="locale">en-us</span>
  <span class="product">Pet Store Sales </span>
  <span class="name">Pet Store</span>
 </div>
 <div class="package-list">
  <div class="package">
   <span class="name">package1</span>
   <a class="current-link"
    href="packages\package1.mshc">package1.mshc</a>
  </div>
  <div class="package">
   <span class="name">package2</span>
   <a class="current-link"
    href="packages\package2.mshc">package2.mshc</a>
  </div>
 </div>
</body>
</html>
```

Figure 8.19

Sample manifest file. A package is a single mshc help file. Name the entire manifest to: helpcontentsetup.msha and place it into the help archive folder.

needed to create the help file—notably the TOC specification and the key words. Entering a TOC value of -1 indicates that the entry (title) will be placed at the top of the hierarchy. To place an item lower in the hierarchy, simply enter the Help.ID value specified in the parent.

The nice thing about the new format is that you no longer need a help compiler to create the final help file. Simply create a new ZIP archive and place all of the text and image files in that compressed folder. Add a manifest file (helpcontentsetup.msha) and rename the archive from .ZIP to .MSHC. Figure 8.19 shows a sample manifest file with links for two "packages" or mshc files. Be sure to specify the names and locations correctly. For instance, the sample file refers to the pages stored within a "packages" subfolder.

At this point in time, your file will probably not open because Windows (and Office) are not set up for the new format. You might be able to use the HelpLibManager.exe program to install your new file and test it. Eventually, either the new help system will be adopted and integrated into Windows, or discarded for something newer (again). Either way, the hard part of creating Help files is identifying the topics and writing text that will actually benefit the users. These HTML files can be used for either of the current versions of HTML Help as well as standalone help files on Web sites.

Handling Errors

What does your application do when something goes wrong? Error handling is a task that is often relegated to coding on individual forms. However, it is a critical step—particularly in terms of security—so you need to review it at this stage of development. Also, error handling should be consistent across the application to avoid confusing users. At the same time, you need to create a logging facility so that runtime errors in the application are recorded and reviewed periodically so the application can be improved.

In terms of security, it is critical that your application catch all errors. Without special handling error messages can crash the entire application. Worse, they can lead to overwritten code providing an opportunity for criminals to take control of a machine. Even relying on the system error handling is dangerous, because default error messages often provide information that can be used to attack your application.

Catching Errors

Most development languages provide commands to trap runtime errors. Most of them use a variation of the try/catch syntax. The code to be protected is run within a try section. If an error occurs, execution is transferred to the catch section. Your error-handling code can look at different types of errors and handle them separately, or simply treat all errors the same. You ultimate mission is to devise error handling code that can automatically deal with common problems. The intelligence built into error-handling code is one way to tell the difference between amateurs and professional developers. Usually, you need users to help create errors so you know what to expect and the best way to handle them. The need for user testing is one of the reasons complex error handling is added at this stage of the application development.

Figure 8.20 shows the basic syntax for several programming systems. Most use a try/catch approach but the syntax varies. Note that Visual Basic is the same as C# but Basic does not use the braces. The basic structure includes a section

Figure 8.20

Common error-handling structure. Most systems use a try/catch structure but use different syntax to define the sections. The SQL 2003 standard supports various conditions (SQLEXCEPTION) and can EXIT the existing code or return to it (CONTINUE) after processing the handler code.

Oracle	SQL Server	C#	Access

BEGIN
  {code}
EXCEPTION
WHEN OTHERS THEN
  {code}
END
	BEGIN TRY
	  {code}
	END TRY
	BEGIN CATCH
	  {code}
	END CATCH
		try
		{
		   {code}
		}
		catch (exception e)
		{
		   {code}
		}
			ON ERROR GOTO errX
			  {code}
			exitX:
			  Exit Sub
			errX:
			  {code}
			End Sub

SQL 2003 Standard	DECLARE EXIT HANDLER FOR SQLEXCEPTION
	  Sql_procedure_name
	{code}		

or routine that is executed when an error arises. You have to make sure that each procedural code section is covered by at least one statement that directs errors to a handler. Then you can write code to identify the specific error and find ways to solve the problem or send a message to the user and exit gracefully.

One of the challenges of database programming is that procedural code can exist in two places: (1) Within procedures on the server, and (2) In routines such as C# that run on the client computer. You need to examine the application to be sure that both types of code are protected by error-handling routines. Errors that are trapped within database procedure code need to be returned to the client system to perform additional error handling, including displaying warnings to the user.

Logging Errors

An important step in trapping errors is to record them. Yes, it is helpful to display problems to the users; but users generally cannot do anything to solve the problem. You need to create a routine that inserts the error message, location, and date into a special table. Each error-handling routine should include a line of code to call this logging procedure. You might want to include additional data, such as values of key local variables, for complex procedures.

When the system has been running in production for a while, you can retrieve the values from the error-logging table. A simply query will show you which code sections cause the most problems and help identify the types of mistakes encountered by users. You use this information to fix code errors and write more intelligent error-handling code. The ultimate goal is to prevent users from having to deal with run-time errors. Your code should be able to identify and handle the main issues automatically. Of course, you cannot solve every problem—such as hardware or network failures—but you can identify them and give advice to the user.

Debugging

Once you know the approximate location of an error, you need to track down the cause. Fortunately, most contemporary systems have interactive debugging tools that enable you to set break points and step through the application code line by line. You can examine the values of local variables and even test queries.

The debugging process is more complicated when you have code running on a server, and considerably more difficult when code runs on multiple tiers including servers, clients, and middle-tier systems. Adding multiple levels requires you to track down the true location of an error. Depending on your tools and the final configuration, it is more difficult to run debuggers on multiple levels.

In many cases, you will have to resort to older debugging methods, such as adding debugging print lines to your procedures that report the current values of key local variables. In multi-tier systems, pay particular attention to the timing of events including the code and when variables are initialized and assigned values.

Testing

How do you know your application works correctly? Every application needs to be tested before it is turned over to users in a production environment. As indicated in Figure 8.21, many levels of tests can be performed, but ultimately, you can never catch all of the errors. Your goal is to find as many of the errors as possible with the time and money available. Keep in mind that errors caught earlier are easier and less expensive to fix, and it is better to catch errors before they cause expensive problems for users. Larger systems with multiple developers will

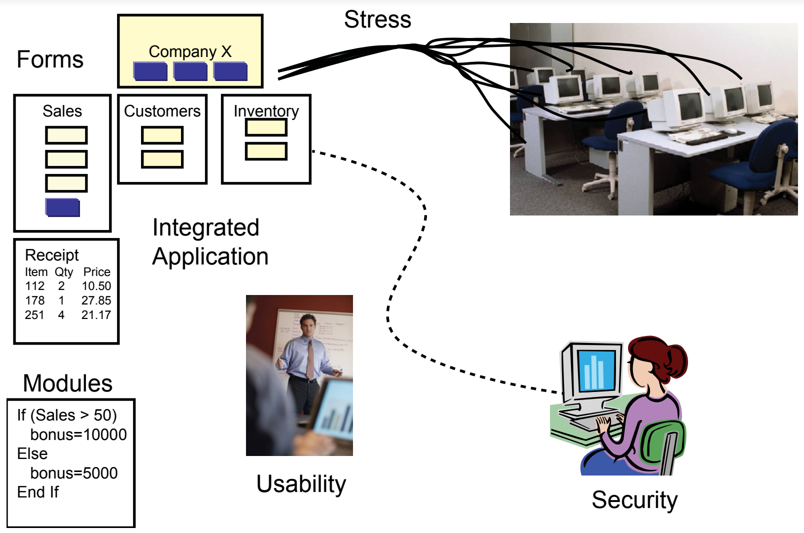


Figure 8.21

Application testing. Testing usually begins at the detailed level of forms and modules. When the application is built, the integrated features are tested. You also test for usability and performance under stress. Security testing should occur at every level.

require special groups of testers dedicated to finding problems. In smaller systems, you might have to test your own work. In either case, you should enlist the assistance of actual users who will always try things that never occurred to you. In a twist, test-based development is a modern approach to development where test cases are created first. Then whenever code is written or changed, the test cases are automatically rerun to ensure the code still works correctly. Several tools have been developed to help automate the testing process, but it still requires considerable time to develop all of the test cases.

Form and Module Testing

The most basic level of testing occurs when you create modules and forms. Any query, report, procedure, or section of code that you create should be tested as it is written. When you first create an object, you should understand its primary purpose and have sample test data to ensure that it works correctly. In particular, if you need to perform complex calculations or logic, you need to work with users to develop suitable test cases. These test cases should be stored and reevaluated at each testing point. Some organizations use pairs of developers, where one person is responsible for collecting test cases and continually testing sections of the project as it is being built.

You should also integrate the testing with form validation. Forms should restrict the data that users can enter to reduce the possibility of bad data, or even intentional attacks on your application. Where possible, you should use drop down lists and option choices so users do not have to type in values. When users do enter values by hand, you should include validation rules on the form to provide immediate feedback to the users so the data can be correct as close to the source as possible.

Integrated Application Testing

Once the overall structure of the application has been created, it needs to be tested as a complete unit. In particular, you need to ensure that data is passed correctly across forms, modules, and reports. Any forms that contain links need to be tested. For instance, linking a Sales Order form to the Customer form should result in displaying the details for the customer currently selected on the Sales form. But you also need to test extreme conditions. What happens if no customer has been selected yet and a user clicks the link button? Likewise, what happens if the user tries to print a blank receipt report? Be sure that the application continues to run even if absurd choices are made. Verify that data is being stored properly, and that security conditions are being maintained.

Stress or Performance Testing

Many developers and companies have encountered problems when an application hits the real world. Forms and reports that run fine on the developer's server die a slow death when pushed out to thousands of users with millions of rows of data. Unfortunately, systems performance is not always linear. For example, a task with 10 users might require 2 seconds to run; but you cannot claim that moving to 100 users will result in 20 seconds. More likely, instead of increasing by ten times, the time will increase exponentially, requiring 40, 50, or even 60 seconds. Some systems are more scalable than others—meaning that performance can be improved by adding hardware capacity and the process is close to linear. Other systems are more complex, but either way, you need to stress test the application to find out what will happen.

The challenge is that it is difficult to test big applications with thousands of users—without actually implementing the system. Where are you going to find the hardware and the thousands of users to test the system with sample data? Some companies sell tools that help stress test an application. The tools automatically generate transactions and send them through your application. You can increase the load on the servers by using only a few automated client computers. You can also test the servers and networks on a smaller scale by throttling down the hardware and networks. Instead of pushing 1,000 transactions through a 100 mbps network, you could test with 100 transactions across a 10 mbps network. It will not give exact results, but it will help you see what happens if a key connection gets overloaded. This test is particularly useful for connections from the Web server to the database server.

Usability Testing

In addition to testing for accuracy, errors, and performance, you also need to ensure that the system performs the tasks that users need. As part of the process, you need to have actual users work with the system. You need to be sure that users understand the forms and the process. The system needs to be easy enough to use so that it does not require huge amounts of training. It also needs to be efficient so that users do not waste time entering unnecessary data or searching for information. A developer can spend hundreds of hours building forms and applications. At some point, everything seems easy. You need the fresh perspective of actual users to identify bottlenecks and other issues. At one level, development is much like artistic design. Developers make dozens of choices when building applications. How do you make the best choice? The answer is that usability needs to become a key component right from the start. And the application specifically needs to be tested for usability.

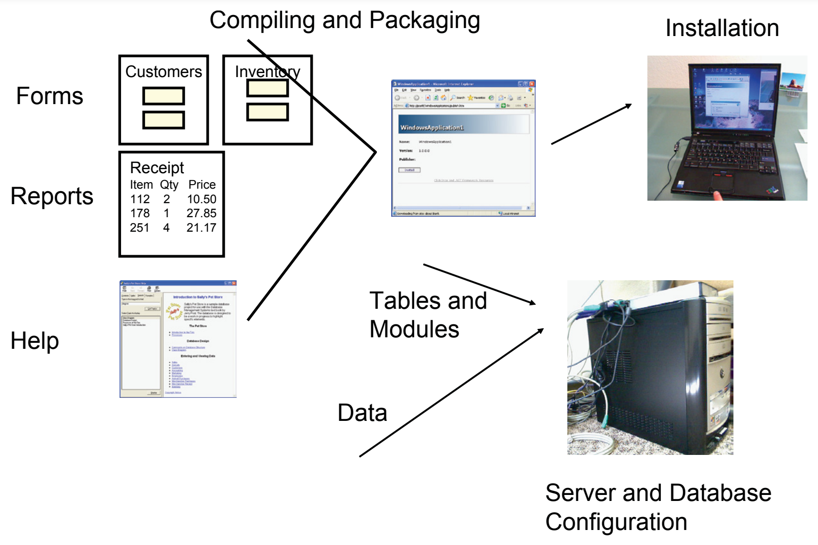


Figure 8.22

Deployment. The forms, reports, and help files are compiled and packaged into an installation file that is run on client computers. The DBMS is installed on a server and the tables and modules are installed and configured. Initial data is loaded and network connections are established.

Usability testing also needs to include testing the accessibility features. First, someone needs to go through the entire application and ensure all of the features are activated. It is too easy for a developer to forget to tag a button, menu, or toolbar so it is accessible with a keyboard. So someone needs to go through every item on every page and verify that accessibility is activated. Also, whenever possible, it would be useful to have someone with accessibility issues to actually use the application. A real-world test can provide valuable insight into the application flow, terminology issues, or other potential problems.

Security Testing

Security concepts are explored in other chapters, but companies have learned that security also needs to be addressed throughout the development process. It is not something that is added on at the end of the design. Testing for security issues includes some of the basic tests—particularly validation and module testing. It includes checking user input for common SQL injection attacks. A **SQL injection** attack consists of an attacker entering malicious SQL code into a text box in your application that replaces your intended SQL statement. The classic example is creating your own login screen and allowing users to enter any text as a username and password. The problem is compounded when you use string concatenation to build the SQL query. You should always use parameterized queries instead of string concatenation. More importantly, you should never trust anything entered by a user—and always restrict or validate what they are allowed to enter.

Consider the simple login example, where your application retrieves a UsernameText and PasswordText variable from the input screen. It is tempting to write the simple lookup query: “SELECT UserID FROM UserList WHERE Username=′ ” + UsernameText + “ ′ AND Password=′ ” + PasswordText + “ ′ ”. Ignoring the fact that the password should be encrypted, this query will work fine as long as users enter legitimate values. However, what happens when an attacker enters a special SQL string for the UsernameText: ′ OR 1=1 --. Plug this value in and write out the SELECT statement. The quotation mark closes the first one, the OR statement is always true, and the two dashes comment out the rest of the SQL command. As a result, the query will always return valid UserID and the attacker will be logged into your system. Worse, it is possible to write more complex SQL statements that do nastier things, such as retrieving all of the data from the UserList table, or even deleting tables in your database. However, all of the SQL injection attacks have a common element. They include the single quotation mark to close the required opening quote, and they use the double hyphen comment mark. The simplest solution is to test all input code and remove or change quotation marks and double hyphens to spaces. Whenever possible, you should restrict the length of data entered by users to prevent someone from writing long, dangerous code. Of course, restricting inputs can impact normal data entry, such as handling the name O’Brian which contains an apostrophe or single quote character.

Security testing also involves testing the entire application—including steps that might not be computerized. For example, how are passwords generated? What happens if a user loses a password—how is it reset? Is this process secure and logged? When the integrated application is being tested, you should also include basic security tests—particularly bad data that includes excessively long values and SQL injection elements. For large projects, at least one person should be assigned to attack the application, listing potential threats and methods that might be used to obtain unauthorized access.

Deploying an Application

How will your application be installed? As shown in Figure 8.22, once you have developed an application, you must collect all of the associated files (e.g., database, system, forms, reports, and help) and distribute them to users or install them on servers. You must also implement security precautions and assign user access rights. The details depend on the type of application system, whether the users are employees, and the size of the application. It is usually easiest to install applications on a server in one location. Even if you need a separate application (Web) server, installation and maintenance are relatively easy when the files and databases are in one location. If your application needs to install elements on client computers, several additional steps are needed.

Packaging Files

One of the first steps is to identify and collect all of the files. These primarily consist of the forms, reports, and help files. With a small application, built by one or two developers, it is relatively easy to identify and collect all of the files. With large applications that include hundreds of forms and reports, you need a version control system to name each file and track the versions and changes.

Some systems store forms and reports internally, some treat them as separate files, and a few compile them into a set of executable files. The method of packaging the files varies in each case, but it must still be done. You also need to test the resulting system to ensure all of the files are included and have the correct names. As much as possible, you need to automate the build process. Some systems include an automatic build, in other cases you will have to write script files. Either way, it is important to automate the steps because you will have to rebuild the application many times, and it is too easy to forget something. Scripts are easy to modify to avoid mistakes.

Installation Programs

If you are going to put any portion of the application on client computers, you need to use an installation program to automate the installation. Several tools exist, some versions are included with the DBMS and other versions are sold by independent vendors. Installation programs bundle the various files, check the target system for prerequisite files, and handle all configuration changes. Most of the installation tools support packages delivered on CD or downloads from a Web server. Some of the newer tools, including the one with Visual Studio, can be installed directly from a Web site and check for updates. The installation system also has to configure the database connections so the client component can attach to the database server.

Microsoft Access adds more complications to the installation process. In its most common form, the client computers will each need a licensed copy of the Microsoft Access software. You will also want to encrypt the database forms and reports to prevent users from changing them. In most cases, you will want to split the database into two pieces. Details are provided in the Access Workbook. Microsoft provides another alternative if you do not want to install the full copy of Access on each client computer. You can purchase the Access Developer kit which includes a runtime module. The installation system can install the runtime module so that you application will run without requiring a full copy of Microsoft Access.

Server and Database Configuration

An application also needs the servers and databases configured. The best way to handle the database configuration and base data loading is to write SQL scripts. You can create the scripts as the application is developed and tested. The scripts make it easy to load a new copy onto a test server. More importantly, they can be used to create a backup server or to reinstall the application is something goes seriously wrong. The applications associated with these books use script files, and you can use them as a template for your own applications. Even if you believe an application will only be installed one time, you should create the server script files. You will be surprised at how many times you will need to delete and reinstall an application while it is being tested.

Summary

An application is a collection of forms and reports designed to function as a system for a specific user task. Applications must be easy to use and designed to match the tasks of the users. Application design begins with the overall structure, which is often held together with startup forms. Menus and toolbars add structure to the application by providing commands that are common to the entire application. Toolbars can also be created for specific tasks and individual forms. A context-sensitive Help system with both general descriptions and detailed help notes is crucial to creating a useful application. Most applications also need to define individual transactions so that related changes will succeed or fail together.

You need to add error handling to all forms and modules and perform several levels of tests, including performance and security testing. You need to create a relatively automated approach to deploying the application—particularly if it needs to be installed on client computers. Several installation tools exist to package the files and support automated installation. For the database and server-based code, forms, and reports, you need to create SQL scripts that will create tables and load the basic data.

A Developer's View

Miranda is learning that applications are useful only if they make the user's job easier. A good application is more than just a collection of tables and forms. That means you have to organize the application by the tasks of the user. You also need to add help files and toolbars. You need to add error-handling code to your application. Once the application is fully tested, you need to create an installation package. For your class project, you should create the overall application structure (switchboard forms, interlocking forms, toolbars, help files, and so on). You should build and test the scripts and installation setup.

Key Terms

accessibility	hypertext markup language (HTML)
administrative tasks	menu
application	middle tier
back end	prototype
cascading style sheet	SQL injection attack
context sensitive help	style sheet
dockable	startup form
error handling	template
front end	toolbars
help system	tooltip

Review Questions

1. What are the fundamental principles to follow when designing an application's structure?
2. How does the purpose of an application (transaction processing, decision support, or expert system) affect the design?
3. ✓ How are startup forms commonly used?
4. What are the potential problems with startup forms?
5. What is the purpose of menus and toolbars in an application?
6. What features are needed to make an application more accessible?
7. What are the primary steps involved in creating a context-sensitive help file?
8. What are the major methods for handling runtime errors in an application?
9. ✓ What are the primary forms of testing?
10. What are the main steps in deploying an application?

Exercises

1. Find examples of two input forms—such as Web applications or business forms. Compare the applications on design and functionality. Explain the similarities and differences.
2. Find a Web site that has a separate mobile-based application. Explain the similarities and differences between the two types of forms. What features had to be sacrificed to make the mobile form? What choices would you have made differently?
3. HTML5 supports graphical actions, although the built-in capabilities are somewhat primitive. Assuming you have programmers to create them, design a new Web-based process to purchase items and handle shopping carts that use graphics and drag-and-drop elements. Just sketch the concepts—it is not necessary to create them.
4. Create a custom toolbar menu with at least two icons and two drop-down menus that include at least three options each.
5. ✓ Briefly explain how a touch-based menu would be different from a mouse-based menu.
6. Application menus can have many options. Briefly explain how you would solve the question of identifying the structure and items on menus.
7. Create a small custom help file that contains three pages of help. Create a form and assign the help key to open one of the help topics.
8. Examine at least three Web sites and explore their help sections. Briefly compare similarities and differences among the three sites. Explain which features you would use in your own applications.

9. Write a function to log runtime errors to a special database table or a file. Create a form with a button that contains error-handling code. When the button is pressed, it should trigger a runtime error (e.g., divide-by-zero), and call the logging function to save the error message.
10. Research and briefly describe the test-based development methodology and explain how it could be used in database applications.

Sally's Pet Store

11. Find at least two Web sites for pet stores and compare them. Select the primary features that you would want to use for a site for the Pet Store. Briefly explain how you would improve and differentiate your site.
12. Design and create a menu system and toolbars for the Pet Store database that would be used by clerks and managers in the store.
13. Design a template for the input forms. At a minimum, specify colors, fonts, and page layout. Rebuild at least two of the forms in the new template to test the styles.
14. Create and write initial help files for the Pet Store. Include at least three new pages of help, the table of contents, and keywords.
15. Find a user (non-CS and non-IS) who can test the application. Observe the user's progress and identify any problems or issues that arise. Describe changes you would make to improve the application.
16. Assuming the store is going to use the finished application, outline a plan to install and deploy it in the store on a single computer.

Rolling Thunder Bicycles

17. Examine the Rolling Thunder Bicycles application and outline the menu structure by checking the forms and reading the help file.
18. Explain how the list box is used to handle receipt of merchandise from suppliers. Outline the process that is used to tie the receipt to the purchase order.
19. Outline a plan for stress testing the application. Begin by identifying where the application will be used and how many people will likely use it at one time.
20. Design a new toolbar or menu that supports operations by categories of users: Managers, Order-clerks, Production, and Finance/Accounting. You can just sketch the toolbars/ribbons instead of actually building them.
21. Work through the application and test it for accessibility. Identify any changes that need to be made.

Corner Med

22. Design a menu or toolbar for Corner Med to make it easy to use within the clinics.
23. Identify potential application problems and failures that might arise and outline a plan to handle them. (Focus on software, not hardware or networks.)
24. Write the deployment plan for the application, assuming there will be one workstation at the central check-in desk and one in each physician office. Typically, there are three to five offices per location.
25. Design the Patient Visit form so it can be used on a mobile tablet with a 10-inch screen.

Web Site References

http://www.microsoft.com/enable/	Microsoft site for accessibility issues.
http://msdn.microsoft.com/windowsvista/uxguide	Microsoft design guide for Windows Vista.
http://www.sigapp.org/	Association for Computing Machinery: Special Interest Group on Applied Computing.
http://oraclea2z.blogspot.com/	Oracle application tips.
http://www.useit.com	Web site run by Jakob Nielson (a researcher in usability).
http://www.helpwaregroup.com/	Help authoring utilities
http://www.section508.gov	Federal government accessibility guidelines and blog.
http://www.w3.org/WAI	W3C (Web governance group) on the Web Accessibility Initiative.

Additional Reading

Cooper, A. *About Face: The Essentials of User Interface Design*. Foster City, CA: IDG Books, 1997. [A good discussion of various design issues.]

Ivory, M. and M. Hearst, The State of the Art in Automating Usability, *Communications of the ACM*, 33(4), December 2001, 470-516. [General discussion on evaluating system usability.]

Raskin, J. *Humane Interface, The: New Directions for Designing Interactive Systems*, Reading, MA: Addison-Wesley, 2000. [The need for a new interface as explained by the creator of the Apple Macintosh project.]

Chapter 9

Data Warehouses and Data Mining

Chapter Outline

What You Will Learn in This Chapter

- What is the difference between transaction processing and analysis?
- How do indexes improve performance for retrievals and joins?
- Is there another way to make query processing more efficient?
- How is OLAP different from queries?
- How are OLAP databases designed?
- What tools are used to examine OLAP data?
- What tools exist to search for patterns and correlations in the data?

A Developer's View

Miranda: Faster. Faster. Come on, run faster!

Ariel: What? Are you training for a marathon?

Miranda: No. It's just these queries they want me to write are taking forever to run. They worked OK when I tested them with small amounts of data. But now, I don't know.

Ariel: Maybe you just need a faster computer?

Miranda: No, I think I need a different system. These queries are retrieving data, but it is data from many different tables. And these managers want all of these strange subtotals.

Ariel: Wow! There are a lot of totals. How do you expect anyone to read those? I think I see four different levels of totals and that's on one page!

Miranda: Yes, and that's only part of what the managers want. I'm happy they are using the system, but I don't see how they can make any sense out of these reports. I think I might need a separate system to reorganize this data and create these reports for the managers. Then they want to do some type of statistical analysis as well!

Getting Started

Most companies have data and databases. What managers need are tools to organize and analyze the data. Relational databases are good for handling transactions data, but it can be difficult to retrieve and analyze huge amounts of data quickly. So you create a data warehouse with a structure designed to retrieve data quickly. Add cube browsers to explore the data. Add statistical tools to analyze the data, and managers can make better decisions.

Introduction

What is the difference between transaction processing and analysis? Relational database systems were designed to store large amounts of data efficiently. In particular, they are very good at quickly storing and retrieving basic transaction data. Look at the common Sale and SaleItem tables, and you will see data stored compactly. For example, the Pet Store SaleItem table has only four columns and they all contain simple numbers. An individual sale can be recorded or retrieved quickly. Each sale uses a different row, which separates transactions. Each new sale or item purchased can be entered into a new row without affecting any of the other rows or sales. However, this structure causes problems for other types of queries. Queries that involve multiple tables use joins that can require the DBMS to match data values from millions of rows. Think about the number of joins and subtotals required when someone asks the DBMS to analyze the data by computing subtotals on several different factors (such as employee, region, product category, and month). Computing breaks and subtotals across many factors, multiple tables, and millions of rows of data can cause performance problems even on fast hardware.

Vendors of database systems attempted to solve some of these problems by creating indexes on the tables. The indexes make it substantially faster for the DBMS to find specific rows of data within a table, and particularly to improve join performance. An index is a sorted list of the key data that can be searched quickly. However, there is a trade-off: Adding indexes to a table speeds retrieval queries but slows down data updates and transactions because the indexes continually have to be rebuilt. This conflict has led to focusing the existing relational systems for **online transaction processing (OLTP)**; whereas different storage and retrieval systems are used for **online analytical processing (OLAP)**. Data from the OLTP is extracted and cleaned, and then it is placed in a data warehouse. The data warehouse is heavily indexed and optimized for data retrieval and analysis. Additional procedures and routines are available to analyze the data, support interactive exploration by managers, and statistically search it for meaningful correlations and information. This chapter looks at the basic concepts to explain why the different approaches are needed. The data mining section defines some of the basic statistical tools available. The specific details of defining indexes and how to use the tools are covered in the workbooks because the details depend heavily on the specific DBMS. If you want more detailed explanations of the statistics and tools, check out the Data Mining book (http://www.JerryPost.com/Books/DMBook). It uses SQL Server and some open-source tools to examine common data mining applications.

Two-Minute Chapter

Relational databases are designed to efficiently store and protect transaction data. Splitting data into separate tables makes it faster and safer to add new rows. But, retrieving the data requires joining multiple tables on primary keys, which can be slow. Managers today need to analyze data—which can require retrieving and summarizing huge numbers of rows. Most systems add indexes to speed retrieval of data. An index is a separate sorted list of data with pointers into the actual tables. Adding indexes reduces retrieval time—which is good for data retrieval and analysis; but bad for data storage because every index requires time to update when data is added or changed. In many cases the best answer is to keep the relational database for transactions but create a new data warehouse that holds copies of the data optimized for data retrieval and analysis. In many cases, extracting and cleaning data from multiple sources is the hardest part of building a data warehouse. The steps need to be automated so data can be extracted on a regular basis.

A data warehouse is often organized in a star design with a fact table at the center, connected to dimension tables that contain attributes of interest to managers. Multi-dimensional cube browsers are useful for enabling managers to browse through data. The cubes display multiple levels of subtotals and managers can interactively select which dimensions and levels to display. Data is often organized in hierarchies (such as time) which managers can roll-up to view totals or drill down into for details. Be cautious when defining computed values—sums are usually fine but averages or computations requiring multiplication can be tricky because you need to specify whether multiplications should be computed first (use a query) or last (in the cube browser). The Microsoft PivotTable is an interactive cube browser that is easy to use and runs inside Excel and can also create interactive charts.

The SQL standard includes modifications to the GROUP BY clauses to display grand totals (super-aggregate totals). The WITH ROLLUP and WITH CUBE options combined with the GROUPING function are useful to compute these additional totals within SQL. SQL also includes the RANK and DENSE_RANK functions to assign sequential numbers to sorted data. For example, employees could be ranked by their total sales value for the month. RANK and DENSE_RANK treat ties differently, where DENSE RANK does not skip values so you would get values such as 1, 2, 3, 3 instead of 1, 2, 2, 4.

The SQL PARTITION command is used to create a "window" to examine a moving subset of data. It is particularly useful for computations such as moving averages such as averaging the three most recent data rows or for computing running subtotals. The LAG and LEAD functions provide access to data in rows behind or ahead of the current rows.

Reports are used to display data for common operations and transactions. Queries are used for ad hoc questions, and transaction programming. OLAP functions are used for aggregates, comparisons, and drill-down operations. Data Mining tools are used for deeper analysis and statistical techniques to identify unknown relationships. Common methodologies include classification, association rules, cluster analysis, and geographic analysis. Classification tools include decision trees, Bayesian analysis, and neural networks which attempt to identify how dimensions influence fact measure variables. The classic application of association rules is market basket analysis to see which items are commonly purchased together. Cluster analysis is used to identify categories or groups of items such as grouping customers who have similar features. Geographic analysis is useful for data based on location and often uses mapping systems to display layers of data.

Indexes

How do indexes improve performance for retrievals and joins? Although tables are often pictured as simple lists of rows and columns, a DBMS cannot simply store all data in sequential files. Sequential files take too long to search and

Figure 9.1

Find an item in a sequential table. Even if you know the primary key value, the system has to start at the first row and continue until it finds the desired match. On average, with N total rows, it takes N/2 row retrievals to find a particular item.

ID	LastName	FirstName	DateHired
1	Reeves	Keith	1/29/2013
2	Gibson	Bill	3/31/2013
3	Reasoner	Katy	2/17/2013
4	Hopkins	Alan	2/8/2013
5	James	Leisha	1/6/2013
6	Eaton	Anissa	8/23/2013
7	Farris	Dustin	3/28/2013
8	Carpenter	Carlos	12/29/2013
9	O'Connor	Jessica	7/23/2013
10	Shields	Howard	7/13/2013

require huge operations to insert new rows of data. Examine the short table of employees in Figure 9.1, and consider the steps involved to find the row where EmployeeID is 7. The DBMS would have to read each row sequentially and check the ID until it found the proper match. In this case, it would have to read 7 rows. If there are N total rows, on average, it takes N/2 rows to find a match. If there are a million rows, a typical search would require reading 500,000 rows! Clearly, this method is not going to work for large datasets. The situation is even worse for inserting new rows of data—if you want to keep the list sorted. The system would have to read each row of data until it found the location for the new row, then continue reading every other row and copy it down by one row. Deletions are actually easy because the DBMS does not really remove the data. It simply marks a row as deleted. Later, the database can be reorganized or packed to remove these marked spaces.

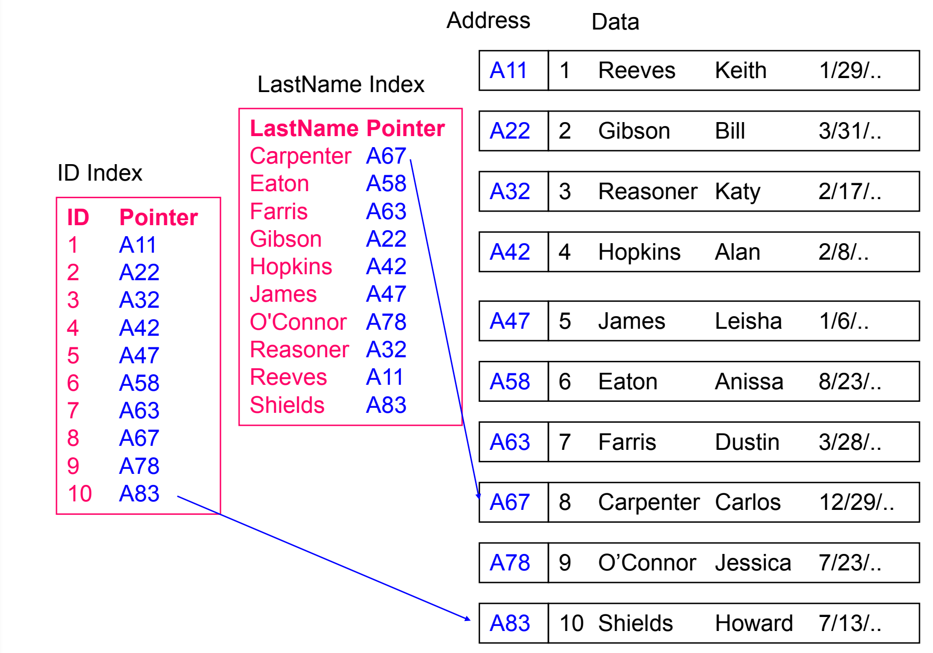
Binary Search

Looking at the data, it is clear that the DBMS is not taking advantage of all of the information. In particular, if the data rows are sorted, a substantially faster search method can be used to find items. Figure 9.2 shows how to take advantage of the sorting. Think of the process as searching through a paper dictionary or a phone book. Instead of starting at the first page and checking each entry, you would open the book in the middle, then decide whether to search the first half or the second half of the book depending on what name you find in the middle. Finding the middle entry of *Goetz*, you know that *Jones* falls in the latter half of the data. With that one retrieval, you instantly cut your search in half. Following the same process, you would divide the remaining entries in half and search only the appropriate section. In the example, only 4 attempts are needed to find the entry for *Jones*. This **binary search** process continues to divide the remaining data in half until the desired row is found. In general, with N total rows, a binary search will find the desired row in a maximum of m = log2(N) attempts. Another way to

Figure 9.2

Binary search. To find the entry for Jones, divide the list in half. Jones falls below that value (Goetz), so divide the second part in half again. Jones falls above Kalida. Continue dividing the remaining sections in half until you find the matching row.

	Adams
	Brown
	Cadiz
	Dorfmann
	Eaton
	Farris
1 ↓	Goetz
	Hanson
3 ↓	Inez
4	Jones
2 ↑	Kalida
	Lomax
	Miranda
	Norman



ID	Pointer
1	A11
2	A22
3	A32
4	A42
5	A47
6	A58
7	A63
8	A67
9	A78
10	A83

LastName	Pointer
Carpenter	A67
Eaton	A58
Farris	A63
Gibson	A22
Hopkins	A42
James	A47
O'Connor	A78
Reasoner	A32
Reeves	A11
Shields	A83

Address	Data			
A11	1	Reeves	Keith	1/29/..
A22	2	Gibson	Bill	3/31/..
A32	3	Reasoner	Katy	2/17/..
A42	4	Hopkins	Alan	2/8/..
A47	5	James	Leisha	1/6/..
A58	6	Eaton	Anissa	8/23/..
A63	7	Farris	Dustin	3/28/..
A67	8	Carpenter	Carlos	12/29/..
A78	9	O’Connor	Jessica	7/23/..
A83	10	Shields	Howard	7/13/..

Figure 9.3

Pointers and indexes. Each piece of data is stored in a location with a specific address. An index consists of the column value to be searched along with the pointer to the rest of the row. Multiple indexes can be assigned to a table and quickly searched.

understand this formula is to realize that because you cut the list in half each time, you are looking for m, where 2m = N. Now consider a table with a million rows of data. What is the maximum number of rows you have to read to find an entry? The value for m is 20, which is considerably better than the average of 500,000 for the sequential approach!

Pointers and Indexes

A binary search is a relatively good way to search data tables that are sorted, so it makes sense when you want to search by primary key, which is common for table joins. But what if the primary key is a numeric CustomerID and you want to search by LastName instead? How can the table be sorted in multiple ways? The answer lies with indexes and pointers.

Data is not actually stored in physical tables. It is usually broken into pieces and stored within a special file. When it is stored, each piece (perhaps an entire row) is placed at an open location and given an address. The address is a **pointer** that tells the operating system exactly where the piece of data is stored. It might be as simple as an offset number that specifies the number of bytes from the start of the file. Figure 9.3 shows that indexes can be created using the column to be searched (ID or LastName) along with the address pointer. The indexes are independent and have been sorted so they can be accessed quickly. As soon as the appropriate entry is found, the address pointer is passed to the operating system

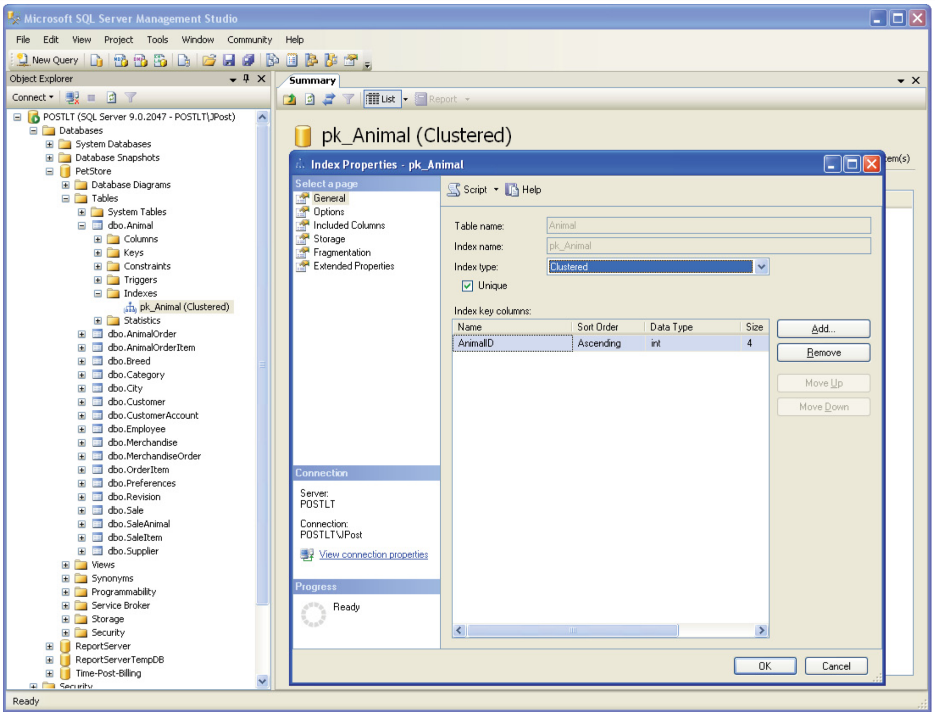


Table name: Animal
Index name: pk_Animal
Index type: Clustered
[✓] Unique

Index key columns:

Name	Sort Order	Data Type	Size
AnimalID	Ascending	int	4

Server: POSTLT
Connection: POSTLT\JPost
View connection properties

Figure 9.4

Index for primary key. SQL Server automatically generates and maintains indexes for primary key columns. Higher-end systems, such as SQL Server, provide several options to optimize the storage and use of indexes. These options can be used to improve the performance of your queries and applications.

and the associated data is immediately retrieved. In practice, even the indexes are not stored sequentially. They are generally stored in pieces as B-trees. B-trees can be searched at least as quickly as can be done with a binary search, and they make it relatively easy to insert and delete key values. B-trees are explained in Chapter 12, but you do not need to know the details to understand the benefits of indexes. You can create indexes in SQL using the CREATE INDEX command. Bear in mind that the reason for indexes is to substantially reduce the time it takes for the DBMS to find (or match) a particular row of data.

Creating Indexes

Most systems automatically create indexes for primary key columns, because these are typically used in JOIN statements. Figure 9.4 shows an example of an index created on the AnimalID primary key column of the Animal table in the pet store case. The example is from Microsoft SQL Server, but other systems, such as Oracle, are similar. Microsoft Access also generates primary key indexes automatically, but it has fewer options.

Most DBMSs have a graphical interface tool to create and edit indexes. However, it is relatively easy to use the SQL CREATE INDEX command. Figure 9.5 shows the basic format. The command is straightforward, since you just list the table name and the columns you want in the index. The example shows a composite

```
CREATE INDEX ix_Animal_Category_Breed
ON Animal (Category, Breed)
```

Figure 9.5

SQL CREATE INDEX command. The basic syntax is straightforward. Give the index a unique name, then specify the table and columns to be used. Most systems support additional options to control the details such as storage and type of index.

index that uses multiple columns. Most systems support additional keywords that control the various options, such as whether the index is unique, how and where it should be stored, or how to handle columns with long data types. In this example, you could include options to specify that the index contains unique values, or to store the index on a separate disk drive partition. These options are different for each DBMS, so you will have to read the documentation carefully to decide which ones you need. Alternatively, most systems provide a query optimizer that will automatically suggest indexes for you to add, with the desired attributes.

Problems with Indexes

Consider a table in which 10 indexes (columns) are defined. When a new row of data is added to the table, every index has to be modified. At a minimum, the database has to insert a new row into each of the 10 indexes. In most cases, it will also have to reorganize each index and probably update the statistics tables for the indexes. An index substantially improves the ability to search a data table. But for every index you create, the DBMS will slow down every time new data is entered or modified because the indexes have to be rebuilt. So your big decision is which columns to index.

Adding more indexes speeds up data retrieval but slows down data entry and data updates. This conflict is the heart of the problem between data analysis and transaction processing. Transaction processing—collecting the data—needs to be fast to efficiently store and protect the data. On the other hand, data analysis focuses on retrieving existing data and often needs to retrieve huge amounts of data quickly. Building multiple indexes and duplicating data are two ways to vastly improve data retrieval speed, but at the cost of interfering with colleting the data. A common solution is to create a data warehouse—which holds a copy of the data used just for data analysis.

Data Warehouses and Online Analytical Processing

Is there another way to make query processing more efficient? Ultimately, the trade-offs with indexes can be insurmountable. To perform complex searches, you need many indexes on every table. But too many indexes slow down the transaction processing. Additionally, a typical organization has data stored in several different databases and sometimes other files. Obviously, the transaction systems need priority—without them, the business cannot operate. But managers increasingly need to perform complex analyses of data. The solution: Keep the transaction systems and create a new database for managers to perform online analytical processing.

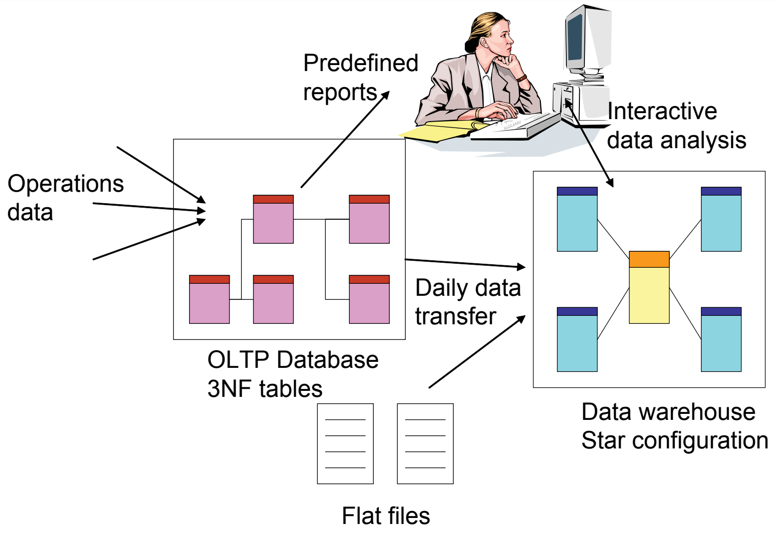


Figure 9.6

Data warehouse. Data from the OLTP system and other sources is cleaned and transferred into a data warehouse on a regular basis. The data warehouse is optimized for interactive data analysis.

Increasingly, managers want more than the traditional reports that are produced by OLTP systems. Managers want the ability to interactively examine the data. They do not always know what questions to ask or what they are looking for. They need the ability to quickly look at different views of the data. These types of queries can involve huge amounts of data and require joins across multiple tables. Fortunately, the access is almost always read-only—very little data is altered—and read-only queries can be several times faster than updateable queries. Managers also want the ability to statistically analyze the data, and these tools generally need to know something about the layout and structure of the data. The answer is to put a copy of the data into a new fixed structure.

Data Warehouse Goals

Many organizations have chosen to avoid these conflicts by creating a second copy of the database. A **data warehouse** holds a copy of the transaction data in a special database that is dedicated to answering managerial queries. Data may come from various sources, but all of it has been cleaned so that it is consistent and meets referential integrity constraints. Data can be stored in pre-joined format, resulting in duplication of data. But, since this data is not updated directly, and since storage space is relatively inexpensive, the duplication is well worth the increased performance. A second option is to build multiple indexes on every table. Again, since the data is not being continually updated, the indexes are rarely updated. In both cases, special functions and query controls are included to rapidly create different views of the data. Generally, data is transferred from the transaction system once or twice a day and moved in bulk to the data warehouse.

The basic concepts of a data warehouse are shown in Figure 9.6. The transaction databases continually collect data and produce basic reports, such as inventory and sales reports. The data warehouse represents a separate collection of the data. Although it might use the same DBMS, it requires new tables. On a regular basis, data is extracted from the transaction databases and from other files. This data is checked to make sure it is consistent; for example, all of the key values must match for referential integrity. Then it is added to the data warehouse, which usually does not store data in normalized tables. Instead, it has special structures like the star configuration. In these cases, data is often duplicated. For example, the same city and state combination may show up in thousands of data records.

Online analytical processing is usually related to data warehouses, but technically, you can build OLAP systems on transaction databases without using the intermediate data warehouse. A bigger challenge is that each vendor offers different technology and different implementations. In general terms, OLAP consists of a set of tools to browse the data and to analyze and compare data in the database.

Managers are also learning to use statistical tools to perform more formal analyses of data. **Data mining** or **business intelligence (BI)** tools use automated or directed statistical methods to search the data for patterns and relationships. The statistical tools include regression, discriminant analysis, pattern recognition (e.g., neural networks), and database segmentation (e.g., clusters, k-means, mixture modeling, and deviation analysis). These tools generally require substantial computing power and extremely high-speed data retrieval. Even with current high-speed systems, many of the techniques would need days or weeks to analyze some of the large datasets that exist. The point is that if users want to work on this type of analysis, the databases will have to be configured and tuned to their specific needs.

Data Warehouse Issues

Despite advances in database management systems and improvements in computer hardware, some queries take too long to run. Additionally, many companies have data stored in different databases with different names and formats, or even data stored in older files. The purpose of a data warehouse is to create a system that collects this data at regular intervals, cleans it up to make it consistent, and stores it in one location. A second primary goal of a data warehouse is to improve the performance of OLAP queries. In most cases, performance is improved by denormalizing the data. Joining tables is often the most time-consuming portion of a query, so new data structures are created that perform all of the joins ahead of time and store redundant data into fewer tables.

Three main challenges exist in creating a data warehouse: (1) Setting up a transfer system that collects and cleans the data, (2) Designing the storage structure to obtain the best query performance when handling millions or billions of rows of data, and (3) Creating data analysis tools to statistically analyze the data. Most companies choose to purchase data mining software for the third step. Few organizations have programmers with experience writing detailed statistical analysis procedures, and several companies sell prepackaged tools that can be configured to search data for patterns. The second issue—OLAP design—is discussed in the next section.

Cleaning and transferring data is often the most difficult part of establishing a data warehouse. Figure 9.7 shows the process known as **extraction, transformation, and transportation (ETT)**. You will quickly find that most companies have many different databases, with different table and column names, and different formats for the same type of data. For instance, one database might have a column Customers.LastName declared at 20 characters, and a second database uses Cli-

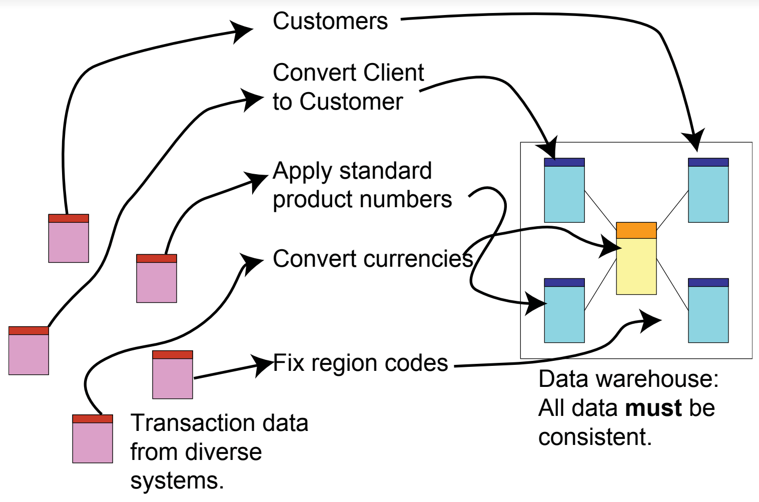

Customers
Convert Client to Customer
Apply standard product numbers
Convert currencies
Fix region codes
Transaction data from diverse systems.
Data warehouse: All data **must** be consistent.

Figure 9.7

Extraction, transformation, and transportation (ETT). Transaction data usually has to be modified to make it completely consistent. This process must be automated so it can run unattended on a regular schedule.

ents.LName set at 15 characters. The process of extracting data from these sources needs to be automated as much as possible; it is too hard and too expensive to try to clean data by hand. Remember that the data has to be transferred on a regular basis—at least daily. So you often have to write complex queries to merge data from different sources. In this small example, you would probably import one table (e.g., Customers), and then run a NOT IN query to get the list of names that are in the Clients table but not in the Customers table. These new names would then be added to the data warehouse. Some of the DBMS vendors have created import tools that will help you automate these data comparisons, but ultimately, most companies end up writing custom code to handle this complex process. For instance, Microsoft uses SQL Server Integration Services (SSIS). A key element in the process is to extract the data from the OLTP systems without interfering with the ongoing operations. Specialized tools and queries utilizing parallel processing on multiple-processor machines are often used in this step, but the details depend on the DBMS, the hardware, and the database configuration.

One method that can sometimes be used to reduce the data volume is to extract and transfer only data that has been changed since the last transfer. However, this process requires that the OLTP system track the date and time of all changes. Many older systems do not record this information for all elements. For instance, a sales database has to record the date and time of a sale, but it probably does not record the date and time that a customer address was changed.

Transforming the data often involves replacing Null values, converting text to numbers, or retrieving a value from a joined table and updating a value in the base table. All of these operations can be handled by SQL statements, and you will have to create modules that can be executed on a regular basis to extract the data, clean the data, and insert it into the new database.

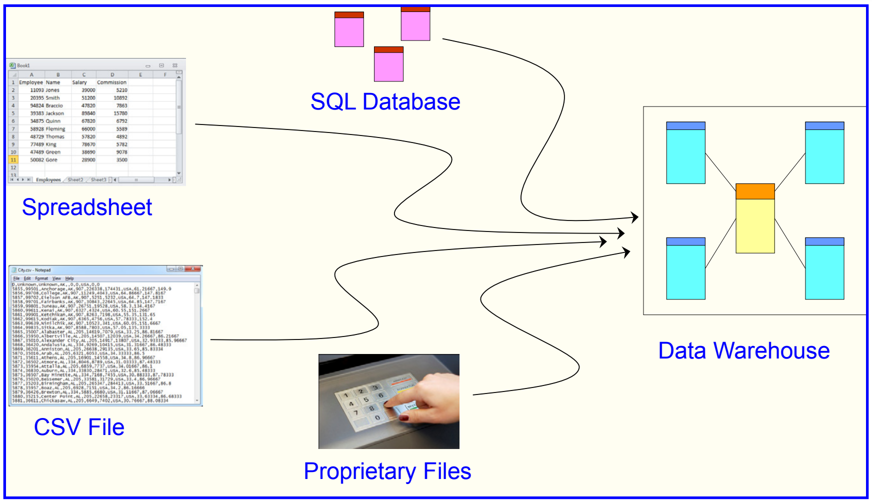


Figure 9.8

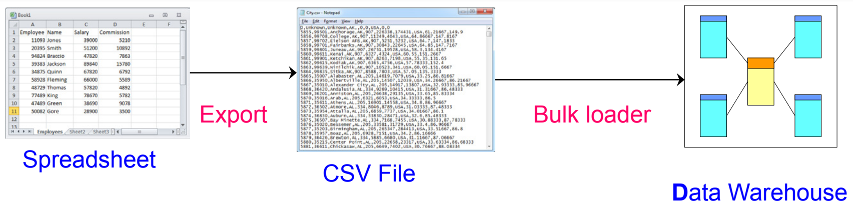
Data sources. The ETL process has to be automated so data can be extracted and loaded automatically every day. SQL sources are generally easy because tables can be linked and used directly. CSV files are relatively standard and can be handled with data loaders. Spreadsheets and other proprietary files can cause problems.

Data Extraction, Tansformation, and Transportation

How is data loaded into the data warehouse? One of the most difficult tasks in creating a data warehouse is setting up the extraction, transformation, and transportation (ETT) or loading (ETL) of data. Basically, you need to find all of the sources of data, find a way to extract it from its existing format, transform the data so it is internally consistent with every other piece of data, and load it into the data warehouse. More importantly, you have to create programs and tools so the entire process is automated. The ETL processes need to run on a timed basis (such as once a day). They need to run automatically, with no human intervention. As a database developer, it will be your responsibility to create and tests programs to handle these tasks. In large projects, developing all of the tools can easily take several months.

Figure 9.8 indicates the importance of focusing on the main sources of data: SQL databases, CSV files, spreadsheets and proprietary files. SQL databases are the easiest to handle. Most of the major DBMSs can be configured to connect to "external" databases. Hence, you can create connections from the data warehouse to the other databases. Once linked, you can write SQL statements to compare, transform, and copy data from the linked table into the warehouse tables. Another standard file type is the **comma separated values (CSV)** file. Data is stored sequentially in rows. The columns are separated by commas; although most tools enable you to change the delimiter to something else. For example, you might want to use tabs (ASCII character 9) in case the text data in a column happens to contain commas. In general, the bulk loaders make it relatively easy to import CSV data files.

Excel spreadsheets and other proprietary formats can be more challenging. In both cases, you might have to use the original tool (e.g., Excel) to save the data



	A	B	C	D	E	F
1	Employee	Name	Salary	Commission		
2	11093	Jones	39000	5210		
3	20395	Smith	51200	10892		
4	94824	Braccio	47820	7863		
5	39383	Jackson	89840	15780		
6	34875	Quinn	67820	6792		
7	58928	Fleming	66000	5589		
8	48729	Thomas	57820	4892		
9	77489	King	78670	5782		
10	47489	Green	38690	9078		
11	50082	Gore	28900	3500		
12						
13						

```
0,Unknown,Unknown,AK,,0,0,USA,0,0
5855,99501,Anchorage,AK,907,226338,174431,USA,61.21667,149.9
5856,99708,College,AK,907,11249,4043,USA,64.86667,147.8167
5857,99702,Eielson AFB,AK,907,5251,5232,USA,64.7,147.1833
5858,99701,Fairbanks,AK,907,30843,22645,USA,64.85,147.7167
5859,99801,Juneau,AK,907,26751,19528,USA,58.3,134.4167
5860,99611,Kenai,AK,907,6327,4324,USA,60.55,151.2667
5861,99901,Ketchikan,AK,907,8263,7198,USA,55.35,131.65
5862,99615,Kodiak,AK,907,6365,4756,USA,57.78333,152.4
5863,99639,Ninilchik,AK,907,10523,341,USA,60.05,151.6667
5864,99835,Sitka,AK,907,8588,7803,USA,57.05,135.3333
5865,35007,Alabaster,AL,205,14619,7079,USA,33.25,86.81667
5866,35950,Albertville,AL,205,14507,12039,USA,34.26667,86.21667
5867,35010,Alexander City,AL,205,14917,13807,USA,32.93333,85.96667
5868,36420,Andalusia,AL,334,9269,10415,USA,31.31667,86.48333
5869,36201,Anniston,AL,205,26638,29135,USA,33.65,85.83334
5870,35016,Arab,AL,205,6321,6053,USA,34.33333,86.5
5871,35611,Athens,AL,205,16901,14558,USA,34.8,86.96667
5872,36502,Atmore,AL,334,8046,8789,USA,31.03333,87.48333
5873,35954,Attalla,AL,205,6859,7737,USA,34.01667,86.1
5874,36830,Auburn,AL,334,33830,28471,USA,32.6,85.48333
5875,36507,Bay Minette,AL,334,7168,7455,USA,30.88333,87.78333
5876,35020,Bessemer,AL,205,33581,31729,USA,33.4,86.96667
5877,35203,Birmingham,AL,205,265347,284413,USA,33.51667,86.8
5878,35957,Boaz,AL,205,6928,7151,USA,34.2,86.16666
5879,36426,Brewton,AL,334,5885,6680,USA,31.11667,87.06667
5880,35215,Center Point,AL,205,22658,23317,USA,33.63334,86.68333
5881,36611,Chickasaw,AL,205,6649,7402,USA,30.76667,88.08334
```

Need to set a timer to automate the data export.
Timer runs in operating system, so you need an OS program to control the tool (Excel).

The bulk loader must run *after* the CSV file has been created.
If anything goes wrong, it will be difficult to fix automatically and a person probably needs to be called.

Figure 9.9

Problems with timing. The operating system has to run a program on a timer that calls Excel to export the data to a CSV file. After the file has been saved, the bulk loader can import the CSV data into the warehouse. If there is a delay or other problem, the system will likely crash and a human has to fix it.

to a CSV file then call the data warehouse bulk loader to import the CSV file. The problem with this approach is that it is more difficult to automate. As shown in Figure 9.9, you need to write a program in the operating system that uses a timer to start Excel and call an Excel macro to export the data to a CSV file. After the file has been saved, the program calls the bulk loader to import the CSV data into the warehouse. Then the warehouse programs can run to extract and transform the data. You need to know how to use several different programing tools to even create this process. It will be difficult to write a program to automatically catch all errors and fix them on the fly. More likely, if something goes wrong, the program will crash and you will be called to fix the problem. And those calls always come at 3 AM.

Any system that relies on multiple steps across different machines, operating systems, and software will have to be modified almost any time one of the components changes. For example, when Microsoft updates Excel or Windows, the programs will have to be tested and probably modified. The goal here is not to scare you (well, maybe a little); but to help you understand some of the challenges to developing ETL programs—and why they take so long to create and test.

One of the goals in building an ETL system is to get the data into a SQL data source as early as possible. Once the data is in relational tables, you can use the full power of SQL to compare and transform the data. You will make heavy use of SQL commands of the form: INSERT INTO warehouse_table (…) SELECT … FROM linked_table.

Remember that the SELECT statement can transform the data as it extracts it. Also, you can use NOT IN or LEFT JOIN clauses to choose only data that is missing or is not already in a second source table. If data needs several processing steps, you might have to write stored procedures or functions to perform more complex calculations. SQL Server has some useful tricks for creating temporary tables within functions and procedures. It is always best to stick with standard

SQL commands, but sometimes you need to rely on the more complex programming tools available.

The main step is to extract and transform the data so that it is internally consistent. Missing (Null) values are sometimes acceptable, but unmatched data is not. For instance, you cannot have a CustomerID in Sales data that lacks a related key in the Customer table. If the data all comes from a relational DBMS with referential integrity constraints, this problem is minimized. When the data sources include multiple databases, spreadsheets, and CSV files, all of the referential integrity constraints have to be built and tested as the data is loaded.

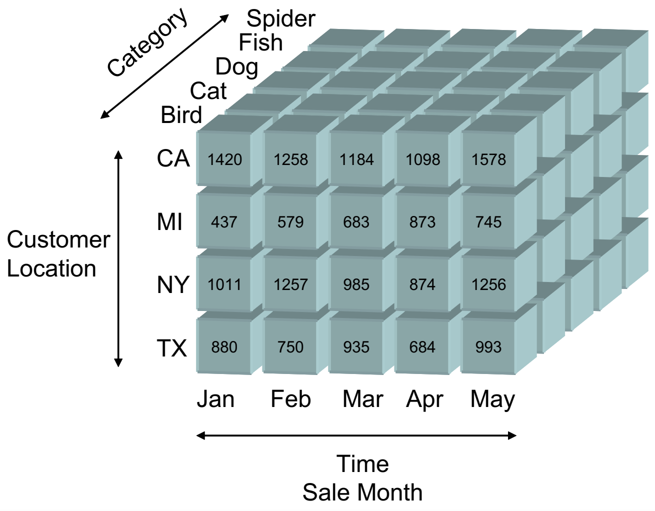
Once the data is consistent, the data warehouse has tools to define fact and dimension attributes. Most warehouses also support renaming attributes, adding descriptions, and assigning formats so that the attributes are easier to understand. For instance, if the database files use abbreviations such as CID or EID, you can assign the more descriptive titles CustomerID and EmployeeID.

OLAP Concepts

How is OLAP different from queries? Probably the most important goal of OLAP is to make the data accessible to managers. They should be able to browse through the data without having to write queries. The concept of the multidimensional cube shown in Figure 9.10 turns out to be a useful approach for many problems. The cube contains data about a specific **fact** (such as sales), and the **dimensions** (sides) represent factors that are potentially interesting to the managers. You could write queries to retrieve all of the data. In fact, the cube is probably defined by a query. However, managers do not want to write queries, and no one wants to assume that managers are going to write accurate queries. Consequently, managers use specific tools to examine the data interactively. For instance, Microsoft provides the PivotTable browser for use on the desktop. It can connect to any

Figure 9.10

Multidimensional cube. The fact element is sales. The dimensions are location, time, and category. Managers are interested in various combinations of the dimensions, and can use a cube browser to look at various subtotals.



common data source and enables managers to interactively see sections of the data and subtotals. Other vendors (and Microsoft) provide additional browsing tools.

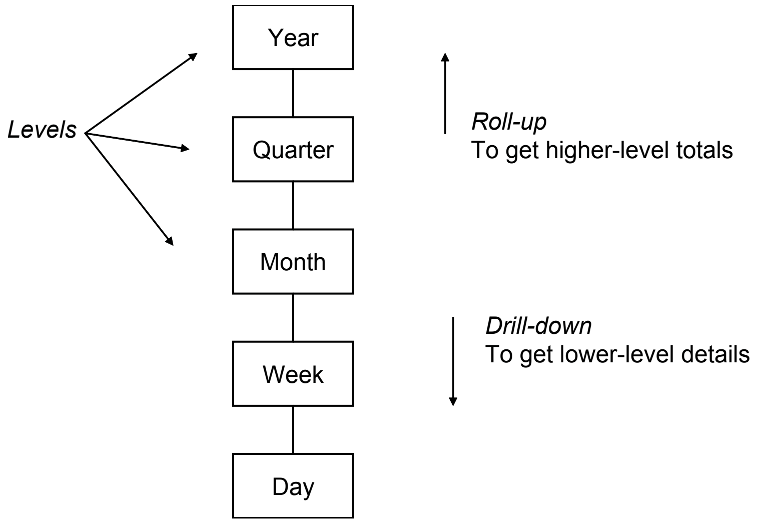
To illustrate the process, consider a simple example from the Pet Store database. Managers are interested in adoptions of animals. In particular, they want to look at adoptions by date, by the category (cat, dog, etc.), and by the location of the customer (state). The attribute they want to measure is the sale price, but you can also create more complex facts, such as price times quantity for merchandise sales. Figure 9.10 shows how this small query could be pictured as a three-dimensional cube. The OLAP tools enable managers to examine any question that involves the dimensions of the cube. For instance, they can quickly examine totals by state, city, month, or category. They can look at subtotals for the different categories or details within individual states. Currently, the front face of the cube shows sales subtotals by state and month. The cube browser makes it easy to rotate the cube to display a different face—such as sales by category over time, or category by state. Users can also examine just one slice of the cube, such as sales by location and category for a specific month. All of these options are performed without asking the manager to write SQL. The desktop tools support drag-and-drop operations to choose the dimensions to be compared.

The OLAP tools also support the ability to look at tools or to change and look at details. Managers might want to start with high-level subtotals and **drill down** to see the details. For instance, a manager might be looking at total sales by month and spot a drop in a particular month. He or she can drill down to see the details of sales by category or location within that month. The opposite of drill-down is to **roll up** the data into totals or averages. Instead of looking at detail sales for a given state, the manager might want to see the totals for an entire month.

A **data hierarchy** is another common element in OLAP. Many dimensions have an explicit hierarchy of values. For instance, Figure 9.11 shows the common

Figure 9.11

Drill down and Roll up. In a given dimension, drill down provides more detail. Roll up aggregates the values from subcategories.



hierarchy for dates. A given event (e.g., sale) occurs on a specific date, but that date is defined by the year, quarter, month, or week in which it occurred. Managers might want to examine data at any level within the hierarchy, or drill down or roll up as they are looking at one level. Several standard hierarchies exist in business data—such as dates and locations—and most systems know how to generate these levels automatically. However, the tools also enable you to create custom hierarchies for specific types of data.

Once the OLAP database is defined, users need tools to analyze the data. A cube browser is important because it enables users to look through the data and follow interesting observations. Statistical tools fall into the category of data mining or business intelligence. Vendors provide several versions of tools, some are more automated than others. The goal in all cases is to identify potentially interesting patterns.

OLAP Database Design

How are OLAP databases designed? Database design for OLAP is different from traditional database design. Some of the concepts are similar, but ultimately, most OLAP tools store the data in cube structures instead of relational tables. Additionally, OLAP design hides table joins from the end user. The manager sees only the cube. Consequently, the heart of OLAP design is to identify: (1) Facts to be measured, (2) Dimensions to be evaluated, and (3) Data hierarchies. The remaining design issues consist of choosing the best way to organize these elements.

Facts are relatively easy to identify. In a business context, a fact is often a dollar value, but you can also include counts of items, such as the number of items sold. In any case, you can simply ask the managers what items need to be measured. All facts must be **measures**—that is, they must be numeric values. For that reason, you cannot include categorical data (such as “small,” “medium,” or “large”). Some systems support multiple facts within a single cube, but they should be related. For instance, you might include the count of the number of items sold with the value of the items sold. Be careful to identify these values with distinctive and accurate names so users clearly identify the correct role of the data.

You need to be careful when the fact is a computed value. The problem is that you need to control the computational order. What happens if you build the cube using the original SaleItem table? Then you could only use Quantity and SalePrice as measures. It would be tempting to create a calculated measure: Amount2 = Quantity * SalePrice. However, this approach can lead to incorrect results. It is critical that you understand the difference between these two approaches. The correct method is to build a query for any computation that needs to be done on a line-by-line basis (Price * Quantity is a common example). If you wait and build it in the OLAP design cube as a calculated measure, then the cube will (1) slice the data, (2) subtotal any measures separately (Price and Quantity), then (3) perform your calculations: Sum(Price) * Sum(Quantity). So your calculations will be performed on data that has already been totaled. Figure 9.12 shows the difference with a small example. When you use a query for the fact table to compute the multiplication, the columns are multiplied first and then summed, giving the correct total or $23.00. If you use the original table as the fact table and specify the computation as the cube’s calculated measure, the cube first adds the quantity and price columns and then performs the multiplication, giving the incorrect result of $45.00. The solution is detailed line-by-line computations in a query and to use that query as the fact table.

Quantity	Price	Quantity*Price
3	5.00	15.00
2	4.00	8.00
5	**9.00**	**45.00 or 23.00**

Figure 9.12

Order of computations. Multiplications should be performed in a query that is used for the fact table to get the correct total of $23.00. Computing it in the cube calculation causes sums to be computed first and then multiplied to give the incorrect value of $45.00.

The second step is to choose the attributes or dimensions that form the sides of the cube. The dimensions come from columns for which the users want to compute subtotals. In one sense, an OLAP cube is like a SELECT statement with multiple GROUP BY statements. Any item that would appear in the GROUP BY clause becomes a dimension. The user gets to dynamically choose which dimensions to include at any time. The catch is that you have to find all of the tables that contain the desired dimensions and be sure they are linked to the desired facts.

The third step is to identify and generate all of the desired hierarchies within the dimensions. Some dimensions (e.g., dates) have well-known hierarchies. In other cases, you will have to talk with users to identify the desired levels and create the hierarchies manually. Each OLAP tool has a different method for defining hierarchies, so the actual steps are not covered here.

Snowflake Design

Once you have identified the facts and dimensions needed for a cube, you can construct the cube within the OLAP tool. Although the details vary, two general models are commonly used to store the data: the snowflake and star designs. The **snowflake design** is similar to a traditional relational design, so it is easy to understand. However, it might not be the most efficient design. Both designs begin with the fact table to define the desired measures. The difference lies in how the dimension data is stored and accessed. With the snowflake design, the system uses predefined joins to connect any tables. As shown in Figure 9.13, you can connect tables through other tables. For instance, City connects through Customer, which connects to the Sale table. The data remains in the original normalized tables, and all columns in the tables are available to be used as dimensions.

The difficulty with the snowflake design is that the OLAP browser needs to process the joins, which can require considerable computational power and time. Systems that use this approach rely heavily on indexes to reduce the access times. Often, the data is moved out of transaction tables, into a read-only set of tables. Since the data is rarely updated, the system can create a huge number of indexes without worrying about needing to update them because insertions and deletions are not supported. When data is transferred from the OLTP system, the indexes are removed, the data loaded, and the indexes are rebuilt at one time. Some tools also add internal pointers within the data, essentially integrating the indexes into the data for even faster performance.

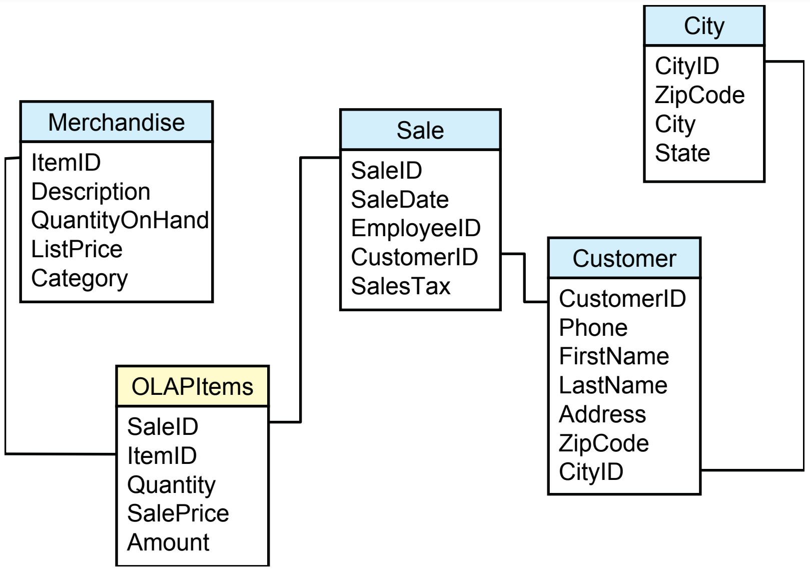


Figure 9.13

Snowflake design. It is less strict than the star design in that dimension tables can be joined to other dimension tables before being connected to the fact table.

Star Design

The **star design** focuses on speeding up data retrieval, essentially by removing joins. It accomplishes this task by denormalizing the data. Essentially, it saves duplicate data. In the standard sales example, the customer data would be entered for every sale. If you could scroll through the raw data, you would see the customer location repeated for every sale. Obviously, this approach requires more storage space. However, remember that insert, update, and delete are not supported on the individual items. Consequently, the problems discussed in Chapters 2 and 3 that are caused by non-normalized data are avoided. Figure 9.14 shows the star design for the sample sales problem. Once you understand the users' goals, the star design is relatively easy to create. You simply identify the fact measures and the dimensions. The system then copies all of the needed data to place the dimensions close to the fact measures. If you add enough dimension tables, you will see the reason for the star name. The fact table sits in the center and is connected to the dimension tables through rays. On the other hand, the snowflake design begins the same way, but you can add tables that connect through other dimensions instead of directly to the fact table. This extended pattern with multiple levels leads to a snowflake appearance.

Which design is better? This question is beyond the scope of this book, because it is a difficult question to answer. In fact, vendors continue to argue over the benefits and weaknesses of each method. They both work best for non-transaction data that is bulk-updated on a regular basis. Both require the storage of additional information (either indexes or duplicate data). In the end, performance depends on multiple factors. If you are thinking about buying a new system, you need to test your specific data with various systems and decide which approach works best in your situation. In terms of configuration, it is easiest to think in terms of the star design. Identify the facts and connect the dimensions directly to the fact table.

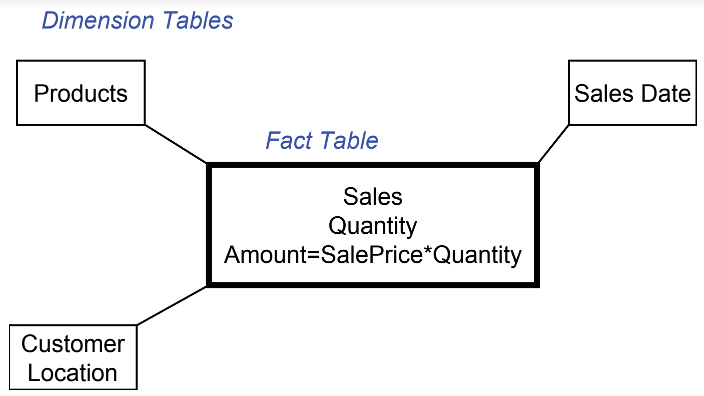


Figure 9.14

Star OLAP design. The fact table holds the numeric data managers want to examine. The dimension tables hold the characteristics. In a star design, all dimension tables connect directly to the fact table.

OLAP Data Analysis

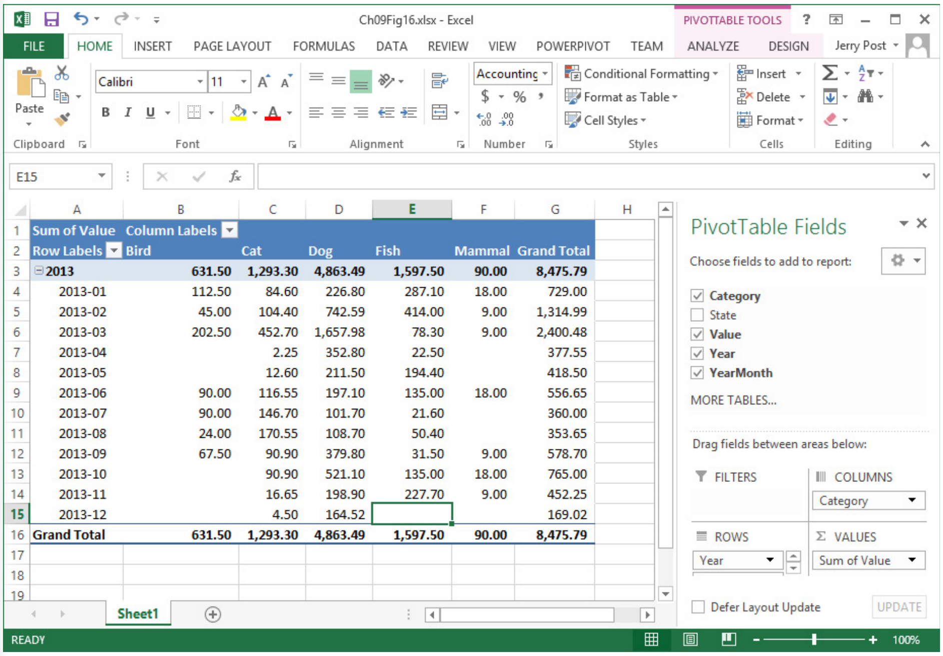
What tools are used to examine OLAP data? Beyond transaction processing, managers collect data to assist in making decisions. Many levels and types of decisions exist in business, so many different tools exist, with new ones created every year. Two general categories of tools exist: (1) Cube browsers, and (2) Statistical tools used for data mining. This section focuses on the cube browsers, and the following section summarizes some of the common data mining tools.

The most common form of cube browsers are interactive tools that make it easy for managers to examine subtotals, select subsets of the data, and drill down to see detailed data. Many vendors provide these interactive cube browsers, but several common features exist. Once you understand the overall structure, you can learn the details of a specific tool. The SQL standard of 2003 introduced SQL extensions to support retrieval and analysis of OLAP data. Additional standardization work concentrates on **multidimensional expressions (MDX)**, or the more recent mdXML. Although they are not interactive, SQL or MDX make it easier to write code that can be executed to retrieve or analyze data.

Cube Browsers

Vendors who provide OLAP tools generally include a cube browser to support interactive browsing by decision makers. All of the major DBMS vendors provide similar tools, but the construction and browsing techniques vary considerably. Also note that the business intelligence tools might require separate development tools and additional licenses (fees) to deploy to users. If you are using a DBMS that does not have an integrated BI system or cube browser, you can generally use Microsoft Office to build a PivotTable on the desktop that connects to your back-end database.

Figure 9.15 shows a sample cube for the Pet Store created with the Microsoft Business Intelligence Development Studio. The cube data was generated by creating a view to define the measures and dimensions related to sales of merchandise items. The Value fact was created as SalePrice*Quantity, and the SaleDate was


Ch09Fig16.xlsx - Excel
PIVOTTABLE TOOLS
FILE HOME INSERT PAGE LAYOUT FORMULAS DATA REVIEW VIEW POWERPIVOT TEAM ANALYZE DESIGN
Jerry Post
Clipboard Font Alignment Number Styles Cells Editing
Conditional Formatting
Format as Table
Cell Styles
Insert Delete Format
E15
Sheet1
READY
100%

Sum of Value	Column Labels					
Row Labels	Bird	Cat	Dog	Fish	Mammal	Grand Total
⊟2013	631.50	1,293.30	4,863.49	1,597.50	90.00	8,475.79
2013-01	112.50	84.60	226.80	287.10	18.00	729.00
2013-02	45.00	104.40	742.59	414.00	9.00	1,314.99
2013-03	202.50	452.70	1,657.98	78.30	9.00	2,400.48
2013-04		2.25	352.80	22.50		377.55
2013-05		12.60	211.50	194.40		418.50
2013-06	90.00	116.55	197.10	135.00	18.00	556.65
2013-07	90.00	146.70	101.70	21.60		360.00
2013-08	24.00	170.55	108.70	50.40		353.65
2013-09	67.50	90.90	379.80	31.50	9.00	578.70
2013-10		90.90	521.10	135.00	18.00	765.00
2013-11		16.65	198.90	227.70	9.00	452.25
2013-12		4.50	164.52			169.02
Grand Total	631.50	1,293.30	4,863.49	1,597.50	90.00	8,475.79

PivotTable Fields

Choose fields to add to report:

- [✓] Category
- [ ] State
- [✓] Value
- [✓] Year
- [✓] YearMonth

MORE TABLES...

Drag fields between areas below:

FILTERS	COLUMNS
	Category
ROWS	**VALUES**
Year	Sum of Value

[ ] Defer Layout Update UPDATE

Figure 9.15

An OLAP cube browser. The time (SaleDate) dimension is shown in the table of data along with the merchandise Category. Users can change the display simply by dragging the dimensions on or off the grid. They can also add filter fields such as the State dimension. The year-month-date hierarchy enables users to drill down or roll up data.

extended into a year-month-date time hierarchy. Once the cube is defined, managers can browse the cube without needing to know anything about the underlying structures.

Browsing the cube is as simple as dragging the desired dimensions and facts onto the display grid. Users can experiment at will, because the dimensions can always be interchanged or removed. Subtotals are automatically generated for hierarchies and the user can click the designated buttons to drill down or roll up the totals. Users can place dimensions on the page to use as filters. To show a different subset of data, the user simply opens the desired dimension (filter, row, or column) and selects the desired attributes. In the example, you could open the Sale filter and select only a couple of states to immediately see the Category and SaleMonth values for the chosen states. Starting with Visual Studio 2010 (and later 2012) the cube browser displays all dimensions in rows and does not support column headings. To see a more tabular approach, use a PivotTable.

Microsoft Office contains the PivotTable and PivotChart utilities that can run inside of Excel, or even deliver interactive Web pages. A PivotTable is the interactive cube browser. A PivotChart uses the same principles to display dynamic charts. The primary advantage of charts is the ability to visualize the data—particularly trends over time on line charts and correlations using scatter charts.

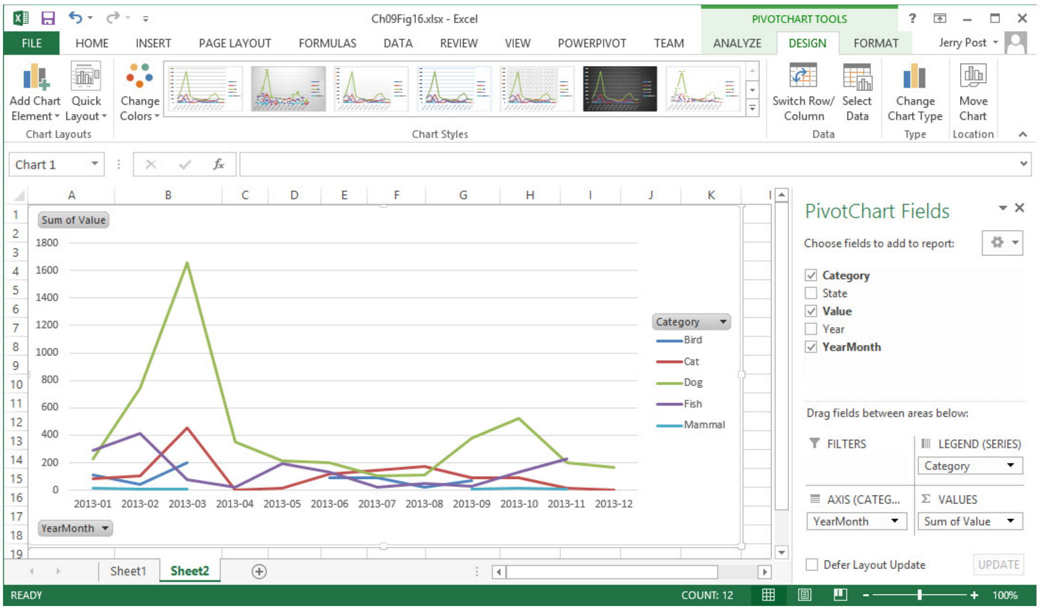


Figure 9.16

Microsoft PivotChart. Pivot tools make it easy for managers to examine cube data from any perspective, to select subsets of the data, to perform calculations, and to create charts.

The process of creating PivotTables and PivotCharts is similar. Although you could use Microsoft Query to collect and refine the data, it is usually easier to save a view in the original database that retrieves the desired data. In particular, you should create any needed calculations in the query. Microsoft Excel has menu options to help you create PivotTables and PivotCharts, so it is relatively easy to create and to use the resulting objects. Figure 9.16 shows a PivotChart based on the merchandise sale data. The operation of the PivotChart is similar to the cube browsers. Once the chart is built, managers can drag the dimensions around to create a new chart.

OLAP in SQL

Think about the concepts of the OLAP cube for a couple of minutes, and you will recognize that it is a method of examining the results of multiple GROUP BY statements. The cube browsers simply make it easier to display the results and interactively explore the relationships. Interactivity is nice but sometimes you need a programmatic approach to a problem. Perhaps you need a formal report, or to transfer data, or to automate a statistical analysis. The SQL 99 standard added some features that provide OLAP-type results within SQL. Several vendors have integrated these new commands, although the syntax might be slightly different for each vendor.

In the Pet Store example, what happens if you use a GROUP BY statement with two columns? Figure 9.17 shows the partial results of a Pet Store query that contains a GROUP BY computation with two columns (animal category and month sold). Notice that it provides a subtotal for each category element for each month.

```
SELECT Category, Month(SaleDate) As SaleMonth,
        Sum(SalePrice*Quantity) As Amount
FROM Sale INNER JOIN SaleItem
        ON Sale.SaleID=SaleItem.SaleID
  INNER JOIN Merchandise ON
        Merchandise.ItemID=SaleItem.ItemID
GROUP BY Category, Month(SaleDate)
```

Category	Month	Amount
Bird	1	135.00
Bird	2	45.00
⋮	⋮	⋮
Cat	1	396.00
Cat	2	113.85
⋮	⋮	⋮

Figure 9.17

SELECT query with two GROUP BY columns. You get subtotals for each animal category for each month. You do not see totals across an entire category (Birds for all months), and you do not get the overall total.

Figure 9.18

ROLLUP option. Adding the ROLLUP option to the GROUP BY statement generates the super-aggregate totals. In this case, the query provides totals for each Category element and the overall total. Notice that the corresponding Month is a null value.

```
SELECT Category, Month(SaleDate) As SaleMonth,
        Sum(SalePrice*Quantity) As Amount
FROM Sale INNER JOIN SaleItem
        ON Sale.SaleID=SaleItem.SaleID
  INNER JOIN Merchandise ON
        SaleItem.ItemID=Merchandise.ItemID
GROUP BY Category, Month(SaleDate) WITH ROLLUP;
```

Oracle syntax:

```
GROUP BY ROLLUP (Category, TO_CHAR(SaleDate, 'mm')
```

Category	Month	Amount
Bird	1	135.00
Bird	2	45.00
⋮	⋮	⋮
Bird	**(null)**	**607.50**
Cat	1	396.00
Cat	2	113.85
⋮	⋮	⋮
Cat	**(null)**	**1293.30**
⋮	⋮	⋮
(null)	**(null)**	**8451.79**

```
SELECT Category, Month(SaleDate) As SaleMonth,
        Sum(SalePrice*Quantity) As Amount,
        GROUPING (Category) AS Gc,
        GROUPING (Month(SaleDate)) AS Gm
FROM Sale INNER JOIN SaleItem
        ON Sale.SaleID=SaleItem.SaleID
  INNER JOIN Merchandise ON
        SaleItem.ItemID=Merchandise.ItemD
GROUP BY Category, Month(SaleDate) WITH ROLLUP
```

Category	Month	Amount	Gc	Gm
Bird	1	135.00	0	0
Bird	2	45.00	0	0
⋮	⋮	⋮	⋮	⋮
Bird	**(null)**	**607.50**	0	**1**
Cat	1	396.00	0	0
Cat	2	113.85	0	0
⋮	⋮	⋮	⋮	⋮
Cat	**(null)**	**1293.30**	0	**1**
⋮	⋮	⋮	⋮	⋮
(null)	**(null)**	**8451.79**	**1**	**1**

Figure 9.19

GROUPING function. The GROUPING function returns a value of one when the row displayed is a super-aggregate for the selected column parameter.

Assuming all animal types were sold in all months, you would see 12 values for birds, 12 for cats, 12 for dogs, and so on. What you do not get are **super-aggregate** totals, or totals for an entire category or across all rows. For instance, what is the total value of bird merchandise sold for the entire year?

ROLLUP

You could get these super-aggregate totals by using additional SELECT statements. However, SQL 99 added the ROLLUP option specifically to compute super-aggregate totals. Figure 9.18 shows the results for the Pet Store query. The total across all months is calculated for each element in the Category column. This total is displayed with a null value for the Month column. At the bottom, the overall total is displayed with two null values. Of course, the super-aggregate totals are not normally printed in bold, so they can be hard to spot. A bigger question is, What happens if there is a missing (null) value for some months? In the case of a missing date for a sale of bird items, the display would contain two similar lines (Bird, null, 32.00). One of the lines would be the total sales of bird products for months with missing dates. The second total would be the super-aggregate total across all months. But how do you know which is which? It is possible to scrutinize the numbers with totals and realize that the larger total should be the super-aggregate value. But with other functions, such as Average, there might not be any way to tell. Notice that the Oracle syntax is slightly different from the SQL Server syntax. The Oracle version is slightly closer to the standard (which does not require the parentheses), but you should understand both versions.

```
SELECT Category, Month(SaleDate) As SaleMonth,
        Sum(SalePrice*Quantity) As Amount,
        GROUPING (Category) AS Gc,
        GROUPING (Month(SaleDate)) AS Gm
FROM Sale INNER JOIN SaleItem
        ON Sale.SaleID=SaleItem.SaleID
  INNER JOIN Animal ON
        SaleItem.ItemID=Merchandise.ItemID
GROUP BY Category, Month(SaleDate) WITH CUBE
```

Category	Month	Amount	Gc	Gm
Bird	1	135.00	0	0
Bird	2	45.00	0	0
⋮	⋮	⋮	⋮	⋮
Bird	**(null)**	**1358.82**	0	**1**
Cat	1	45.00	0	0
Cat	2	113.85	0	0
⋮	⋮	⋮	⋮	⋮
Cat	**(null)**	**1293.30**	0	**1**
⋮	⋮	⋮	⋮	⋮
(null)	**(null)**	**8451.79**	**1**	1
(null)	1	**1358.82**	**1**	0
(null)	2	**1508.94**	**1**	0
⋮	⋮	⋮	⋮	⋮
(null)	12	**164.70**	0	0

Figure 9.20

CUBE option. The CUBE option computes super-aggregate values for all columns in the GROUP BY statement. The rows near the bottom with the Gm indicator value of 1 are the totals by month for all categories of products.

To help identify the super-aggregate lines, the SQL standard introduced the GROUPING function. As shown in Figure 9.19, the function usually returns a value of 0. When the row displayed is a super-aggregate computation, it displays a value of 1. In the example, the totals across months for each category produce a value of one for the GROUPING(Category) function. The overall total contains values of one in both indicator columns. This function could also be used in other computations or even in WHERE conditions. For instance, you might want to perform a computation with the super-aggregate totals.

CUBE

Looking at the results, it is clear that the ROLLUP option does not provide all of the information a manager might want. Notice that the super-aggregate totals only apply to the Category column in the examples. There are no corresponding totals for the Month column, which would represent sales of all categories for a given month. Of course, you could obtain those totals if you rewrite the query and reverse the order of the Category and Month columns in the GROUP BY clause.

The CUBE option provides the solution. The CUBE option is similar to ROLLUP, but it computes and displays the super-aggregates for all GROUP BY columns. In Figure 9.20, notice that the only change to the SQL was replacing the

```
SELECT Category, Month(SaleDate) As SaleMonth,
        Sum(SalePrice*Quantity) As Amount
FROM Sale INNER JOIN SaleItem
        ON Sale.SaleID=SaleItem.SaleID
 INNER JOIN Merchandise ON
        SaleItem.ItemID=Merchandise.ItemD
GROUP BY Category, Month(SaleDate) WITH CUBE
HAVING GROUPING(Category)=1 Or GROUPING(Month(SaleDate))=1
```

Category	Month	Amount
Bird	(null)	607.50
Cat	(null)	1293.30
⋮	⋮	⋮
(null)	(null)	8451.79
(null)	1	1358.82
(null)	2	1508.94
⋮	⋮	⋮
(null)	12	164.70

Figure 9.21

GROUPING SETS to hide detail. The GROUPING function can be used to hide the details so users can focus on the super-aggregate totals.

ROLLUP keyword with CUBE. The result still includes the super-aggregate totals across months for each category. These totals have a value of 1 for the Gc indicator column. But, the query also produces the super-aggregate totals for each month across all categories of products. The values for the three months are displayed near the bottom of the results. Notice the null value under Category, and the Gm column value of 1 indicating that it is the super-aggregate total for the month. Again, the Oracle syntax is slightly different, where the key phrase becomes: GROUP BY CUBE (Category, TO_CHAR(SaleDate, 'mm')).

Because of these additional totals, you will most likely use the CUBE option more often than ROLLUP. However, if you add several columns to the GROUP BY statement, you could get so many subtotals that you might prefer to use ROLLUP to simplify the display. Ultimately, the decision comes down to what the users need to see, or which values you need in additional computations. Remember that you cannot rely on the null value to identify super aggregates. You must use the GROUPING (e.g., Gc and Gm) function instead.

The SQL standard provides additional options, including the ability to create CUBEs or ROLLUPs based on the combined value from multiple columns. The standard calls for a GROUPING SETS function to hide the detail subtotals and only display the super-aggregate totals. However, this function is not supported by all systems, and it is actually easier to use the GROUPING function directly. As shown in Figure 9.21, the SQL is straightforward by adding the conditions to a HAVING statement.

Although the ROLLUP and CUBE options bring new features to SQL, the results can be difficult to read. In terms of OLAP value, you would not want to show the results to managers or expect them to be able to use these tools interactively. On the other hand, they could be useful for feeding data into a procedure that you write which needs to perform more advanced computations or transfer the data to a spreadsheet.

SQL Analytic Functions

The SQL-99 standard added some mathematical functions that are useful for common OLAP analyses. For example, the statistical functions of standard deviation (STDDEV_POP and STDDEV_SAMP), variance (VAR_POP and VAR_SAMP), covariance (COVAR_POP and COVAR_SAMP), correlation (CORR), and linear regression (REGR_SLOPE, etc.) are now part of the standard. Because most database systems already had proprietary versions of these functions, the impact is not that great, but it will help if vendors adopt the standard names for the functions.

Two of the more interesting new functions are **RANK** and **DENSE_RANK**. These functions assign numbers to the sorted results that indicate the ranking of the data. A new table (SampleSales) was created to illustrate the functions. The sample data was created specifically to illustrate the difference between the two functions. You could create a view in the Pet Store database and run the same query, but the results will be different. Figure 9.22 shows the query and the results. First, note that the syntax is somewhat complicated. The reason for the complexity is because these two functions are designed to work with partitions—which are explained in the next section. The query in this example uses all of the rows of data, but you still need the OVER clause to specify the correct sort order.

Look closely at the results, and you will see the difference between the RANK and DENSE_RANK functions. If you have every worked with ranked data (such as sports or election results), you know the basic problem: How do you handle ties? Both functions give tied values the same rank (2 in this case). But the RANK function keeps counting the number of entries and assigns a rank to the next non-tied value that includes all of the entries above it. In this example, White receives a rank of 4 because three people have higher sales. The DENSE_RANK function counts the ranks instead of the rows. Consequently, White receives a dense rank of 3 because it is the next ranking value. You can choose whichever function you need for a particular problem. The syntax is the same, but you have to remember the difference between the two.

Figure 9.22

RANK functions. The sort order for the rank function is specified separately. Ties are given the same rank. RANK skips values that would have been assigned to tie values. DENSE_RANK does not skip values.

```
SELECT Employee, SalesValue,
RANK() OVER (ORDER BY SalesValue DESC) AS Rank,
DENSE_RANK() OVER (ORDER BY SalesValue DESC) AS Dense
FROM SampleSales
ORDER BY SalesValue DESC, Employee;
```

Employee	SalesValue	Rank	Dense
Jones	18000	1	1
Black	16000	2	2
Smith	16000	2	2
White	14000	4	3

```
CREATE VIEW qryMonthlyMerchandise AS
SELECT Category,
  TO_CHAR(SaleDate, ‘yyyy-mm’) As SaleMonth,
  sum(SalePrice*Quantity) As MonthAmount
FROM Sale INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
  INNER JOIN Merchandise ON Merchandise.ItemID=SaleItem.ItemID
GROUP BY Category, TO_CHAR(SaleDate, ‘yyyy-mm’)

SELECT Category, SaleMonth, MonthAmount, AVG(MonthAmount)
   OVER (PARTITION BY Category
        ORDER BY SaleMonth ASC ROWS 2 PRECEDING)
   AS MA
FROM qryMonthlyMerchandise
ORDER BY Category, SaleMonth;
```

Category	SaleMonth	MonthAmount	MA
Bird	2013-01	135	135
Bird	2013-02	45	90
Bird	2013-03	202.5	127.5
Bird	2013-06	67.5	105
⋮	⋮	⋮	⋮
Cat	2013-01	396	396
Cat	2013-02	113.85	254.925
Cat	2013-03	443.7	317.85
Cat	2013-04	2.25	186.6
⋮	⋮	⋮	⋮

Figure 9.23

SQL-99 OLAP PARTITION versus GROUP BY. The window PARTITION statement enables you to display aggregate data (average) along with the detail rows. The GROUP BY statement only provides the summarized data. Also note the use of the PRECEDING statement in the partition to calculate across previous rows of data. This version is based on the Oracle syntax.

SQL OLAP Windows Partition

The SQL-99 standard defines some a useful extension for OLAP that should make certain types of queries substantially easier in SQL. The standard introduced the concept of **partitions** or data windows. A partition is similar to a GROUP BY clause because you specify columns whose values are used to define the partitions. But partitions offer additional options and enable you to display detail and aggregate data at the same time. Figure 9.23 demonstrates an advanced capability using the Oracle syntax, which is almost identical to the standard. SQL Server 2012 supports the same syntax but the date conversion (TO_CHAR) has to be replaced with Year(SaleDate)*100+Month(SaleDate).

The main query needs a little explanation. Its goal is to compute a moving average over time within each Category. A moving average computes the average of a specified number of rows, then moves to the next row and slides the window to the next rows. The PARTITION BY Category command specifies that the computations are to be performed for each separate value of the Category variable and reset when a new Category value is found. The ROWS 2 PRECEDING command

```
- - Create a view to get the simple monthly merchandise totals
CREATE VIEW qryMonthlyTotal AS
SELECT SaleMonth, Sum(MonthAmount) As Value
FROM qryMonthlyMerchandise
GROUP BY SaleMonth;

SELECT SaleMonth, Value,
          SUM(Value) OVER (ORDER BY SaleMonth) AS running_sum,
          SUM(Value) OVER (ORDER BY SaleMonth RANGE
                    BETWEEN UNBOUNDED PRECEDING
                    AND CURRENT ROW) AS running_sum2,
          SUM (Value) OVER (ORDER BY SaleMonth RANGE
                    BETWEEN CURRENT ROW
                    AND UNBOUNDED FOLLOWING) AS remaining_sum
FROM qryMonthlyTotal
ORDER BY SaleMonth;
```

Month	Value	Sum1	Sum2	Remain
2013-01	1358.82	1358.82	1358.82	8451.79
2013-02	1508.94	2867.76	2867.76	7092.97
2013-03	2362.68	5230.44	5230.44	5584.03
2013-04	377.55	5607.99	5607.99	3221.35
2013-05	418.50	6026.49	6026.49	2843.80
2013-06	522.45	6548.94	6548.94	2425.30
2013-07	168.30	6717.24	6717.24	1902.85
2013-08	162.70	6879.94	6879.94	1734.55
2013-09	288.90	7168.84	7168.84	1571.85
2013-10	666.00	7834.84	7834.84	1282.95
2013-11	452.25	8287.09	8287.09	616.95
2013-12	164.70	8451.79	8451.79	164.70

Figure 9.24

OVER and RANGE functions. The first SUM function computes the total from the beginning to through the current row. The second SUM function does the same thing more explicitly. The third SUM function totals the values from the current row through the remaining rows in the query.

specifies that three rows are to be included in the computation: the current row and the two rows before it. If fewer than three rows exist, the DBMS uses only the values that do exist.

One of the strengths of the OVER statement is that you can specify different partitions within the same SELECT statement. The standard also supports relatively powerful options to specify a variety of ranges of rows. It is used to perform calculations relative to the current row, so you can compute differences and averages backward and forward. Figure 9.24 shows some commonly used options for the RANGE function. The entire query computes three values of totals. The first SUM command totals the values in the rows from the beginning of the query through the current row. The second SUM column does the same thing, but more explicitly states the beginning and ending rows. The third SUM column computes the total from the current row through the last row of the query. In the second and

```
- -LAG or LEAD (Column, # rows, default)
SELECT SaleMonth, Value,
          LAG(Value, 1, 0) OVER (ORDER BY SaleMonth) AS Prior_
Month,
          LEAD(Value,1,0) OVER (ORDER BY SaleMonth) AS Next_
Month
FROM qryMonthlyTotal
ORDER BY SaleMonth
```

SaleMonth	MonthAmount	Prior_Month	Next_Month
2013-01	1358.82	0	1508.94
2013-02	1508.94	1358.82	2362.68
2013-03	2362.68	1508.94	377.55
⋮	⋮	⋮	⋮
2013-12	164.70	452.25	0

Figure 9.25

LAG and LEAD functions. As inline functions, they easily return a value from a prior or following line. You can specify how many lines to go backward or forward.

third examples, notice the use of the UNBOUNDED keyword to specify the start or end row. You could have replaced those with specific numbers if you wanted to compute only the totals for a specified number of preceding or following rows.

Most database systems also make it easy to use LAG and LEAD functions. These functions are designed to be used as inline functions that refer backward or forward to a specified number of rows. For example, the LAG function refers to values on previous rows. Figure 9.25 shows the basic syntax for Oracle and SQL Server with the result of a one-period lag and one-period lead. The power of the functions is that it is also easy to use the lag or lead variables in additional calculations. These functions are not part of the official SQL standard, so there are still some differences among the vendors. For example, you might not be able to specify the default value, which is useful for the first (or last) few rows that do not have defined values. But because most systems support the functions, and because they are so useful, they are worth studying. As of SQL Server 2012, the PARTITION, LAG, and LEAD functions are available in SQL Server with the same syntax.

Data Mining and Business Intelligence

What tools exist to search for patterns and correlations in the data? The goal of data mining is to discover unknown relationships that can be used to make better decisions. Figure 9.26 summarizes the various methods available to retrieve data from the database. Reports are predefined and generated as part of the transaction system. Queries are used to answer ad hoc questions, but require knowledge of SQL or a query builder. The OLAP cube

The topics in this section are more advanced and might be saved for a second course. The examples require installation of the Business Intelligence tools. In-depth details and explanations are provide in the separate Data Mining textbook. But this section provides an introduction to the basic concepts and goals.

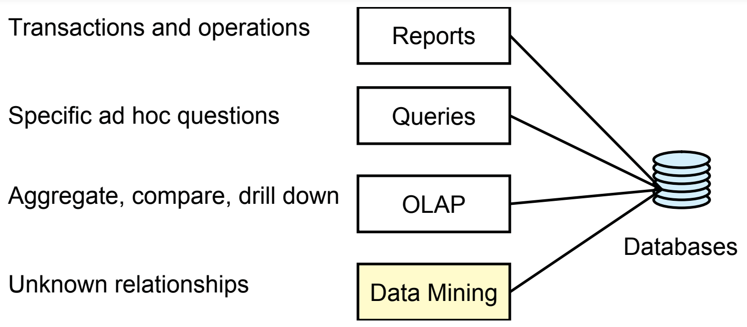

Transactions and operations
Specific ad hoc questions
Aggregate, compare, drill down
Unknown relationships
Reports
Queries
OLAP
Data Mining
Databases

Figure 9.26

Data mining. With a goal of identifying unknown relationships. Data mining is a bottom-up approach. Highly specialized tools scan the data searching for information that might be useful.

browser enables managers to retrieve data interactively, but primarily focuses on subtotals. Data mining is different in that the tools use statistical comparisons to search for patterns, and many of the tools are relatively autonomous. Managers have to select the appropriate tool and interpret the results, but the goal of data mining is to run with minimal input.

A few tools require more input and specification by model builders. Most of the technologies are exploratory, in the sense that you are searching for unknown relationships as opposed to trying to confirm a suspected one. Some of the routines are derived from statistical analysis; others are highly detailed and created for specialized tasks. This section presents an overview of some of the more popular technologies. Detailed statistical and programming issues are not covered here, but can be found in specialized textbooks.

Figure 9.27 lists some of the common data mining categories. Occasionally, a DBMS vendor will include a few of the technologies with the base system. However, most vendors sell business intelligence tools as add-on products. Many other tools are available from specialized data mining companies. In either case, you generally require the services of a modeler to help build the proper models and interpret the results. Data classification and market basket analysis are two common methods of analyzing data in business because they are useful for many types of problems. Geographic systems are powerful solutions to specific questions. Web site analysis through time-series evaluation of logs is increasingly popular. New technologies and new methodologies that can evaluate ever-larger datasets are being developed continually.

Data Configuration

Configuring data is one of the most critical tasks of analysis, and the task a database developer is most likely to focus on. The categories in this section are organized in terms of how the data needs to be organized—which is related to the ultimate task. For example, several classification tools exist, but they all rely on the same data structure.

Data for analysis can come from a data warehouse or from relational tables. Standalone tools, such as the open-source tools, often require that data be stored in CSV files. Most data warehouses and relational DBMSs can export data into

- Classification/prediction/regression
- Association rules/market basket analysis
- Clustering
  - Data points
  - Hierarchies
- Neural networks
- Deviation detection
- Sequential analysis
  - Time series events
  - Website analysis
- Spatial/geographic analysis
- Textual analysis

Figure 9.27

Data mining techniques. Classification and market basket analysis are popular technologies in business. New technologies and new methods of estimating relationships are still being developed.

CSV files. If nothing else, a SQL SELECT statement can be used to extract the data—complete with commas—and then the output can be redirected or saved to a text file.

Classification

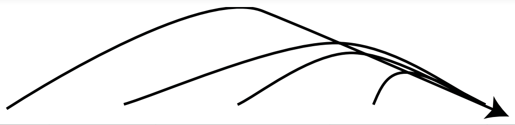
As shown in Figure 9.28, many business problems can benefit from **classification analysis**. Several tools have been developed to estimate relationships that can predict an outcome. Statistical methods like regression are readily available. However, the two drawbacks to statistical methods are that they tend to assume linear relationships exist, and the estimates are based on averages—but often the most important hidden relationships are too small to be identified by averages. For example, you might be searching for new customers that can be encouraged to return and make more purchases. Since they are new, you might not have enough average data to create a statistically important effect.

Problems that can be evaluated by classification analysis have an outcome that is affected by a set of indicator attributes. The basic objective is to estimate the strength of the effect of each indicator variable and its influence on the outcome. For instance, a bank would have historical data on borrower attributes such as job stability, credit history, and income. The data mining system could estimate the

Figure 9.28

Classification examples. Many common business problems can benefit from classification analysis. Each problem has an outcome and the goal is to classify elements into the outcome choices based on a set of attributes.

- Which borrowers/loans are most likely to be successful?
- Which customers are most likely to want a new item?
- Which companies are likely to file bankruptcy?
- Which workers are likely to quit in the next six months?
- Which startup companies are likely to succeed?
- Which tax returns are fraudulent?



Income	Married	Credit History	Job Stability	Success
50000	Yes	Good	Good	Yes
25000	Yes	Bad	Bad	No
75000	No	Good	Good	No

Figure 9.29

Bank loan classification. The indicator attributes affect the outcome in some fashion. The data mining software estimates the strength of each attribute on a set of test data. The resulting model can be applied to future data to predict the potential success or failure of new loans.

effect of each of these variables on the ultimate outcome (paying off the loan or defaulting). These weights could be applied to future customer data to help determine whether to grant a loan, or to affect the interest rate to charge.

Figure 9.29 shows a tiny sample of data for the lending situation. Note that the data might be categorical (Yes/No) or continuous (e.g., Income). Some classification tools can work with either type of data, but some require you to convert to categorical data. For example, the income data could be converted to bins, such as low: 0-30,000; medium: 30,000-70,000; high: 70,000-120,000, and wealthy: above 120,000. Of course, then you face the new data mining question of where to draw the lines to separate the categories. Some tools provide techniques to help make this decision as well.

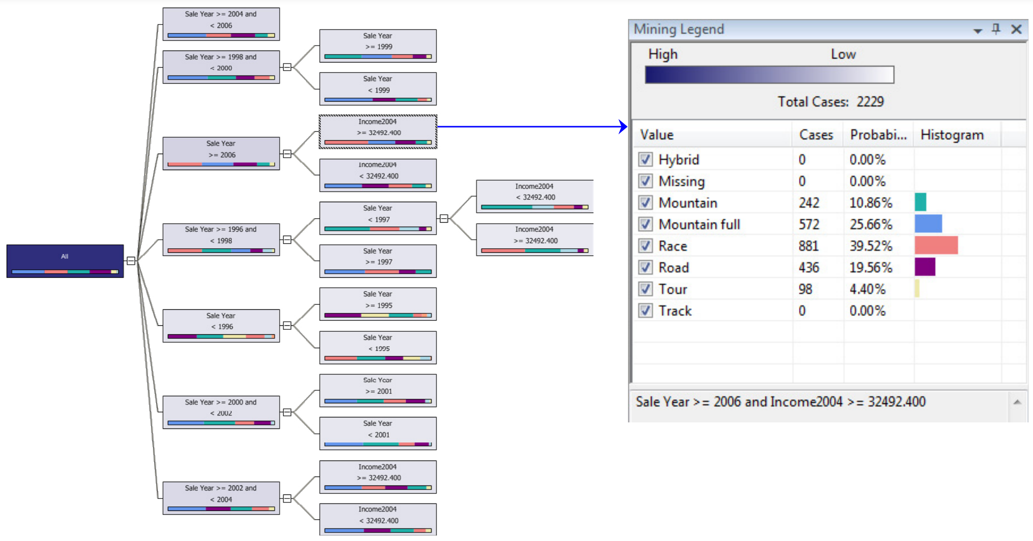
Common classification tools include: various regression methods, Bayesian analysis, decision trees (particularly for hierarchical data), genetic algorithms, and neural networks. Of the group, neural networks typically require the least supervision, whereas advanced regression techniques rely on the skills of an experienced modeler. The key issue with any classification analysis is to determine how accurately the model can predict both existing and new cases. All of the techniques have strengths and weaknesses that you need to evaluate before choose a tool for a specific problem. Most require a solid knowledge of fundamental statistics to interpret the results.

Data for Classification

Data for classification problems is typically stored similar to a relational table. Each row holds one instance of data and the columns represent the attributes. At least one attribute (column) is the predicted or dependent column, but that decision is made by the analyst or modeler. This data is usually easy to generate because a SQL SELECT command can commonly be used to choose the columns and rows and to perform simple computations.

Example

Most data mining tools require a considerable amount of data to work effectively. In most business transaction applications, you will have plenty of data. However, the sample Pet Store database is intentionally kept small to make it easier to handle. The Rolling Thunder Bicycle company database has considerably more data



Total Cases: 2229

Value	Cases	Probabi...	Histogram
✓ Hybrid	0	0.00%	
✓ Missing	0	0.00%	
✓ Mountain	242	10.86%	🟦
✓ Mountain full	572	25.66%	🟦
✓ Race	881	39.52%	🟥
✓ Road	436	19.56%	🟪
✓ Tour	98	4.40%	🟨
✓ Track	0	0.00%	

Sale Year >= 2006 and Income2004 >= 32492.400

Figure 9.30

Rolling Thunder Bicycles Model Type decision tree. Attributes Gender, SaleYear, Income, and city Population were used to predict the model type selection. Each node in the tree represents a significant change variable. The displayed legend shows percentage sales by model type for sales from 2006 with relatively higher personal incomes ($35,000 or more).

and works better for data mining illustrations. As a classification example, consider a basic goal of examining sales by model type. The goal is to see what factors affect the choice of model type, so the detail data consists of each individual bicycle sale. The fact attribute is the ModelType column. The company collects only minimal demographic data—something the managers might want to add in the future. However, Gender is available, the SaleYear is available in case purchases changed over time. Additionally, the City table contains two values from the Census Bureau: Income and Population which represent average customers within a city. A straightforward query generates the data in the correct format.

The data can be analyzed with standalone tools or a model can be built within the SQL Server Business Intelligence Studio. Every tool uses slightly different techniques and algorithms, so the results can vary slightly depending on the tool selected. The results shown in Figure 9.30 come from SQL Server's Decision Tree model. This tool examines the data to identify significant change points which are then marked as nodes in the tree. The selected node represents sales from 2006 on from cities with per capita income of about $35,000 and more. The legend for that node shows the percentage breakdown of model type sales for that group. The marketing managers can compare the values to the nearest node (incomes less than $35,000) to see the different purchase patterns.

Many other classification tools can be used. They all examine the impact of the selected attributes on the target fact variable. However, the outputs are slightly different and can provide different perspectives on the data. Some tools, such as regression, require pure numeric data; others can use categorical data.

Association Rules/Market Basket Analysis

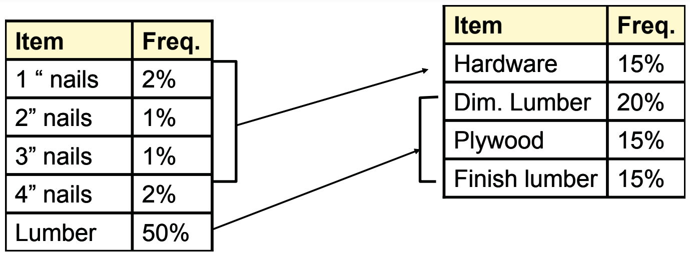
Market basket analysis is the tool that is credited with driving the acceptance of data mining. Originally, the techniques were applied to analyzing consumer purchases at convenience stores, hence the term market basket. The more generic term of **association rules** indicates that the methodology can be used for other situations. The basic question these systems answer is, What items are customers likely to buy together? Or, in terms of rules, Does the existence of A imply the existence of B? In the classic example, a convenience store discovered that shoppers who purchase diapers often purchase beer at the same time—particularly on Thursday and Friday nights. The importance of this piece of information is that managers can use it to increase sales. For instance, you might consider placing the two items close to each other in the store to encourage even more customers to purchase both items. Likewise, manufacturers might use similar knowledge to cross-sell items by providing coupons or product descriptions in the packaging of the related items.

Market basket analysis requires that you have a set of transaction data that contains a list of all items purchased by one person. Today, this data is readily available from supermarkets and large chains that use bar-code scanners. Most companies sell this data to specialized firms that resell it to other companies. The analysis software then scans the data and compares each item against the others to see if any patterns exist. In the process, the software computes three numbers that you use to evaluate the strength of the potential relationship or rule. The definitions are easier to understand with pairs of items, but they also apply to multiple items. The **support** for a rule is measured by the percent of transactions that contain both items. Statistically, the probability is denoted as $P(A \cap B)$ (the probability of A and B occurring together) and computed by counting the number of transactions with both items and dividing by the total number of transactions. Similar numbers can be computed for A and B alone, or the percentage of times each individual item has been purchased. Higher values of support indicate that both items are frequently purchased together—but the number does not tell us that one causes the other. The **confidence** of the rule (A implies B) is measured by the percentage of transactions with item A that also contain item B. Statistically, it is the probability that B is in the basket, given that A has already been chosen, denoted $P(B|A)$. By statistical definitions, $P(B|A) = P(A \cap B) / P(A)$, so it is relatively easy to compute. Again, higher values of confidence tend to indicate that purchases of item A lead to purchases of item B. The third statistic reported by most data mining tools is lift. **Lift** is the potential gain attributed to the rule, compared to purchases without the rule. If the value is greater than 1, the lift is positive. Conceptually,

Figure 9.31

Evaluating a market basket association. Support is the percentage of both items being purchased in one transaction. Confidence is the probability of purchasing beer (B) given that diapers (D) are purchased. Lift is the contribution of the effect to sales and should be greater than 1.

Support:	$P(B \cap D) = .6$	$P(D) = .7$	$P(B) = .5$
Confidence:	$P(B\|D) = P(B \cap D)/P(D) = 0.857$		
Lift:	$P(B\|D)/P(B) = 1.714$		



Item	Freq.
1 " nails	2%
2" nails	1%
3" nails	1%
4" nails	2%
Lumber	50%

Item	Freq.
Hardware	15%
Dim. Lumber	20%
Plywood	15%
Finish lumber	15%

Figure 9.32

Balanced frequencies. Items that are rarely purchased will lead to false rules. The solution is to define the items so that they balance. In this case, combine nails into a hardware category and split lumber into smaller categories.

it indicates the gain in sales resulting from the association. Statistically, it can be computed as $P(A \cap B) / (P(A) * P(B))$ or as $P(B|A)/P(B)$.

Figure 9.31 shows how the numbers are computed for the diapers and beer example. The numbers are fictional but representative of the situation. Notice that the lift is substantially higher than 1 (1.714), indicating that the association strongly contributes to sales of beer. Data mining software computes all of these numbers for essentially all pairs of items. If there are many items, the process can take quite a while to run. Also, multiple items could be considered in the analysis: Does the purchase of sheets and pillowcases lead to the sale of more towels? However, combining too many dimensions leads to huge computational issues, so most analyses are done with a limited set of comparisons.

Working with market basket analysis, you will quickly encounter several problems. First, items with a small number of purchases can result in misleading values. If an item is purchased only once or twice, then almost anything else purchased with it will seem to be related. Consequently, you will have to examine the data and change groupings to ensure that most items are purchased with approximately the same frequency. Figure 9.32 shows a hypothetical situation at a hardware store that sells a lot of lumber but only a limited number of nails and screws. To prevent spurious rules, the answer is to combine the nails and screws into a broader hardware category, and split the lumber transactions into more detailed definitions. How do you know if problems exist? You can use additional queries to quickly count the number of sales of each item. The newer OLAP functions also make it easy to compute the percentages if the raw count numbers are hard to read.

The other problems that you can encounter with market basket analysis include the fact that some rules identified will be obvious to anyone in the industry. For example, a fast food chain would undoubtedly see a relationship between burgers and fries. A tricky problem arises when the system returns rules that do not make sense or cannot be explained. For example, a hardware chain found that sales of toilet rings were closely tied to the opening of new stores. Even if this correlation is true, what do you do with it?

Data for Association Analysis

Data for association analysis generally comes from transaction systems—particularly sales data. The catch is that analysis systems use two different methods for

SaleID	ItemID	Description	Category
4	36	Leash	Dog
4	1	Dog Kennel-Small	Dog
6	20	Wood Shavings/Bedding	Mammal
6	21	Bird Cage-Medium	Bird
7	40	Litter Box-Covered	Cat
7	19	Cat Litter-10 pound	Cat
7	5	Cat Bed-Small	Cat
8	16	Dog Food-Can-Premium	Dog
8	36	Leash	Dog
8	11	Dog Food-Dry-50 pound	Dog

Transaction data. It is easy to extract with SQL on the SaleItem and Merchandise tables.

```
36, 1
20, 21
40, 19, 5
16, 36, 11
```

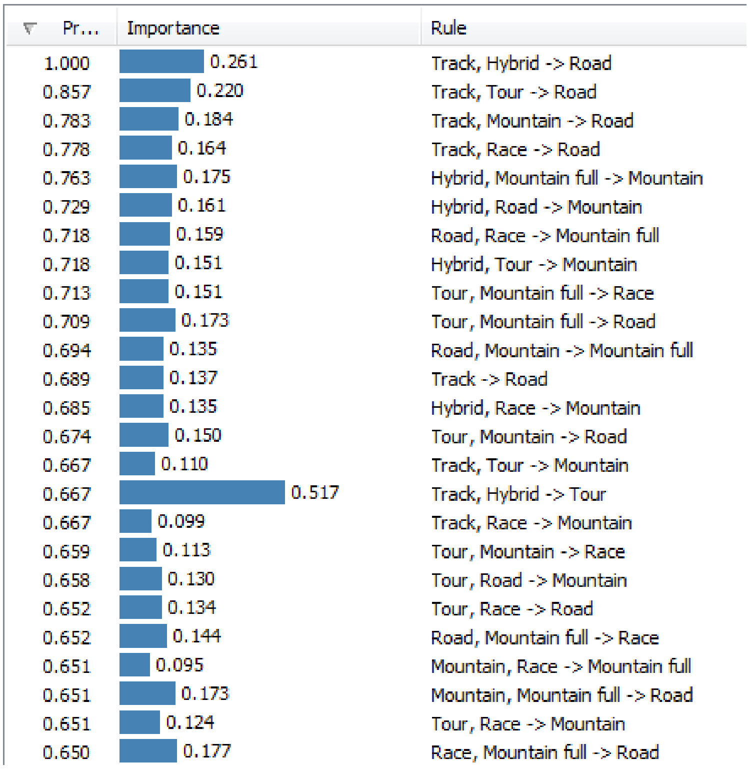
Basket data. Converting from the SaleItem table to this format requires a cursor program that builds each row as a string for each SaleID.

Figure 9.33

Two data formats for association analysis. Coming from a relational database, the top format is the easiest to create. The second format requires programming. It could be based on Category instead, but not both ItemID and category at the same time.

arranging data. You need to read the tool's documentation carefully to determine the correct layout. Figure 9.33 shows the two common layouts, labeled *transaction* and *basket*. The transaction format mirrors the relational database approach. Essentially you just need data from the SaleItem table, and perhaps the Merchandise table if the manager wants to analyze data by category instead of ItemID. The data can be retrieved easily using a standard SELECT statement.

Unfortunately, some of the early tools created for association analysis were written with the requirement that each basket be specified as one line of text, with the item values separated by commas. Each row represents one basket and the rows are variable length. There is no easy way to convert relational data into this basket format. It can be done, but it requires writing programming code that users a cursor to track through each row of data in the SaleItem table. The ItemID values are collected and built into a new string that appends a comma and the new ItemID for each row. When the SaleID switches, the new string row is written to the file. The code is straightforward, but eventually you will need a generic program that can be applied to any table or query because you will tire of rewriting the code every time it is needed.



Pr...	Importance	Rule
1.000	0.261	Track, Hybrid -> Road
0.857	0.220	Track, Tour -> Road
0.783	0.184	Track, Mountain -> Road
0.778	0.164	Track, Race -> Road
0.763	0.175	Hybrid, Mountain full -> Mountain
0.729	0.161	Hybrid, Road -> Mountain
0.718	0.159	Road, Race -> Mountain full
0.718	0.151	Hybrid, Tour -> Mountain
0.713	0.151	Tour, Mountain full -> Race
0.709	0.173	Tour, Mountain full -> Road
0.694	0.135	Road, Mountain -> Mountain full
0.689	0.137	Track -> Road
0.685	0.135	Hybrid, Race -> Mountain
0.674	0.150	Tour, Mountain -> Road
0.667	0.110	Track, Tour -> Mountain
0.667	0.517	Track, Hybrid -> Tour
0.667	0.099	Track, Race -> Mountain
0.659	0.113	Tour, Mountain -> Race
0.658	0.130	Tour, Road -> Mountain
0.652	0.134	Tour, Race -> Road
0.652	0.144	Road, Mountain full -> Race
0.651	0.095	Mountain, Race -> Mountain full
0.651	0.173	Mountain, Mountain full -> Road
0.651	0.124	Tour, Race -> Mountain
0.650	0.177	Race, Mountain full -> Road

Figure 9.34

Association rules for bicycles purchased by each customer. Same customer, possibly different times. Microsoft's probability and importance calculations are non-traditional but the interpretation is the same. Rules with high probability and high importance are likely to repeat.

Example

It can be fun to experiment with market basket analysis. In some cases, it is obvious which items are purchased together (burgers and fries), in other cases the results are surprising. It is the surprising results that are the most useful. Still, association analysis can generate hundreds or even thousands of rules. It takes time and some experience to read through the rules and find the ones that can be useful. Rolling Thunder Bicycles has a couple of possibilities for using association analysis. It might be tempting to look at the traditional market basket and see which items were purchased at the same time. However, remember that almost all bicycles are built using groups of components. A group specifies all of the default components, so a market basket analysis would simply identify all of the components within a group. But, we already know those values, so there is no surprise. If customers routinely overrode the defaults and selected their own components, the results would be more interesting.

Instead, Figure 9.34 shows the results of examining model type purchases by customer. Essentially, CustomerID is the market basket and model types are the items purchased. Customers can buy multiple bicycles, perhaps at different times.

The question being asked is whether there is a pattern in purchases of model types. Consider the first rule which has a high probability (1.00) and relatively high importance (0.261). It says the customers who purchased a track and a hybrid bicycle also purchased a road bike. Consequently, future customers who purchased the first two model types should be contacted to suggest that they might also want to buy a road bike. Alternatively, the company could offer discounts on hybrid or track bikes which might then increase the sales of road bikes—which would not be discounted. Only some of the rules are shown in the figure. The challenge with association rules is to find the ones that are strong, important, and meaningful.

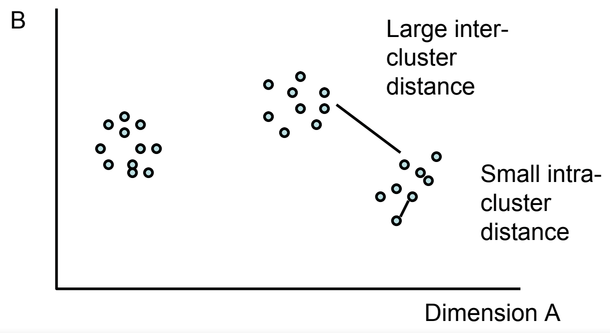
Cluster Analysis

Cluster analysis is used to identify groupings of data—data points that tend to be related to each other. It can be used to identify groups of people, for example, to categorize customers. If you know that customers fall within certain groupings, you can use the information about a few customers to help sell additional products to the others in the group. Most likely, customers in the same group will want similar products. For instance, a bookstore can use the purchases of some items to categorize a customer and then identify books that similar customers bought and suggest them to the other shoppers. Likewise, you could use cluster analysis to categorize the skills of employees that work in various departments and use that information when hiring new workers.

As shown in Figure 9.35, clusters are relatively easy to see in two dimensions. The objective of the software is to identify the data points that are close to each other (small intra-cluster distance), yet further away from other points (larger inter-cluster distance). Unfortunately, most datasets do not exhibit clustering as strongly as shown in this example. But cluster analysis is a useful data exploration technique because it can reveal patterns that you might not see with other tools. However, keep in mind that datasets with a large number of observations (rows) and many dimensions are extremely difficult to cluster. Even with relatively modern computers, it can take hours or days to evaluate large, complex problems. So start cautiously and try to build clusters using smaller samples and a limited number of dimensions.

Figure 9.35

Cluster analysis. The goal is to find data points that are grouped close to each other and farther from other groups. Larger datasets with multiple dimensions are difficult and time-consuming to evaluate.



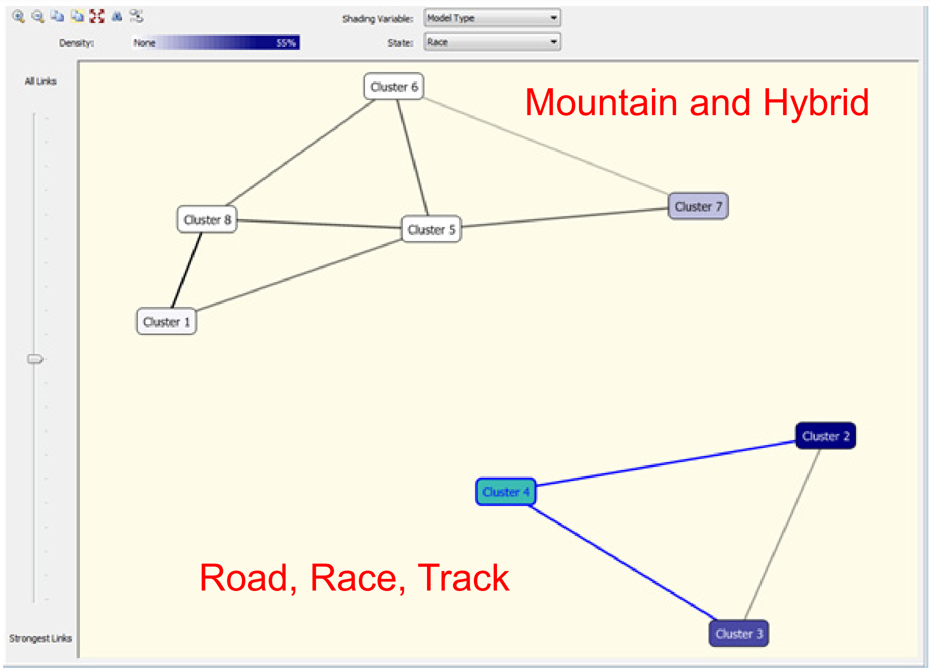


Figure 9.36

Cluster based on bicycle attributes. This chart focuses on model type. Based on the shading, the two main clusters are split by road/race bikes versus mountain/hybrid types. Details within the two groups are based on sale price and order year. Bike size also plays a role in differentiating the clusters.

Data for Cluster Analysis

Data for cluster analysis is straightforward because it is similar to relational tables. Each column defines values for a chosen attribute. Each row represents one instance of the data. If the query results contain two attributes (columns), then each row represents one point on the two-dimensional chart. This data is easily retrieved using a standard SELECT query.

The one catch with cluster analysis is that some versions will not run if the dataset is too large. Too large is defined both in terms of the number of dimensions (columns) and the number of observations (rows). The specific limits depend on the algorithm used by the tool and the processing speed of the computer. This problem is similar to the issue of dimensionality in association analysis. You might need to reduce the number of dimensions, or combine items into aggregates. For example, it might make sense to examine sales of categories instead of individual items. Ultimately, this decision must be made by the manager or statistical analyst. However, it is often wise to start with smaller problems using aggregated data. Once these work, you can begin disaggregating the data and looking at larger problems.

Example

Rolling Thunder Bicycle Company presents several opportunities for cluster analysis. Aligning with the other examples, this example builds cluster based on model type. It is possible to include multiple attributes for each point so the data examines each bicycle in detail: Construction type (which is a proxy for material used), order year, sale price, bike size, and time to build. The attributes selected depend on the goals of the analysis. You might want to start with a smaller number of attributes—partly because including too many dimensions makes the model more difficult and time-consuming to estimate.

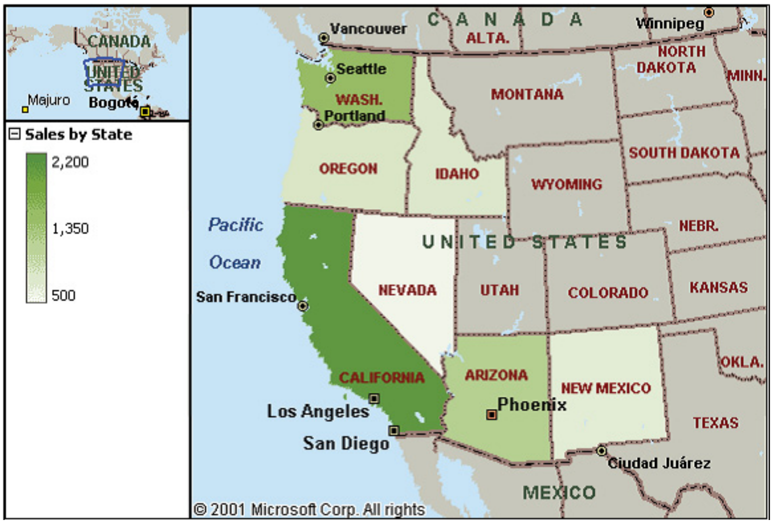
Figure 9.36 shows one version of the cluster results. Note the two large groupings—these are largely determined by model type. The top grouping consists of mountain and hybrid bikes. The lower group consists of road and race bikes. The tools provide additional charts to enable you to determine the differences between the clusters within the groups. These charts are not shown here, but they indicate that the details are determined by order year, sale price, and bike size. If managers want to examine these effects in more detail, it would make sense to run additional cluster analyses focusing on two or three of these attributes at one time. Ultimately, managers will want to see results from a variety of different models. For example, clusters might generate some intuition about the data, which could then be analyzed with classification tools.

Geographic Analysis

Geographic information systems (GIS) display data in relation to its location. The systems are generally classified as visualization systems. They are useful for displaying geographical relationships and showing people how data is influenced by location. Few systems have true data mining capabilities for scanning the data to find patterns. Nonetheless, they are an important tool in analyzing data. Some relationships are much easier to understand if you see them on a map. Figure 9.37

Figure 9.37

Geographic analysis. This basic map shows sales by state. As shown by the key, darker colors represent larger sales. Additional data, such as income, could be shown as overlays or compared in charts.



shows a simple map of sales by western states. Additional data could be displayed with more colors or charts could be placed on each state.

Larger DBMS vendors have begun incorporating spatial and GIS systems into their offerings. You can also purchase standalone systems from other vendors. Beyond drawing maps, a true GIS has several methods for displaying data on the map. Basic techniques include shading and overlays, often used to display sales by region. Overlays show multiple items on different levels, making it easier to see how several items relate to each other as well as to location. For instance, marketers might compare sales, income, and population by geographic region.

In addition to the software, you need two important components for geographic systems. First, you need map data. Generally, this data is sold with the analysis system, but detailed data is sometimes sold as an add-on option. Highly detailed data down to individual street level is available for the United States (and much of Europe), but it is a large database. Second, you need to **geocode** your data and probably buy additional demographic data that is already geocoded. Essentially, you need to collect and store some type of geographical tag for your data. At a basic level, you probably already know country and state. But you might also want to add a region code, or a city code, or perhaps even latitude and longitude. If all of your sales are through individual stores, it is relatively easy to get the geographic position of each store from maps or GPS systems. An interesting possible option in the future arises from the increasing use of cell phones. Because of federal emergency regulations (e-911), cell phones are required to have positioning systems. Eventually, it is conceivable that this information will be provided to businesses, so your transaction systems can record exact locations of salespeople, and possibly even of customers. Please keep in mind the serious privacy issues these technologies create, but as you build new databases, you should think about incorporating geocode information into the data capture tables. Once the data has been collected, the GIS makes it easy to display relationships.

Data for Geographic Analysis

Most GIS tools are standalone tools. For example, Microsoft's MapPoint is integrated into Excel. On a larger scale, ESRI's ArcGIS is definitely a standalone (or Web based) tool. Similarly, Google Earth is largely Web based. Most of these

Figure 9.38

Common geographic identifiers. At least one of these attributes must be coded into the data to use a GIS.

State
Country
Region (custom defined)
Latitude, Longitude
Address
City
County
ZIP Code
Census Tract
Standard Metropolitan Statistical Area

tools can extract data from a DBMS and use it in their displays and analyses, so data preparation can still be handled within the DBMS.

GIS data is typically stored in a relational format. Each column represents a single attribute and each row contains values for one location. The critical point is that at least one column must contain a geographic identifier. For example, a query might compute sales by state, so one column contains the state code. Each tool supports different types of geographic codes, but Figure 9.38 shows the types of geographic identifiers support by most systems. Some of the items in the list are defined within the U.S. only; however, there are often similar values in other nations. For instance, Postal Code is an international version of the U.S. ZIP Code.

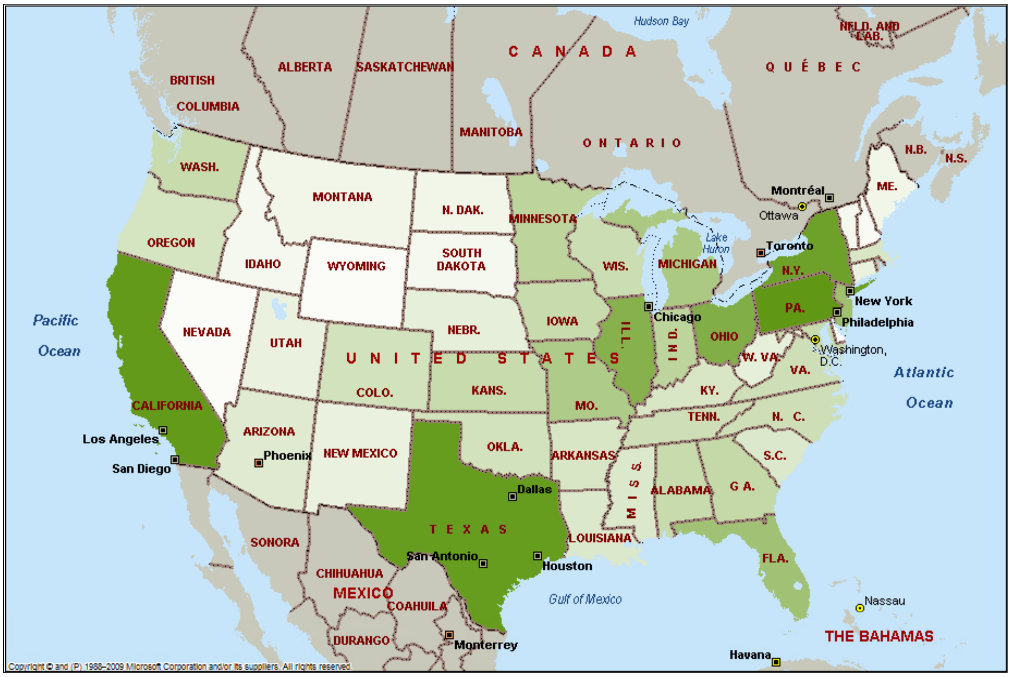
At least in the U.S., some national data is already coded geographically. In particular, data collected by the Census Bureau is tagged by several identifiers such as City, State, ZIP Code, Census Tract, and Standard Metropolitan Statistical Area (SMSA) or large city region. Some tools include access to common Census data, but much of the data is available for free download from the Census Bureau Web site. This data is useful for comparisons or overlays with your business data. In particular, economic models suggest that it is useful to compare average income to sales.

Example

GIS systems are different from most other data mining tools. You need a specific tool to be able to plot data geographically, and these tools are almost always standalone tools. Consequently, you generally export the data from the database.

Figure 9.39

Sales of bicycles by state in 2009. The legend is hidden but states colored in darker green represent higher sales based on dollar value.



A relatively inexpensive tool is Microsoft's Map Point software. You also might be able to use online tools such as Google Earth—but even some of those carry fees if you want to add your own data. Some applications can be handled online, such as placing stick pins or drawing routes in Google Maps or Microsoft Maps. Shading regions or states based on sales data has usually more difficult with the online tools.

As a small example of Microsoft Map Point, it is straightforward to write a query to retrieve sales value by state for 2009. Running the query, the results can be copied and pasted into an Excel spreadsheet. Once Map Point is installed, it can be run as an add-in. The tool automatically picks up the state codes (although it does not recognize PR for Puerto Rico). Figure 9.39 shows the data plotted using darker colors for higher sales. It is also straightforward to insert push pins, sized dots, or data charts.

Summary

Large databases are optimized for transactions processing—to handle day-to-day operations efficiently, data is stored in normalized tables. But most managers need to join several tables to retrieve and understand the data. Indexes speed joins and data retrieval, but slow down transactions. This dichotomy means that it is often better to create a separate data warehouse to use for data analysis. Data can be extracted and cleaned from transaction systems, and placed into star or snowflake designs enabling managers to focus on the dimensions that surround a particular fact.

OLAP cubes are a powerful tool to enable managers to quickly sift through data and examine subtotals from a variety of perspectives. Without writing intense SQL queries, managers can compare values across product categories, time, and even across multiple dimensions simultaneously. OLAP cube browsers also contain easy methods to filter the data to specific rows or cube sections.

Many statistical data mining tools have been developed to help managers analyze data. They often require training and specialized knowledge by the workers, but can be powerful tools to understand relationships among the data. Classification and clustering algorithms help break the data into groups. Comparing the various groups makes it possible to better understand customers and expand the market. Association or market basket rules are popular with stores that sell a large variety of items. Identifying items that are purchased together makes it possible to suggest products to other customers. It can also lead to insights in store layout and customer psychology. Geographic systems are useful for any problem involving location. Specialized tools and demographic data are available to see the geographic relationships that exist.

A Developer's View

Miranda saw that some business questions are difficult to answer, even with SQL. When managers are not exactly sure what they are looking for, you need to consider the OLAP and data mining approaches. Providing an OLAP cube is a good first step because it makes it easy for managers to see subtotals and slice the data to whatever level they want. More sophisticated statistical data mining tools are available, but generally require additional training and knowledgeable users. Just remember that performance often requires moving OLAP data into a separate data warehouse.

Key Terms

association rules
binary search
business intelligence (BI)
classification analysis
cluster analysis
comma separated values (CSV)
confidence
data hierarchy
data mining
data warehouse
DENSE_RANK
dimensions
drill down
extraction, transformation, and transportation (ETT)
fact table
geocode
geographic information systems (GIS)
lift
market basket analysis
measures
multidimensional expressions (MDX)
online analytical processing (OLAP)
online transaction processing (OLTP)
partition, SQL
pointer
RANK
roll up
snowflake design
star design
super-aggregate
support

Review Questions

1. ✓ Why are indexes so important in relational databases?
2. Given the power of a relational DBMS, why might a company still need a data warehouse?
3. What main problems are encountered in setting up a data warehouse?
4. ✓ How are OLAP queries different from traditional SQL queries?
5. What is an OLAP cube?
6. What are hierarchical dimensions and how do they relate to roll up and drill down operations?
7. ✓ What basic analytical functions are defined in SQL?
8. What is the goal of data mining?
9. What are the main categories of data mining tools?
10. How is data organized in a data warehouse?

Exercises

1. Find at least two commercial OLAP tools and compare the features.
2. Find a commercial data mining tool and outline the steps needed to extract and transform data from a typical DBMS so it is usable by the system.
3. Find a commercial data mining tool and outline the steps needed to perform a market basket analysis.
4. Assume you have two separate sets (tables) of customer data. You need to merge the two sets and eliminate the duplicates. The two tables use different ID/key values. Describe any problems you expect to encounter and how you might resolve them.
5. This question requires some tricky SQL. Assume you have a query (AnnualSales) with columns for SaleYear and Sales. Write the plain SQL (without the LAG function) to compute the difference in sales (current year value – prior year).

Most of the following questions require an OLAP cube processor. You should have access to SQL, an OLAP browser within the DBMS, or a PivotTable. For the data mining tools, if you do not have access to specialized software, you can use Excel for simple analyses.

Sally's Pet Store

6. Create a cube to browse merchandise sales by date, state, employee, and item category.
7. Create a cube to browse animal adoptions by time, category, breed, gender, and registration.
8. If you are using SQL Server or Oracle, write the grouping and cube query to compute sales by employee by month, similar to the query in Figure 9.21.
9. Create a cube to browse purchases of merchandise from suppliers based on time, employee, and location. As facts, include the value of the purchase, the shipping cost, and the delay between order and receipt.
10. If you have access to market basket software, evaluate the sales tables to see if any associations exist. If you do not have the software, set up the query to retrieve the data.
11. Using SQL Server or Oracle, create a view to compute sales by month (YearMonth). Use the Lag function to compute the percentage change from the previous month.
12. Using SQL Server or Oracle, create a view as in the previous question that computes the sales by month. Then use the AVG and OVER functions to compute the three-month moving average.
13. Using SQL Server or Oracle, create a view that computes the total merchandise purchases by month. Then create a query that displays the month, total, and running total to date.

14. Identify at least two specific data mining tools that would be useful for this company and explain what data would be used and how they might be used to improve sales or operations.
15. Using monthly sales of merchandise, forecast sales for the next three months.
16. Is there a geographic pattern to sales? Do some states or regions have more sales?
17. Compute the total sales by employee for the year and list them in descending order with the computed ranking, similar to Figure 9.22.
18. Use the bulk load or import facilities of your DBMS to load several new items into the Merchandise table. File: MerchandiseNew.csv. Hint: Import the CSV file into a new table and use an INSERT statement to move the data into the Merchandise table.
19. Import the CSV file NewCustomers.csv into the database as a new, temporary table. Add the customers to the Customer table but be careful. Some of the "new" customers already exist in the Customer table—do not add duplicate values.
20. Import two CSV files (NewSales.csv and NewSaleItems.csv). The Sale file has a SaleID and a CustomerID. The CustomerID is valid, but the SaleID values are temporary can cannot be used in the main database. The NewSaleItems file has the matching SaleID and an ItemID. The ItemID is valid, and the SaleID matches the temporary value in the matching NewSales file. Import both files into the database, insert the new sales into the main Sales table, generating a new SaleID value. Assign that new SaleID value to insert the NewSaleItems entries into the SaleItem table.

Rolling Thunder Bicycles

21. Create an OLAP cube to evaluate sales (value and quantity) by model type, state, time, and sales employee.
22. Create an OLAP cube to evaluate production time (ShipDate – OrderDate) by order date (time), model type, month, and employee who assembled the frame.
23. Create an OLAP cube to evaluate purchases of components by time, manufacturer, road or mountain bike, and component category.
24. Run a regression analysis to determine how sales by city by year are affected by population and income.
25. Using monthly sales by model type, forecast sales for the next six months.
26. Write a query to retrieve the data to perform a market basket analysis of component sales—to test which components were installed on the same bike.
27. Create an OLAP cube to evaluate sales (quantity) by paint type, letter style, and model type.
28. Create a query that computes total sales by year. Create another query that displays those annual values and computes the percentage change from year to year. Hint: Define a new column as PriorYear = Year-1 and use it in a join.

29. Using SQL Server or Oracle, create query that computes total sales by Year and Model Type and compute and show only the super-aggregate totals for model type and year.
30. Using SQL Server or Oracle, create a query that displays Year, Month (year/month), Sales, and year-to-date sales using the SQL Analytic functions. Hint: The syntax is slightly easier if you first create a view to compute MonthlySales (Year, YearMonth, Sales).
31. Using SQL Server or Oracle, if it does not already exist, create a View that computes total sales by YearMonth. Using the SQL Analytic functions create a query to compute a 3-month moving average by model type. Hint: Leave out the Hybrid and Track model types because of their limited sales.
32. If you have access to a GIS such as Microsoft MapPoint, write the query and import the data to display a map similar to Figure 9.39 showing sales of Race bikes in 2012.

Corner Med

33. Use association software or computations to see if some diagnoses commonly arise together.
34. Assume the ICD10 conversion does not exist. Use the crosswalk tables to identify the matching values for the existing ICD9 codes in the VisitDiagnoses table. Comment on any problems you find.
35. Using categorization software, such as regression, neural network, or decision tree, try to identify features of patients that spend the most money.
36. Create an OLAP cube to explore physician data in terms of patients and procedures. Managers want to focus on revenue and patients visited per day, week, and month.
37. Forecast the number of patients expected for a specific month. Hint: Use simple regression unless you have access to a time series analyzer.
38. Using SQL Server or Oracle SQL Analytic functions, show the monthly revenue generated by procedures by each of the physicians, along with the super-aggregate totals. Including the Grouping values.
39. Using SQL Server or Oracle SQL Analytic functions, count the number of patient visits per day for the month of March, and show the running total for the month.
40. Create a view that computes Revenue by month. Using either the Lag function or a JOIN by Year – 1, compute the percentage change in revenue by month.

Web Site References

http://www.oracle.com/technology/tech/bi/index.html	Oracle business intelligence tools
http://www.microsoft.com/en-us/bi/default.aspx	Microsoft SQL Server analysis tools
http://www-03.ibm.com/software/products/us/en/category/SWQ20	IBM DB2 business intelligence tools
http://publib.boulder.ibm.com/infocenter/rbhelp/v6r3/index.jsp?topic=%2Fcom.ibm.redbrick.doc6.3%2Fsqlrg%2Fsqlrg36.htm	SQL 99 OLAP standards and example.

Additional Reading

Apte, C., B. Liu, E. Pednault, and P. Smyth, Business applications of data mining, *Communications of the ACM*, 45(8) August 2002, 49-53. [Some examples of data mining, also part of a special issue on data mining.]

Golfarelli, M, and S. Rizzi, A Methodological Framework for Data Warehouse Design, *Proceedings of the first ACM international workshop on Data warehousing and OLAP*, 1998, ACM Press, 3-9. [Relatively formal definition of facts, dimensions, and hierarchies.]

Han, J. and M. Kamber, *Data Mining: Concepts and Techniques*, San Francisco: Morgan Kaufmann/Academic Press, 2001. [A general introduction to data mining techniques.]

Hastie, T., R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning/2e*, New York: Springer-Verlag, 2009. [A strong foundation book on the statistics and algorithms of data mining including all of the math.]

Peterson, T., J. Pinkelman, and B. Pfeiff, *Microsoft OLAP Unleashed*, Indianapolis: Sams/Macmillan, 1999. [Details on OLAP queries and data warehouses in SQL Server.]

Post, Gerald, *Data Mining Applications/2e*, 2012, http://www.JerryPost.com/Books/DMBook. [Detailed applications of data mining with SQL Server and some open-source tools.]

Scott, J. Warehousing over the Web, *Communications of the ACM*, 41(9), September 1998, 64-65. [Brief comments on Comcast using a Web interface for its data warehouse.]

Part 4

Database Administration

Large applications require careful support. Most organizations hire a database administrator to monitor application performance, assess security, and ensure database integrity. Chapter 10 highlights the tasks of the data administrator and the database administrator—with special emphasis on database security. Once again, SQL has a strong role in managing and protecting the database.

Information systems (IS) managers are increasingly concerned with issues of providing access to data regardless of location. Networks and the Internet provide multiple options for distributing data and providing answers throughout the organization. Chapter 11 explores some of the challenges and options of distributed databases.

Chapter 12 introduces the technologies that a DBMS uses to physically store the data tables on a typical operating system. Database administrators need to understand the basic concepts so they can select the proper storage mechanisms for high performance or large databases. Students with a computer science or programming background will recognize the importance of the underlying data structures.

Chapter 13 introduces the reasons for the interest in the newer non-relational or non-SQL databases. The big target is Web-based applications requiring high performance to handle data for millions of users—inexpensively. Performance is critical because database design is optimized for specific operations and queries. The tools support only simple queries without JOINs or complex conditions.

Chapter 10

Database Administration

Chapter Outline

What You Will Learn in This Chapter

- What administrative tasks need to be performed with a database application?
- How do you ensure data is consistent across multiple databases?
- What are the basic tasks of a database administrator?
- How does a DBMS support multiple databases?
- How does a DBA find out what is stored in each database?
- What DBA tasks need to be performed as an application is developed?
- How do you back up data that is constantly changing?
- How should computers be configured for DBMS software and database files?
- What security techniques are used to protect databases?
- How do you prevent eavesdroppers or hackers from reading data?
- What security conditions would be needed at Sally's Pet Store?

A Developer's View

Miranda: Finally, everything seems to be running well.

Ariel: Does that mean you finally got paid?

Miranda: Yes. They gave me the check yesterday. They even liked my work so well, they offered me a job.

Ariel: That's great. Are you going to take it? What job is it?

Miranda: I think so. They want me to be a database administrator. They said they need me to keep the database running properly. They also hinted that they want me to help their existing programmers learn to build database applications.

Ariel: Wow! That means you'll get more money than the programmers.

Miranda: Probably. But I'll have to learn some new material. I'm really starting to worry about security. The accounting manager talked to me yesterday and gave me some idea of the problems that I can expect with the sales application.

Getting Started

Someone has to perform several administrative tasks to keep a database running. The DBMS software has to be installed, monitored, and updated. Databases have to be backed up. Security permissions have to be assigned, tested, and revised. Applications and queries need to be optimized.

Introduction

What administrative tasks need to be performed with a database application? The power of a DBMS comes from its ability to share data. Data can be shared across many users, departments, and applications. Most organizations build more than one application and more than one database. Large organizations might use more than one DBMS. Most companies have several projects being developed or revised at the same time by different teams. Imagine what happens if you just turn developers loose to create databases, tables, and applications anyway they want to. It is highly unlikely the applications would work together. Just using a DBMS is not enough. An organization that wants to build integrated applications must have someone in charge of the data and the databases.

Data administration consists of the planning and coordination required to define data consistently throughout the company. Some person or group should have the responsibility for determining what data should be collected, how it should be stored, and promoting ways in which it can be used. This person or group is responsible for the integrity of the data.

Database administration consists of technical aspects of creating and running the database. The basic tasks are performance monitoring, backup and recovery, and assigning and controlling security. Database administrators are trained in the details of installing, configuring, and operating the DBMS. Smaller organizations might have a single person responsible for the data administration and DBA roles. Larger companies tend to have multiple DBAs, but only one or two data administrators.

Computer security is increasingly important to organizations. Large organizations usually have a head computer security officer responsible for setting policies, identifying major threats, monitoring compliance, and organizing responses to threats. However, security topics need to be studied by all application developers and database managers. Security is not a separate set of topics that can be added on at the end of a project. Security issues need to be integrated into the design and development process. Some of the more critical issues have been mentioned in earlier chapters. In particular, the SQL injection attack is best solved by developers adding code to test all input values. Likewise, developers can use selection boxes and radio buttons to limit user input. They can also create menu options that are visible only to selected groups of users.

Database security is a subset of computer security topics, and it is important to build security at multiple levels. Some of these levels are best handled centrally by the DBA. If database security is assigned properly, it has the ability to reduce many types of fraud. If database security is ignored or performed poorly, major assets of the company could be manipulated or stolen from any computer in the world. It pays to understand the security issues and to handle security properly. This chapter presents an introduction to DBA and security tasks. Both of these roles require considerable additional learning and practice. This chapter focuses on the general tasks that need to be performed with any DBMS, but many DBA tasks rely on specific features or a particular DBMS. If you choose to become a DBA, you will have to study a particular system in detail.

Two-Minute Chapter

Database systems are powerful tools and they easily handle several complex tasks. But because data is so critical, someone has to monitor and manage the databases. Database design and consistency are critical to being able to integrate data. But performance, security, and backup and recovery are also critical operations to ensure the long-term value of the data. A database administrator (DBA) is in charge of keeping the database system running, planning upgrades, and monitoring performance. The larger systems including SQL Server, Oracle, and DB2 are complex tools with many internal controls and options that can be configured. It takes several months of study and practical experience to become a good DBA.

Computer security concepts are important in database design and operations. Typically, roles are assigned to cover specific tasks, such as an Order Entry Clerk which needs the ability to lookup Customer and Product data and create new Orders. Permissions to access tables have to be given to each role. Then the roles are assigned to individual employees or groups of employees. The permissions have to be thoroughly tested so that the individuals have enough access to complete their tasks.

Physical security has to be established through physical locks, fire safety, and other standard precautions. Backup facilities are critical to any company. If time is critical, companies will run duplicate data centers and share the workload and data. If one center fails, the other can immediately pick up the load and later be expanded to handle more of the operations. Cloud-based systems are useful for backup because they can be expanded or contracted without incurring huge fixed costs.

Some data needs to be encrypted both for data transmission and for safety in storing the data. Common examples include credit card numbers and taxpayer IDs. Encryption tools are built into the higher-end systems but they usually require configuration. An important step is securing the encryption keys so that even if someone steals the database, they will not be able to decrypt the data.

Data Administrator

How do you ensure data is consistent across multiple databases? Data is an important asset to companies. Think about how long a modern company would survive if its computers were suddenly destroyed or all the data lost. Some organizations might survive as long as a few days or a week. Many, like banks, would be out of business immediately. A company should not have to lose any data before it recognizes the value of the information contained in the data. As indicated by Figure 10.1, companies have many databases for different purposes. Over time, organizations build different databases and applications to support decisions for operations, tactics, and strategies. Each application is important by itself, but when the applications and databases can coordinate and exchange data, managers receive a complete picture of the entire organization.

Despite the power and flexibility of database systems, applications built at different times by different people do not automatically share data. The key to integrating data is to put someone in charge of the data resources of the company. In most companies the **data administrator (DA)** fills this position.

As summarized in Figure 10.2, the primary role of the DA is to provide centralized control over the data for the entire organization. The DA sets data definition standards to ensure that all applications use consistent formats and naming conventions. The DA coordinates applications and teams to ensure that data from individual projects can be integrated into a corporatewide information system. If disputes occur among developers or managers, the DA serves as the judge, mak-

Figure 10.1

Data administration. With many projects and developers, a data administrator coordinates the projects so data can be integrated across applications.

- Provide centralized control over the data.
  - Data definition: format and naming convention.
  - Data integration.
  - DBMS selection.
- Act as data and database advocate.
  - Application ideas.
  - Decision support.
  - Strategic uses.
- Coordinate data integrity, security, privacy, and control.

Figure 10.2

Data administrator roles. The DA is responsible for maintaining the quality of the data and for integrating data across the organization. The DA also advocates the use of databases and is often in charge of security.

ing decisions to ensure compatibility across the organization. The DA also monitors the database industry and watches trends and technologies to advise the company on which database systems and tools to consider for long-term benefits.

The DA plays a crucial role as an advocate. Most managers and many developers are not aware of the power and capabilities of modern database systems. By understanding the managerial tasks and the database capabilities, the DA is in a position to suggest new applications and expanded uses of the existing data.

Ultimately, the DA is also responsible for the integrity of the data: Does the data contained in the DBMS represent a true picture of the firm? Does the firm have the proper systems and controls in place to ensure the accuracy and timeliness of the data?

The DA position is largely a management job. The DA tasks consist of organizing and controlling the design aspects of application development. Control is maintained by setting standards, monitoring ongoing development and changes, and providing assistance in database design as needed. The DA also spends time with business managers to evaluate current systems, monitor business trends, and identify future needs. The person hired for this position usually has several years of experience in designing databases and needs a detailed knowledge of the company. The DA also needs technical database skills to understand the various storage implications of the decisions. The DA must also be able to communicate easily with technical managers and business managers.

Database Administrator

What are the basic tasks of a database administrator? A DBMS is a complex software package. Installing, running, and upgrading a DBMS are not trivial tasks. Even with personal computer-based systems, these tasks can require the services of a full-time person. Every database requires the services of a **database administrator (DBA)**. The DBA position is generally staffed by a specialist who is trained in the administration of a particular DBMS. In smaller companies, instead of hiring a specialist, one of the lead developers may be asked to perform DBA duties.

The DBA role is relatively technical. As highlighted in Figure 10.3, the DBA's responsibilities include installing and upgrading the DBMS. Additional tasks include creating user accounts and monitoring security. The DBA is also responsible for managing backups. Although the actual backup task may be performed by a

Install and upgrade DBMS.
Create user accounts and monitor security.
Backup and recovery of the database.
Monitor and tune the database performance.
Coordinate with DBMS vendor and plan for changes.
Maintain DBMS-specific information for developers.

Figure 10.3

Database administrator roles. The DBA tasks are fairly technical and require daily monitoring and changes to the DBMS.

system operator, the DBA is responsible for setting schedules and making sure the data backups are safe. The DBA also monitors the performance of the databases and plans upgrades and additional capacity. The DBA must stay in contact with the DBMS vendor to track system problems and to be notified of changes. As new utilities, tools, or information are provided, the DBA functions as a liaison to gather this knowledge and make it available to developers. The DBA has complete access to the data in the application. In many organizations the DBA is in charge of security for each database. Larger companies might appoint a special security officer to specify policies and procedures and to help with the monitoring. However, the DBA is generally in charge of carrying out the technical details of assigning security privileges for the database.

Data allocation and storage are an important part of the daily tasks of the DBA. Some large database systems require the DBA to preassign a space on the disk drive for each database. Many systems allocate physical space by creating datafiles and **tablespaces**, which are logical collections of space where data can be stored.

Separate space is usually allocated for the data tables, the indexes, and the transaction logs, and the DBA must estimate the size of each component. If the DBA allocates too little space, performance will suffer; on the other hand, allocating too much space means that the company will waste money on unneeded disk drive capacity. Most systems provide tools to add space later, but it is best to get good estimates up front. The data volume estimates from Chapter 3 provide crucial information in determining the space requirements. For tables, the main concept is to determine the size of an average row (in bytes) and multiply by the expected number of rows in the table. Note that each DBMS stores data slightly differently and some add bytes per row of storage. The documentation will provide details for each DBMS. A more accurate solution is to set up a temporary database, create a few rows of data in each table, and then use the actual average space to estimate future needs. Space required for the indexes and rollback log depend on the specific DBMS and the computer system. If you need highly accurate estimates, you will have to consult the documentation and support tools for your specific DBMS. Space for indexes and logs also depends on the number and length of transactions defined in the applications. For example, the transaction log in a database used for transaction processing will have to be substantially larger than the log in a database used primarily for decision support and data retrieval. The larger DBMSs provide tools to help estimate and monitor storage space.

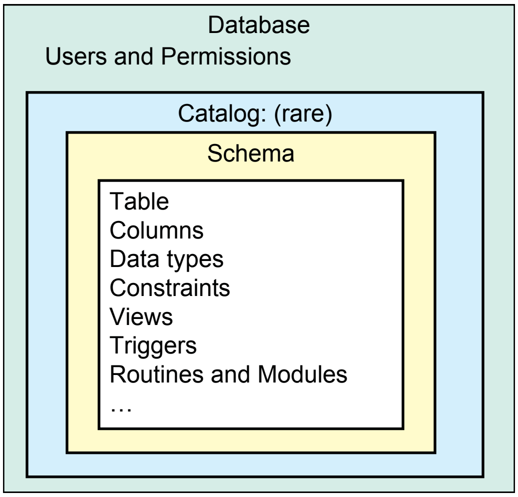


Figure 10.4

Database structure. The schema serves as a container for other elements to minimize potential naming conflicts.

Database Structure

How does a DBMS support multiple databases? The DBA works on a daily basis with the structure of the database. Although each DBMS has slightly different characteristics, Figure 10.4 shows the overall structure of a database as defined by the SQL standard. Users are defined within the individual database instance and granted permissions by the DBA. The **schema** is a container that serves as a namespace so that duplicate table names can be avoided. Originally, it was defined so that each user would have a separate space to create tables. Two users could each create a table named Employee without causing a problem. Today, schemas can be created for any purpose, not just for each user. The catalog was proposed in the SQL 99 standard, primarily to make it easier to find and access related schemas by placing them into one container. At this point, it is not likely that any DBMS supports the catalog element. However, the schema approach is relatively common. Users and applications are assigned to a default schema, and tables and views within that schema are directly accessible (depending on the security permissions, of course). But sometimes you need to access tables or views stored in a different schema. In these situations, you need to use the full name of the item. The full name includes the schema name (and eventually the catalog name). For example, if you want to access the Employee table in the Corporate schema, you would use SELECT * FROM Corporate.Employee to indicate the full name of the table. If you need to specify the catalog (e.g., Main), you would use Main. Corporate.Employee as the full name of the table. More commonly, the name of the database is used instead of a catalog name. The standard database elements such as tables, views, and triggers reside within each schema. One of the tasks of the DBA (and the DA) is to identify when to create new schemas. Although there are no specific rules, keep in mind that the purpose of a schema is to isolate and compartmentalize applications.

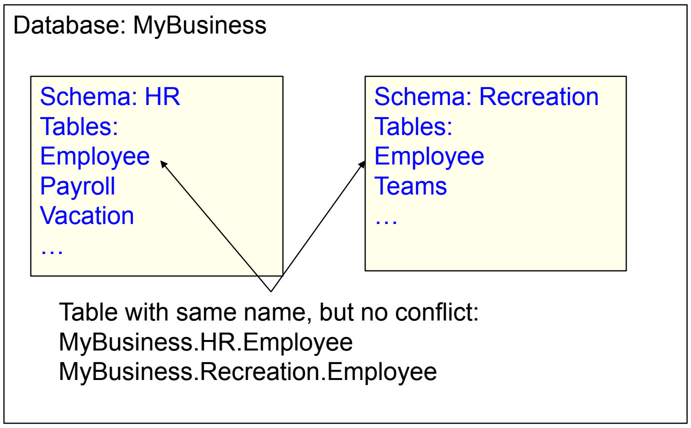


Figure 10.5

Usefulness of schemas. Schemas are separated collections of tables, queries, functions, and triggers. Notice how each schema has a table named Employee, but they are completely separate tables.

Figure 10.5 gives an example of using a schema to support two different applications within the same database. The database name is *MyBusiness* and the two schemas are *HR* and *Recreation*. Note that both schemas contain a table named *Employee*. These two tables are completely separate, can contain different columns and have different security permissions. This approach is particularly useful when an application is purchased from an outside vendor and it is not possible to rename the tables. Placing the application definition within its own schema keeps it separated from everything else. But all of the tables can be shared across schemas as necessary.

The other option would be to place all applications into separate databases. What is the difference? The two big differences are (1) backup/files and (2) security permissions. Each database uses different files for data storage and rollback. Plus, security objects are defined separately within each database. So every time a new database is created, procedures and people need to be assigned to handle backup, and assign and test security permissions. Sometimes the increased separation is worth the effort, other times it is easier and faster to simply create a new schema within an existing database. This is just one of the decisions that needs to be made by the DBA.

Metadata

How does a DBA find out what is stored in each database? Each vendor provides tools to help DBAs accomplish common tasks. Most have a graphically oriented approach to make them easier to use. On the other hand, DBAs often choose to perform tasks using SQL by building specific procedures. The SQL commands provide detailed control over an operation and can be written to handle dozens or hundreds of operations at one time. For example, the graphical approach is easy to use for adding one user, but if you need to add 100 users, it is easier to write an SQL procedure that pulls the list of users from a file or a temporary table.

In terms of administration, one of the powerful aspects of relational database systems is that even the administrative data is stored in tables. This **metadata**

Schemata	
Tables	
Domains	
Views	SELECT Table_Name, Table_Type
Table_Privileges	FROM Information_Schema.Tables
Referential_Constraints	WHERE table_name LIKE 'Emp%'
Check_Constraints	
Triggers	
Trigger_Table_Usage	
Parameters	
Routines	

Figure 10.6

Information Schema. A few of the 61 views in the standard are listed on the left. The sample query shows how DBAs can query the metadata views to quickly find a specific item.

is data about the data. For example, a system table contains a list of all the user tables. The SQL 99 standard describes the Information_Schema which consists of a set of views that provide documentation on the database. Technically, the Information_Schema views retrieve data from the Definition_Schema tables; however, DBMS vendors might choose not to implement the Definition_Schema. DBMS vendors have already developed proprietary system tables to hold the metadata. The drawback to this approach has been that there is no consistency across products, so DBAs have to learn different commands for each DBMS. As vendors implement the newer standards, DBAs should find it easier to work with products from multiple vendors. As of 2013, only some vendors have implemented the Information_Schema views. Currently, most DBAs use vendor-specific queries, such as the Describe command or the USER_TABLES view in Oracle; or the MSysObjects system table in Access; or the sys.xxx views in SQL Server; or the syscat views (e.g., syscat.Tables) in IBM's DB2. It would be relatively easy to create an Information_Schema in most of these systems and add your own view definitions for the standard metadata. Essentially, the standard views extract the information from the underlying metadata tables. Note that Microsoft's SQL Server does support the Information_Schema views (along with proprietary sys. views). Technically, Oracle does not support the Information_Schema, but search the Web and you will find downloadable files that can be installed in an Oracle DBMS to provide most of Information_Schema definitions.

Figure 10.6 shows some of the common elements of the Information_Schema. The SQL command illustrates how to obtain a partial list of the tables, based on the name. Commands of this type are useful when a database has hundreds of tables and views. Instead of scrolling through dozens of pages looking for a specific table, you can use the power of SQL to quickly find the exact table needed. In the example, note that you should always retrieve the Table_Type as well as the name. Tables can be base types (that actually hold the data), views, or derived tables. You can use these standard views to get lists of tables, views, triggers, and other procedures. You can also use them to get detailed information about tables and views, such as a list of columns, data types, and SQL statements for the views and functions.

SELECT MSysObjects.Name, MSysObjects.Type
FROM MSysObjects
WHERE MSysObjects.Name Like “EMP*”;
	Access

SELECT *
FROM sys.tables
WHERE name Like N’Emp%’;
	SQL Server

SELECT *
FROM ALL_TABLES
WHERE TABLE_NAME Like ‘Emp%’;
	Oracle

SELECT *
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_NAME LIKE N’Emp%’;
	SQL Standard

Figure 10.7

Usefulness of schemas. Schemas are separated collections of tables, queries, functions, and triggers. Notice how each schema has a table named Employee, but they are completely separate tables.

Figure 10.7 shows examples of queries used to obtain metadata in four different systems. The structure of the queries is the same in each case, but the table/query names are different as are the columns and the data retrieved. In particular, the Type values are different in each case. Still, it is useful to have the power of SQL to find objects by partial name if necessary.

Database Tasks by Development Stages

What DBA tasks need to be performed as an application is developed? Whichever development methodology you follow (e.g., traditional systems development life cycle, rapid development, or prototyping), certain database tasks are required at each step. Most tasks are performed by the application developers, but some involve the DA. Many require communication with the DBA, both to get advice and to provide information to help the DBA set up the databases.

Database Planning

During the feasibility and planning stages, you will have to make an estimate of the data storage requirements. These initial estimates will be rough, but they will help determine the size and capacity of the hardware needed to support the application. For example, if you are building a simple database to track materials that will be used by five people, the database might require less than 100 megabytes of storage and run on a desktop computer. If the initial size estimates start to exceed a few hundred megabytes of storage, a file server with high-speed disk drives might be more appropriate, and the system can probably run on the “free” copies of DBMS software. As the database estimates approach gigabytes or terabytes, you should investigate special database hardware and parallel-processing systems.

The initial investigation should also provide some idea of the number of forms and reports that will be needed, as well as their complexity. These numbers will be used to estimate the time and cost required to develop the system. An experienced DBA can provide estimates of space requirements from similar projects. Company records on other projects can provide estimates of the average time to develop forms and reports.

- Teamwork
  - Data standards.
  - Data repository.
  - Reusable objects.
  - CASE tools.
  - Networks and communication.
- Subdividing projects
  - Delivering in stages: versions.
  - Normalization by user views.
  - Assigning forms and reports.

Figure 10.8

Managing database design. Database design requires teamwork and standards to ensure that individual components can be integrated into a complete application. CASE tools and networks improve communication through a centralized repository of design data.

Database Design

The basic goal of the design stage is to identify the user needs and design the appropriate data tables. Data normalization is the primary database-related activity in this stage. The final table definitions will also provide better estimates of the storage requirements.

Teamwork coordination and project management are important administrative tasks at this stage. As highlighted in Figure 10.8, teamwork is supported with data standards as defined by the DA. Projects can be split into pieces and assigned to each team member. The ability to integrate the pieces into a complete application is provided through standards and communication. Communication is enhanced through a shared data repository, networked tools, e-mail, and **computer-aided software engineering (CASE)** tools. Leading CASE tools include Oracle Designer/2000, Rational Rose, IEF, and IBM's Visual Age. These tools provide a centralized repository for all project work, including diagrams, data definitions, and programming code. As team members work on their portion of the project, they can see the rest of the project. In an OO project, they can use the objects created by other teams.

From the perspective of data design or normalization, the project is often split by assigning forms and reports to individual team members. Each person is then responsible for identifying the business assumptions and defining the normalized tables needed for the assigned forms. Periodically, the individuals combine their work and create a centralized list of the tables that will be used in the database. This final list must follow the standards established by the DA.

Database Implementation

The primary database tasks required for implementation are listed in Figure 10.9. The major steps are development of the application and the user interface. Management and organizational tasks largely entail determining the overall look and feel of the application. Once the overall structure is determined, programming standards and testing procedures facilitate teamwork and ensure quality. Another important management task is to assign ownership of the various databases. Owners should be from business management. Data owners are responsible for identi-

- Standards for application programming.
    - User interface.
    - Programming structure.
    - Programming variables and objects.
    - Test procedures.
- Data access and ownership.
- Loading databases.
- Backup and recovery plans.
- User and operator training.

Figure 10.9

Implementation management. The user interface must be carefully chosen. Programming standards and test procedures help ensure compatibility of the components and provide quality control. Business managers should be assigned ownership of the data, so they can make final determinations of security conditions and quality. Backup and recovery plans have to be created and tested. Training programs have to be created for operators and users.

fying primary security rules and for verifying the accuracy of the data. If the DBA has any questions about access rights or changes to the data, the DBA can obtain additional information and advice from the data owner.

Backup and recovery procedures have to be established and tested. If any component fails, the database logs should be able to fully restore the data. Backups are often handled in two forms: full backup at predefined checkpoints and incremental backups of changes that have occurred since the last full backup. Complete backups are easier to restore and provide safer recovery. However, they can be time-consuming and require large amounts of backup space. For small databases, full backups are not a problem. For large, continually changing transaction databases, it may only be possible to perform a full backup once a week or so.

Users and operators also have to be trained. No matter how carefully the user interface is designed, there should always be at least an introductory training session for users. Similarly, computer operators may have to be trained in the backup and recovery procedures.

Database Operation and Maintenance

Once the database is placed in operation, the DBA performs most of the management tasks. The primary tasks are to (1) monitor usage and security, (2) perform backups and recovery, and (3) support the user.

Monitoring performance and storage space is a critical factor in managing a database and planning for growth. All of the big DBMS vendors provide graphical tools to display a variety of performance measures. Figure 10.10 shows some of the basic measures generated in Oracle's Enterprise Manager. This example shows a database with almost no load. In a production environment, you would look for spikes in certain statistics—particularly if the occur at set times every day. You would also watch for trends over time. For instance, if you see usage rates increasing and performance declining over time, you can use the data to predict future problems and schedule upgrades.

Monitoring performance and storage space is a critical factor in managing a database and planning for growth. All of the big DBMS vendors provide graphical tools to display a variety of performance measures. Figure 10.10 shows some

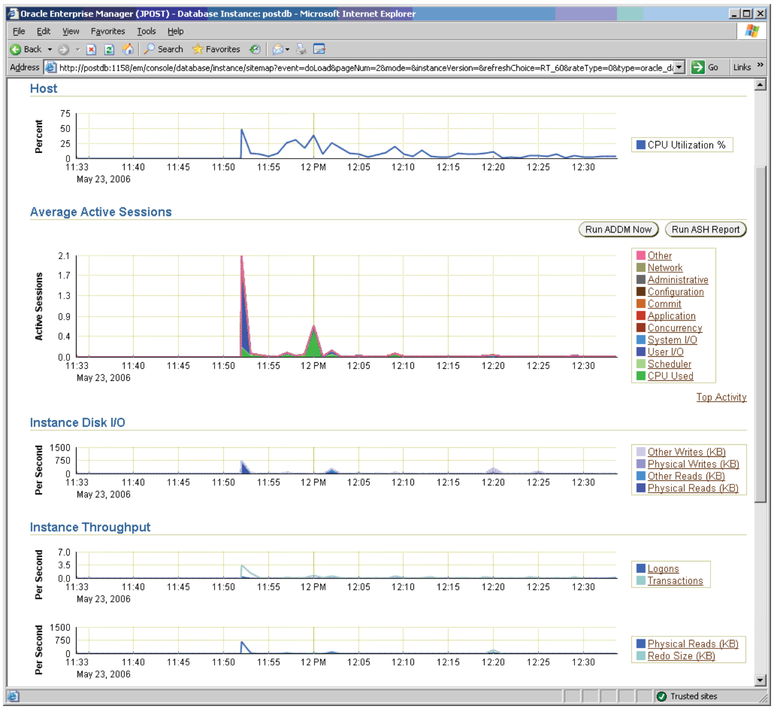


Figure 10.10

Oracle's basic performance monitoring statistics. The load on this server is minimal, but the DBA can watch these charts on a regular basis to spot problems and identify trends.

of the basic measures generated in Oracle's Enterprise Manager. This example shows a database with almost no load. In a production environment, you would look for spikes in certain statistics—particularly if they occur at set times every day. You would also watch for trends over time. For instance, if you see usage rates increasing and performance declining over time, you can use the data to predict future problems and schedule upgrades.

Monitoring is also used to fine-tune the application performance and to estimate growth and plan for future needs. Security access and changes are also monitored. Security logs can track changes to critical data. They can also be specified to track usage (both read and write) by individual users if there is a suspected problem. Monitoring user problems as well as performance provides useful feedback on the application. If users consistently have problems in certain areas, the design team should be encouraged to improve those forms. Similarly, if some users are running queries that take a long time to execute, the design team should be called in to create efficient versions of the queries. For example, do not expect a user to recognize or correct a correlated subquery. Instead, if the DBA sees users running complex queries that take too long to run, the team should add a new section to the application that stores and executes a more efficient query.

Similarly, if some people are heavily using certain sections of the database, it might be more efficient to provide them with replicated copies of the main sections. If the users do not need up-to-the-minute data, a smaller database can be set up on a server and updated nightly. The users end up with faster response times because they have a smaller database and less communication time. The rest of the database runs faster because there are fewer heavy users.

Database vendors provide some powerful tools to help analyze queries and database performance. With these tools, you can break apart the entire query process to see exactly which step is taking the most time. With this knowledge, developers can work on alternative solutions to avoid the bottlenecks. Other tools can monitor for deadlock and transaction problems, making it relatively easy to correct problems. **Tuning** a large database to improve performance is a complex issue and depends heavily on the capabilities and tools of the specific DBMS.

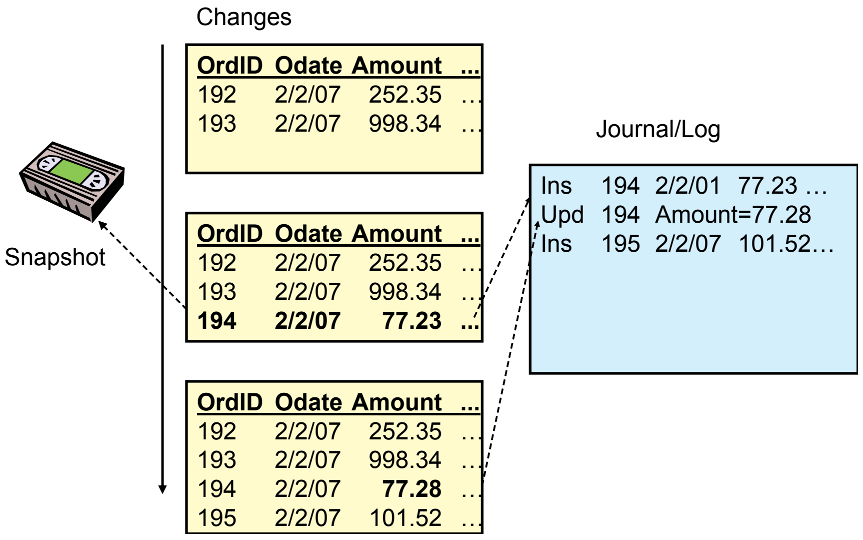
Backup and Recovery

How do you back up data that is constantly changing? Perhaps the most critical database management task is backup. No matter how well you plan, no matter how sophisticated your security system, something will go wrong. Database managers and developers have an obligation to plan for disasters. The most critical aspect of planning is to make sure that a current copy of the database is easily accessible. Any type of disaster—fire, flood, terrorist attack, power failure, computer virus, disk drive crash, or accidental deletion—requires backup data. Given the low cost of making and storing backup copies, there is no excuse for not having a current backup available at all times.

As shown in Figure 10.11, database backups provide some interesting challenges—particularly when the database must be available 24 hours a day, 7 days a week (abbreviated to **24-7**). The basic problem is that while the database is making a backup copy, changes could still be made to the data. That is, every copy of the database is immediately out of date—even while it is being made. A related

Figure 10.11

Backup of a changing database. Backup takes a snapshot at one point in time. New changes are stored in the journal or log. Recovery loads snapshot and adds or deletes changes in the journal.



issue is that the DBMS copy routines might have to wait to copy portions of the database that are currently in use (possibly creating a deadlock situation).

Fortunately, the larger database systems provide many tools to solve these problems. One approach is to take a **snapshot** of the tables. The snapshot represents the status of a table at one instant in time. Because of ongoing changes, this snapshot is likely to be inconsistent or record only portions of a transaction. To solve these problems, the **transaction log** (or journal) records all changes as they are written to the database. These log files must also be backed up. However, since the transaction system only adds new rows to the logs, it is relatively easy to back them up with no contention or deadlock issues. On restore, the system loads the snapshot data and then plays back the changes in the log files to make everything consistent, and record changes that were made after the instant the snapshot was taken.

If a problem arises with the main database files, the database has to be restored from the backup tapes. First the DBMS loads the most recent snapshot data. Then it examines the transactions. Completed transactions are rolled forward, and the changes are rewritten to the data tables. If the backup occurred in the middle of a transaction and the transaction was not completed, the DBMS will roll back or remove the initial changes and then restart the transaction. Remember that a transaction consists of a series of changes that must all succeed or fail together. The DBMS relies on the application's definition of a transaction as described in Chapter 7.

Backups have to be performed on a regular schedule. Occasionally, the schedule will have to be revised—particularly if the database records many changes. Remember that every change since the last backup is recorded in a journal or transaction log. The DBA has to watch the space on the transaction log. If it becomes too full, a backup has to be run earlier than scheduled. If these unexpected backups happen too often, the schedule should be changed. It is possible to make complete backups and incremental backups. An incremental backup saves only the data that was changed since the last backup, making it considerably faster than a full backup. However, the system has to work harder to piece everything together if you need to restore the entire database. With high-speed processors and storage, the additional time to restore the data might be minimal today, but you still face a greater risk of damage if one of the incremental backups has problems.

As a side note, be careful with your development databases on your own workstation. In particular, SQL Server default logging mode automatically extends the transaction log—potentially reaching several gigabytes in size. You have to run a full backup to clean up the transaction log. Alternatively, your development databases can be configured without transaction logging.

Backup tapes must be stored offsite. Otherwise, a fire or other disaster might destroy all data stored in the building. Snapshot and journal logs should be copied and moved offsite at least once a day. Networks make it easier to transfer data if the company is large enough to support computer facilities at more than one location. Several companies provide disaster-safe vaults for storage of data tapes and disks. In extreme situations, it might pay to have duplicate computer facilities and to program the system to automatically mirror changes from the main database onto the secondary computer in a different location. Then when something goes wrong, the secondary computer can immediately pick up the operations. However, even in this situation, you should make physical backup copies.

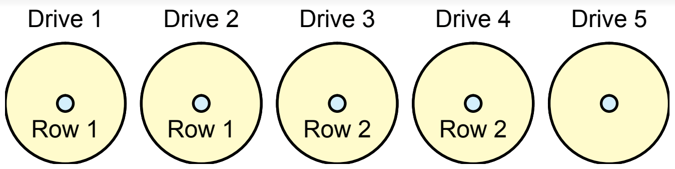


Figure 10.12

RAID drives. Pieces of data, such as a row, are written on two different drives. Other rows are spread across drives so that multiple disk reads and writes can occur at the same time to dramatically improve performance.

An increasingly popular approach to backups is to create mirrored copies of the database—using either software or hardware. With a high-speed network, you can configure the DBMS to write all changes to a second location (mirror). This server could pick up the load if the first server is slow or something crashes. You can also mirror the data in one place—by using a **redundant array of independent drives (RAID)**. With RAID (and other striping systems), each piece of data is written in two locations on different physical drives. If one drive fails, the DBA simply removes it and installs a replacement. The system automatically uses its internal copy of the data that is spread across the other drives. As indicated in Figure 10.12, the system is also considerably faster than using a single drive, since each portion of data can be simultaneously written to a different physical drive. Because disk drives are mechanical, they are the slowest and least reliable component of the computer system (not counting humans). Striping data across multiple drives means that data can be read or written in parallel at the same time—providing substantially faster throughput, as well as providing duplication to protect from the failure of a single drive.

Physical Configuration

How should computers be configured for DBMS software and database files? Obviously, this question has multiple answers and each situation is different from others, so the final selection needs to be based on the size of the project and the amount of money available. However, a few key elements tend to be useful in many situations based on current computer and network capabilities.

It is important to recognize that computers and networks have improved dramatically in the past few years. These trends are likely to continue for at least a few more years. One of the biggest trends in computer architecture is the growing importance of parallel processing—most computer processors are being built with multiple cores, so servers can easily contain a handful to thousands of independent processors. A key consequence of having multiple processors is the creation of a **virtual machine (VM)**. With a VM, the computer running tasks is essentially hosted on a parent computer.

Figure 10.13 shows the logical structure of a VM. A single physical machine runs a base operating system that contains a VM hypervisor program that is used to configure and control VMs. A VM is loaded with its own operating system (with its own license). The VM operating system thinks it is running on an independent machine, but it actually shares processors and memory on the base computer. One of the main strengths of this approach is that more RAM and more processing power can be given to the VM when it is needed. Also, the VM definition,

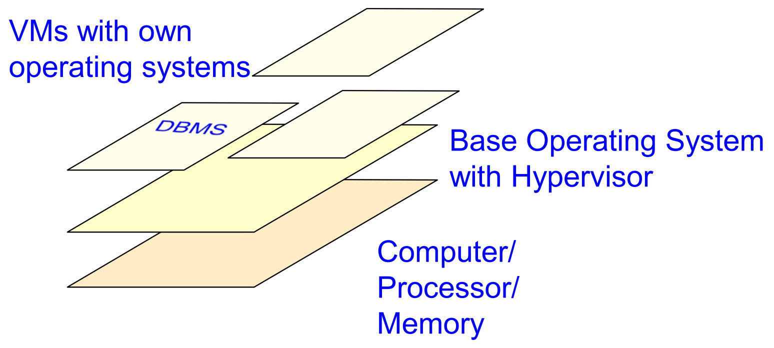

VMs with own operating systems
DBMS
Base Operating System with Hypervisor
Computer/ Processor/ Memory

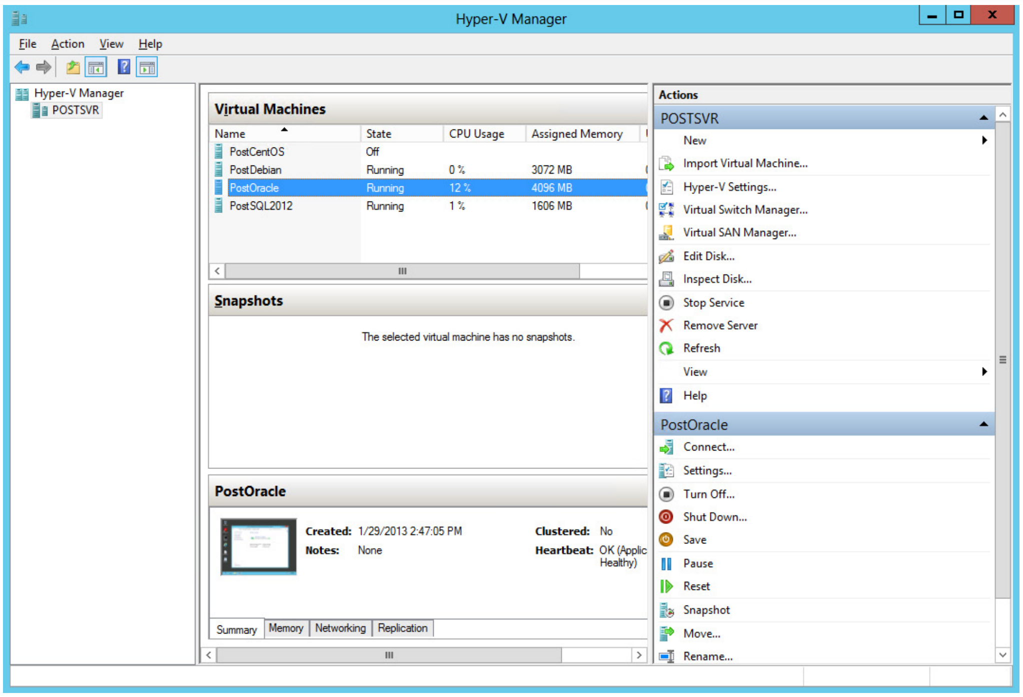
Figure 10.13

Virtual Machine. A physical computer is the base layer and it runs a base operating system that contains a hypervisor program. Each VM is installed with a separate operating system and appears to be a separate physical machine—but it shares the physical processor and memory on the base machine.

including the operating system and the software configuration is stored as a file on the base computer. It is straightforward to back up this VM definition and restore it on a different computer if the base machine crashes or is destroyed in a fire. Spend several hours configuring a new OS, installing, and configuring Oracle and you will quickly appreciate the value of a simple VM backup and restore facility.

Figure 10.14 shows the Microsoft Hyper-V tool to manage VMs in Server 2012. The base server consists of a single computer with an 8-core processor and 32 GB of actual RAM. The Hyper-V instance is running on this physical server. Four virtual machines are defined (two running versions of LINUX, two with Windows). Disk space is allocated on the main server to hold the operating systems in VHD files. This space is allocated to an individual VM which sees it as its own disk drive. Shared space can also be allocated using the virtual storage area network (SAN) manager. Using VMs makes it easier to set up systems for testing. It is also straightforward to clone a system, so if anything goes wrong with an upgrade or expansion, the original version can be restored quickly.

A second major trend is the increasing speed of local area networks. Most networks can easily handle 1 gigabits per second (gps) transfer speeds. This speed is already faster than most hard-drive transfer speeds; and network speeds are increasing every few years. Consequently, as shown in Figure 10.15, hard drives no longer need to be located in the same box as the server. Several vendors now sell **network attached storage (NAS)** or **storage area network (SAN)** devices. The devices usually have RAID configurations—both for speed and for internal backup. Several independent vendors sell these devices, and even provide separate, automatic backup systems. The DBMS sees the storage device as a giant disk drive, but the device automatically writes striped copies of the data, and writes a cached copy that is backed up to tape at regular intervals. The system provides immediate online copies of all data, as well as long term copies. Essentially, the systems transfer the responsibility for backup from software to hardware, and they store and retrieve data very rapidly. Separating the server from the data improves reliability and makes it easier to restore operations if either the server crashes or the SAN fails.



Virtual Machines

Name	State	CPU Usage	Assigned Memory
PostCentOS	Off		
PostDebian	Running	0 %	3072 MB
PostOracle	Running	12 %	4096 MB
PostSQL2012	Running	1 %	1606 MB

Snapshots

The selected virtual machine has no snapshots.

PostOracle

Created: 1/29/2013 2:47:05 PM
Notes: None
Clustered: No
Heartbeat: OK (Applic Healthy)

Figure 10.14

Hyper-V Management. Windows Server 2012 includes the Hyper-V tool to manage virtual machines. It is used to assign physical attributes such as memory, processors, and network access.

When using a VM, it is always a good idea to store the actual database on an external drive. Except for small systems for development and testing, the data should not be stored on a virtual drive associated with a VM. The intervening layers of software on a VM virtual drive will slow down all data operations. It is faster to write directly to the drive hardware, and SAN devices can be optimized for transaction performance.

Security and Privacy

What security techniques are used to protect databases? Computer security is an issue with every company today, and any computer application faces security problems. A database collects a large amount of data in one location and makes it easy for people to retrieve and change data. In other words, a database is a critical resource that must be protected, and it is a tempting target for attackers. Yet the same factors that make a database so useful also make it more difficult to secure. In particular, the purpose of a database is to share data. In a security context, you want to control who can share the data and what those users can do with it.

Computer security is often split into three categories: (1) physical security, (2) logical security, and (3) behavioral security. **Physical security** is concerned with physically protecting the computing resources and preparing for physical disasters that might damage equipment or data. **Logical security** consists of protecting the data and controlling access to the data. **Behavioral security** is trickier because it emphasizes the role of people or employees. It is related to logical security but involves interesting problems because it deals with mistakes that people make.

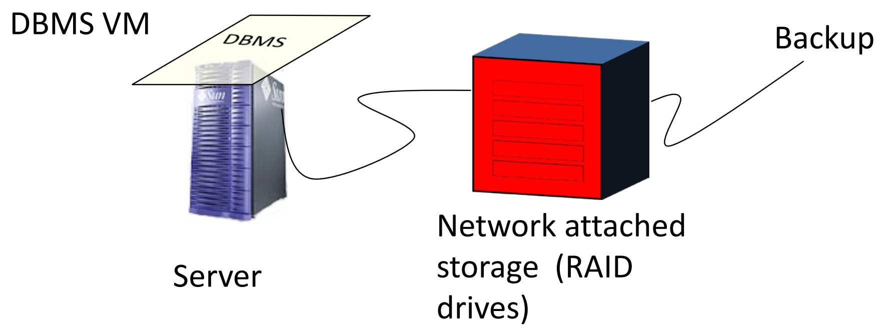


Figure 10.15

Network attached storage. The database files should be stored on a network drive. These RAID devices are fast and handle their own backups through RAID and streaming to tape drives. Drive maintenance and replacement can be handled independently of the server.

Data Privacy

Privacy is related to security but with a slight twist. Companies and governmental agencies collect huge amounts of data on customers, suppliers, and employees. Privacy means controlling the distribution of this data and respecting the wishes of these external people. Figure 10.16 shows some of the demands placed on business data in terms of marketing, employee management, and governmental requests. The concepts of keeping data accurate and limiting who has access to it are the same for security and for privacy. The differences lie in the objectives and motivation. In terms of security, every company has a self-interest in keeping its data safe and protected. In terms of privacy—at least in the United States—there are few regulations or limitations on what a company can do with personal data. Whereas customers and employees may want a company to keep personal data private, companies may have a financial incentive to trade or sell the data to other companies.

In terms of data privacy, the most important question is, Who owns the data? In most cases the answer is the company or individual that collects the data. Some people, particularly in Europe, have suggested that the individuals should be considered to be the owners. Then companies would have to get permission—or pay for permission—to use or trade personal data. So far, technical limitations have prevented most of the suggested payment schemes from being implemented. However, companies must pay attention to changes in the laws regarding privacy.

Database workers have an ethical obligation in terms of data privacy. Many times, you will have access to personal data regarding customers and other employees. You have an obligation to maintain the privacy of that data: You cannot reveal the data to other people. In fact, you should avoid even reading the data. You should also not tolerate abuses by other workers within the organization. If you detect privacy (or security) violations by others, you should report the problems and issues to the appropriate supervisors.

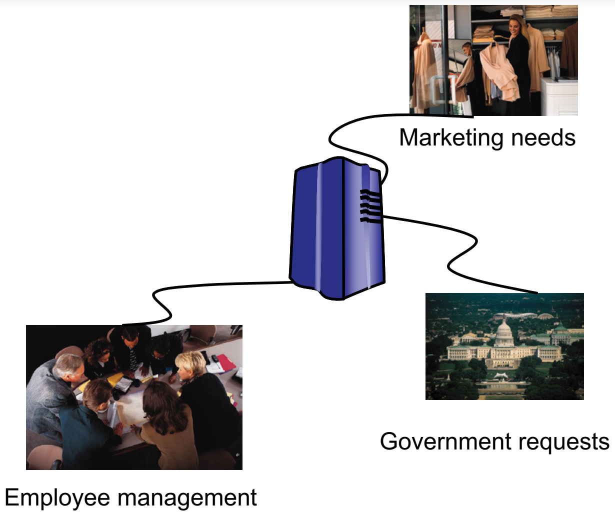


Figure 10.16

Privacy. Many reasons exist to collect and analyze data. People might find some reasons invasive. Although few privacy laws exist, businesses and database administrators should consider the implications and trade-offs of using this data.

Threats

What are the primary threats to computer security? What possible events cause nightmares for database administrators? Is it the outside hackers or crackers that you see in the movies? Is it tornadoes, hurricanes, or earthquakes (also popular in movies)?

No. The primary threat to any company comes from "insiders." Companies can plan for all of the other threats, and various tools exist to help minimize problems. However, you have to trust your employees, consultants, and business partners. For them to do their jobs, they need physical access to computers and logical access to the databases. Once you are committed to granting access, it becomes more difficult to control what they do. Not impossible, just more difficult. Even when employees are honest and cautious, they are still going to make mistakes because they are human. Some behavioral issues are easier to deal with than others. Writing down passwords or giving them out over the phone are risky behaviors. Picking up a USB drive in the parking lot and inserting it into a computer is flat out dangerous—but hard for people to resist. Yes, the volume of attacks from the Internet is high, but in many ways they are easier to stop.

Another, more insidious threat comes from programmers who intentionally damage data. One technique is to embed a time bomb in a program. A time bomb requires the programmer to enter a secret code every day. If the programmer leaves (or is fired) and cannot enter the code, the program begins deleting files. In other cases programmers have created programs that deliberately alter data or transfer funds to their own accounts. These examples illustrate the heart of the problem. Companies must trust their programmers, but this trust carries a potential for considerable damage or fraud. It is one of the reasons that companies are so sensitive about MIS employee misconduct. As a developer, you must always project an image of trust.

- Backup data
- Backup hardware
- Disaster planning and testing
- Prevention
    - Location
    - Fire monitoring and control
    - Control physical access

Figure 10.17

Physical security controls. Backup data is the most important step. Having a place to move to is a second step. Disaster plans and prevention help prevent problems and make recovery faster.

Physical Security

In terms of physically protecting the computer system, the most important task is to make sure you always have current backups. This policy of maintaining backups also applies to hardware. In case of a fire or other physical disaster, you need to collect the data tapes and then find a computer to load and run them. Instead of waiting until a disaster happens, you really need to create a disaster plan.

A **disaster plan** is a complete list of the steps that the IS department will take if a disaster hits the information system. The plan details who is in charge, describes what steps everyone will take, lists contact numbers, and tells how you will get the systems up and running. One popular method of finding an alternative computer is to lease a hot site from a disaster planning company. A **hot site** consists of a computer facility that has power, terminals, communication systems, and a computer. You pay a monthly fee for the right to use the facility if a disaster occurs. If there is a disaster, you activate the disaster plan, collect the data tapes, load the system, load your backup tapes, and run the system from the hot site. A slightly cheaper alternative is to lease a cold site. A **cold site** or **shell site** is similar to a hot site, but it does not have the computer and telecommunications equipment. If a disaster occurs, you call your hardware vendor and beg for a new computer. Actually, vendors have been very cooperative. The catch is that it can still take several days to receive and install a new computer. Can your company survive for several days without a computer system? To replace smaller computers, some of the disaster recovery companies can deliver a truck to your site and run the system from your parking lot.

As summarized in Figure 10.17, prevention is another important step in providing physical security. Computer facilities should have fire detection and protection systems. Similarly, computer facilities should be located away from flood plains, earthquake faults, tidal areas, and other locations subject to known disasters. Physical access to computers, network equipment, and personal computers should be limited. Most companies have instituted company badges with electronic locks. Access by visitors, delivery people, and temporary employees should be controlled.

Today, with the need for continuous online systems, the role of backup facilities has changed. Many companies now have multiple data centers in separate locations. Each center can run all of the operations but normally share the load and the data. If one system goes down, the other can immediately pick up the transactions and operations. Cloud-based operations can also be used to increase the scale of a system if something goes wrong in one location.

Managerial Controls

Because the major threats to data security come from company insiders, traditional managerial controls play an important role in enhancing security. For example, one of the most important controls begins with the hiring process. Some firms perform background checks to verify the character and trustworthiness of the employees. Even simple verification of references will help to minimize problems. Similarly, firms have become more cautious when terminating employees—particularly MIS employees with wide access to databases. Even for routine layoffs, access rights and passwords are revoked immediately.

Sensitive jobs are segmented. For example, several employees are required to complete financial transactions. Transactions involving larger amounts of money are routed to higher level employees. Similarly, outside institutions like banks often call back to designated supervisors to verify large transactions. Transactions are often monitored and recorded in terms of the time, location, and person performing the operations.

In some cases, security can be enhanced through physical control over the hardware. Centralized computers are placed in locked and guarded rooms. Employees are often tracked through video monitors. Security badges are also used to track employee access to locations and computer hardware.

Consultants and business alliances also raise security concerns. Generally, you have less control over the selection of the consultant and any partnership employees. Although you have control in the selection of a consulting firm, you have little control over the specific employees assigned to your location. These risks can be controlled by limiting their access to the data and restricting their access to physical locations. In some situations, you may also want to pair an internal employee with each consultant.

Logical Security

The essence of logical security is that you want to allow each user to have some access to the data, but you want to control exactly what type of access the user will have. You also want to monitor access to the data to identify potential problems. Figure 10.18 notes the three basic problems that you want to avoid: confidentiality (unauthorized disclosure), integrity (unauthorized modification), and accessibility (unauthorized withholding of information or denial of service).

Confidentiality refers to information that needs to be protected so that only a select group of users can retrieve it. For example, the company's strategic marketing plans need to be protected so that no competitor can retrieve the data. To be safe, only a few top people in the company would have access to the plans.

Integrity applies to information that is safe to display to users, but you do not want the users to change it. For example, an employee should be able to check the human resource files to verify his or her salary, remaining vacation days, or merit

Figure 10.18

Logical security problems. Each situation can cause problems for the company, including financial loss, lost time, lost sales, or destruction of the company.

Confidentiality	Unauthorized disclosure
Integrity	Unauthorized modification
Accessibility	Unauthorized withholding

Do not use words in a dictionary.
Do not use personal (or pet) names.
Include nonalphabetic characters.
Use at least eight characters.
Change it often.

Figure 10.19

Password suggestions. Pick passwords that are not in a dictionary and are hard to guess. The catch is, Can you remember a convoluted password?

evaluations. But it would be a mistake to allow the employee to change any of this data. No matter how honest your employees are, it would be a dangerous temptation to allow them to alter their salary, for example.

The third problem is subtle but just as dangerous. Consider what would happen if the chief financial officer needs to retrieve data to finalize a bank loan. However, the security system is set incorrectly and refuses to provide the data needed. If the data is not delivered to the bank by the end of the day, the company will default on several payments, receive negative publicity, lose stock value, and potentially risk going under. The point is that withholding data from authorized users can be just as dangerous as allowing access to the wrong people.

Assuming you have a sophisticated computer system and a DBMS that supports security controls, two steps are needed to prevent these problems. First, the computer system must be able to identify each user. Second, the owner of the data must assign the proper access rights to every piece of data. The DBA (or a security officer) is responsible for assigning and managing user accounts to uniquely identify users. The application designer and data owners are jointly responsible for identifying the necessary security controls and access rights for each user.

User Identification

One of the major difficulties of logical computer security is identifying the user. Humans recognize other people with sophisticated pattern-recognition techniques applied to appearance, voice, handwriting, and so on. Yet even people can be fooled. Computers are weak at pattern recognition, so other techniques are required.

The most common method of identifying users is by accounts and passwords. Each person has a unique account name and chooses a password. In theory, the password is known only to the individual user and the computer system. When the user enters the correct name and matching password, the computer accepts the identity of the person.

The problem is that computers are better than people at remembering passwords. Consequently, people make poor choices for passwords. Some of the basic rules for creating passwords are outlined in Figure 10.19. The best passwords are long, contain nonalphabetic characters, have no relationship to the user, and are changed often. Fine, but a user today can easily have 30, 40, or more different accounts and passwords. Almost no one can remember every account and the convoluted passwords required for security. So there is a natural inclination either to write the passwords in a convenient location (where they can be found by others) or choose simple passwords (which can be guessed).

Passwords are the easiest (and least expensive) system to implement at this time. Within a given company, it is possible to implement a central security server (e.g., Kerberos used in Microsoft's Active Directory), where a user logs into the main server and all other software verifies users with that server. Another approach is to use password generator cards. Each user carries a small card that generates a new password every minute. At login, the computer generates a password that is synchronized to the card. Once the password is used, it is invalidated, so if an interloper observes a password, it has no value. The system still requires users to memorize a short password just in case a thief steals the password card. Of course, if a user loses the card, that user cannot get access to the computers. Encrypted software variations can be loaded onto laptop computers, which then provide access to the corporate network.

Other alternatives are being developed to get away from the need to memorize passwords. Biometric systems that measure physical characteristics already exist and are becoming less expensive. For example, fingerprint, handprint, iris pattern, voice recognition, and thermal imaging systems now work relatively well. The advantage to biometric approaches is that the user does not have to memorize anything or carry around devices that could be lost or stolen. The main drawback is cost, since the validating equipment has to be installed anywhere that employees might need access to a computer. A secondary problem is that although the devices are good at preventing unauthorized access, many of them still have relatively high failure rates and refuse access to authorized users.

After individual users are identified, most systems enable you to assign the individuals to groups. Groups make it easier to assign permissions to users. For example, by putting 100 employees into a clerical group, you can grant permissions to the group, which is much faster than assigning the same permissions 100 times.

Most DBMSs identify the user either internally (database login) or through integration with the operating system (particularly Windows Active Directory). Similarly groups or roles can be created within the DBMS or within Active Directory. The choice depends on the company and the type of application being created. Centralizing logins through the operating system (Windows) makes it easier for users to access data and makes it easier for security personnel to monitor and change groups. But it requires close cooperation between the DBA, developers, and the network security group.

Access Controls

After users are identified, they can be assigned specific permissions to any resource. From a database perspective, two levels of access must be set. First, the user must be granted access to the overall database or standalone application, using operating system commands. Second, the user or group must be granted individual permissions, using the database security commands.

The privileges listed in Figure 10.20 apply to an entire table or query. The most common privileges you will grant are read, update, and insert. Delete permission means that a user can delete an entire row: —it should be granted to only a few people in specific circumstances. Privileges that are more powerful can be granted to enable users to read and modify the design of the tables, forms, and reports. These privileges should be reserved for trusted users or DBAs. Users will rarely need to modify the design of the underlying tables. These privileges are granted to developers.

Read data
Update data
Insert data
Delete data
Open/run
Read design
Modify design
Administer

Figure 10.20

DBMS privileges. These privileges apply to the entire table or query. The first three (read, update, and insert) are commonly used. The design privileges are usually granted only to developers.

With most database systems the basic security permissions can be set with two SQL commands: **GRANT** and **REVOKE**. Figure 10.21 shows the standard syntax of the GRANT command. The REVOKE command is similar. SQL 92 provides some additional control over security by allowing you to specify columns in the GRANT and REVOKE commands, so you can grant access to just one or two columns within a table. However, the privilege applies to every row in the table or query.

Most of the database systems provide a visual security tool to help assign access rights. The SQL commands are useful for batch operations when you need to assign permissions to a large number of tables or for several users at the same time. The visual tools are easier to use for single operations. They also make it easy to see exactly which permissions have currently been assigned to individuals. However, the SQL statements can be used in scripts to automate security assignments.

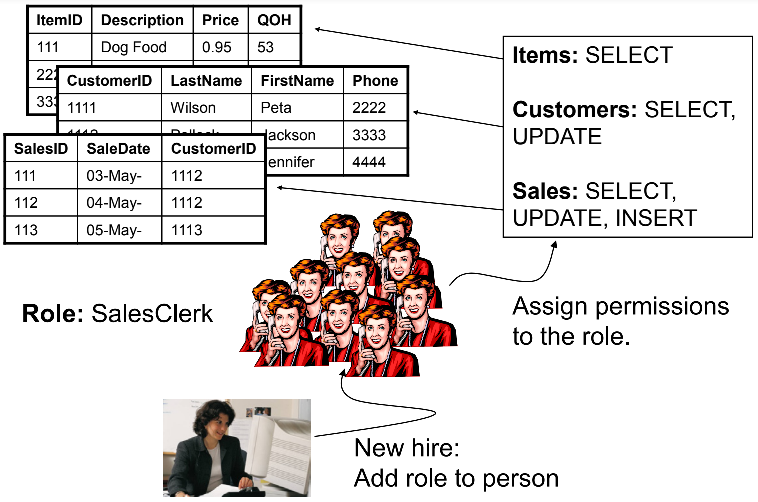
The GRANT command offers an additional option that is sometimes useful. As the owner of a database element, you have the ability to pass on some of your powers when you grant access. If you add the phrase **WITH GRANT OPTION**, then whoever just received the privileges you specified can also pass those on to someone else. For example, you could grant SELECT (read) privilege on a table WITH GRANT OPTION to the head of the marketing department. That person could then give read access to the other employees in the marketing department.

Figure 10.21

SQL security commands. Most systems also provide a visual tool to assign and revoke access rights.

```
GRANT privileges
ON objects
TO users

REVOKE privileges
ON objects
FROM users
```



ItemID | Description | Price | QOH
111 | Dog Food | 0.95 | 53
CustomerID | LastName | FirstName | Phone
1111 | Wilson | Peta | 2222
Jackson | 3333
Jennifer | 4444
SalesID | SaleDate | CustomerID
111 | 03-May- | 1112
112 | 04-May- | 1112
113 | 05-May- | 1113
Items: SELECT
Customers: SELECT, UPDATE
Sales: SELECT, UPDATE, INSERT
Role: SalesClerk
Assign permissions to the role.
New hire: Add role to person

Figure 10.22

Database roles. Create a role and assign permissions to the role. When an employee is added or released, you simply assign or remove the desired roles to instantly provide the appropriate permissions.

Database Roles

Imagine the administrative hassles you would face if you have to assign security permissions for hundreds or thousands of employees. Each time an employee is hired or leaves, someone would have to assign or remove rights to possibly hundreds of tables and views. The task would be time consuming and highly error prone. Even if you build SQL procedures to help, it would require almost constant maintenance. A far more effective solution is to define database roles. A **role** is often associated with a group of users and consists of a set of permissions that are assigned together. As shown in Figure 10.22, you create a role and assign the desired permissions to the role instead of to the individual users. Then you assign the role to the specific users. When a new employee is hired, adding the role to that user provides all of the necessary permissions. The SQL 99 standard supports the ability to assign roles to other roles. For instance, you could define smaller sets of tasks as roles (sell item, add customer, update inventory) and then assign those tasks to broader roles (sales clerk, inventory clerk, and so on).

Queries as Controls

With many systems, the basic security commands are powerful but somewhat limited in their usefulness. Many systems only grant and revoke privileges to an entire table or query. Although the SQL 2003 standard supports specifying columns in the GRANT and REVOKE commands, this option might not be available in all situations. Also, queries provide detailed control through the WHERE clause. The true power of a database security system lies in the ability to assign access to individual queries. Consider the example in Figure 10.23. You have an Employee table that lists each worker's name, phone number, and salary. You want to use the table as a phone book so employees can look up phone numbers for other work-

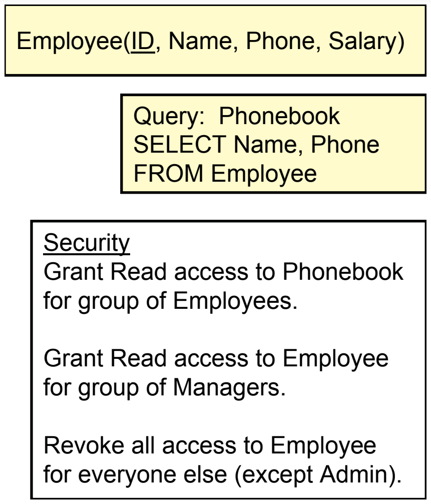

Employee(ID, Name, Phone, Salary)
Query: Phonebook
SELECT Name, Phone
FROM Employee
Security
Grant Read access to Phonebook for group of Employees.
Grant Read access to Employee for group of Managers.
Revoke all access to Employee for everyone else (except Admin).

Figure 10.23

Security using queries. You wish to let all employees look up worker phone numbers. But employees should not be able to see salaries. Define a query that contains only the data needed, and then give employees access to the query—not to the original table.

ers. The problem is that you do not want employees to see the salary values. The solution is to create a query that contains just the name and phone number. Remember that a query does not duplicate the data—it simply retrieves the data from the other tables. Now, assign the SELECT privilege on the Phonebook query to all employees and revoke all employee privileges to the original Employee table. If you want, you can also choose specific rows from the Employee table. For example, you might not want to display the phone numbers of the senior executives.

Chapters 4 and 5 showed you the power of queries. This power can be used to create virtually any level of security that you need. Almost all user access to the database will be through queries. Avoid granting any access directly to a table. Then it will be easier to alter the security conditions as the business needs change.

The basic process is to confer with the users and to determine exactly which type of access each user needs to the data. In particular, determine which users need to add or change data. Then create the users and assign the appropriate security conditions to the queries. Be certain to test the application for each user group. If security is a critical issue, you should consider assigning a couple of programmers to "attack" the database from the perspective of different users to see whether they can delete or change important files.

As a reminder from Chapter 8, a key issue in modern databases—particularly Web-based applications—is the problem of SQL injection. Injection attacks arise when queries are created that use data entered by users—particularly uncontrolled users on the Web. The worst cases arise when SQL queries are built as strings by appending data entered on Web forms. A process should be established for reviewing all SQL statements created in applications that use Web forms. Ensure that they use parameters and remove quote and comment characters from parameter values whenever possible.

Division of Duties

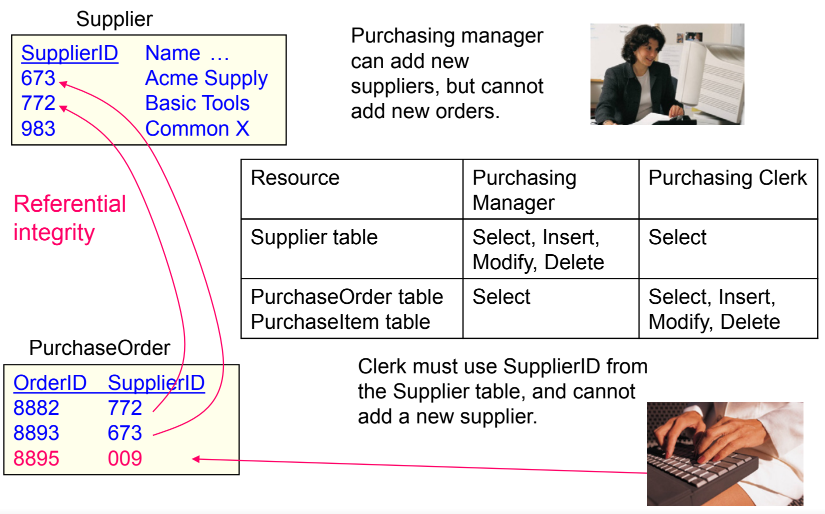
For years, security experts have worried about theft and fraud by people who work for the company. Consider a classic situation that seems to arise every year. A "bad" purchasing manager sets up a fake supplier. The manager orders supplies from the fake company, and pretends that shipments arrive and authorizes payments. Of course, the manager cashes the payment checks. Some of these frauds run for several years before the perpetrators are caught. Ultimately, it is a weak scam because eventually the manager will be caught. Nonetheless, you want to stop the problem and catch the thief as quickly as possible.

The standard method to avoid this type of problem is to divide the duties of all the workers. The goal is to ensure that at least two people are involved in any major financial transaction. For example, a purchasing manager would find new suppliers and perhaps issue purchase requisitions. A different person would be in charge of receiving supplies, and a third person would authorize payments. The goal of separating duties can be challenging to implement. Companies try to reduce costs by using fewer employees. Business picks up, and someone takes advantage of the confusion. It is impossible to eliminate all fraud. However, a well-designed database application can provide some useful controls.

Consider the purchasing example shown in Figure 10.24 in which the basic tables include a Supplier table, a SupplyItem table, a PurchaseOrder table and a PurchaseItem table. In addition, financial tables authorize and record payments. The key to separation of duties is to assign permissions correctly to each table. The purchasing manager is the only user authorized to add new rows to the Supplier table. Purchasing clerks are the only users authorized to add rows to the PurchaseOrder and PurchaseItem tables. Receiving clerks are the only users authorized to record the receipt of supplies. Now if a purchasing clerk tries to create fake orders, he or she will not be able to create a new supplier. Because referential

Figure 10.24

Division of duties. The clerk cannot create a fake supplier. Referential integrity forces the clerk to enter a SupplierID from the Supplier table, and the clerk cannot add a new row to the Supplier table.


Supplier
Referential integrity
PurchaseOrder
Purchasing manager can add new suppliers, but cannot add new orders.
Clerk must use SupplierID from the Supplier table, and cannot add a new supplier.

SupplierID	Name …
673	Acme Supply
772	Basic Tools
983	Common X

Resource	Purchasing Manager	Purchasing Clerk
Supplier table	Select, Insert, Modify, Delete	Select
PurchaseOrder table PurchaseItem table	Select	Select, Insert, Modify, Delete

OrderID	SupplierID
8882	772
8893	673
8895	009

integrity is enforced between the Order table and the Supplier table, the clerk cannot even enter a false supplier on the order form. Likewise, payments will be sent only to legitimate companies, and a purchasing manager will not be able to fake a receipt of a shipment. The power of the database security system is that it will always enforce the assigned responsibilities.

Software Updates

Modern software is large and complex. Software vendors and other researchers often find security problems after the software has been released. Typically, the company has a process to evaluate security threats, create patches to fix the problems, and notify users to update their systems. The difficulty is that announcing the security flaw to users also means that potential hackers are alerted to the problem. As long as all users patch their systems immediately, these announcements would be effective. But as a database administrator, the issue is not that simple. Even if you receive the notice, you still need to test the patch and ensure that it does not cause problems with your applications. Several major attacks have been launched against companies by attackers using known security holes that were not patched.

Consequently, a major task of database administrators is to monitor security releases for the DBMS and operating system software. These patches need to be installed on test machines and moved to the production systems as soon as possible. In the meantime, the network administrators can block specific ports and prevent outsiders from accessing features on the database. The DBAs and the network administrators also have to carefully monitor the network and the servers to see if rogue processes have been started. Most vendors have an automated system so you can discover which patches need to be installed; and even install them automatically. Of course, you must still monitor the process, and choose an update time that does not interfere with operations. You should also perform backups before updating your system.

Encryption

How do you prevent eavesdroppers or hackers from reading data? Encryption is a method of modifying the original information according to some code so that it can be read only if the user knows the decryption key. Encryption should be used when transmitting information from one computer to another—particularly when using the Internet. Sensitive information, particularly passwords and credit card numbers, stored within a database also can be encrypted. Without the encryption key, the files are gibberish. Encryption is critical for personal computer-based systems that do not provide user identification and access controls. Encryption is a useful technique, but you need to be aware of issues involving the keys.

Two basic types of encryption are commonly used today. Most methods use a single key to both encrypt and decrypt a message. For example, the **advanced encryption system (AES)** method uses a single key. Although AES is a U.S. standard, versions of it are available throughout the world because it is based on Rijndael created by two Belgian cryptographers. The AES algorithm is fast and supports key lengths of 128, 192, and 256 bits—protecting it from a **brute force attack** that tries all possible key values. Note that adding one bit to the key length provides twice as many keys; for example, moving from 56 to 128 bits adds 2 to the 72nd power more possibilities to test, making it virtually impossible to attack with brute force—using today's computers. Figure 10.25 shows a basic use

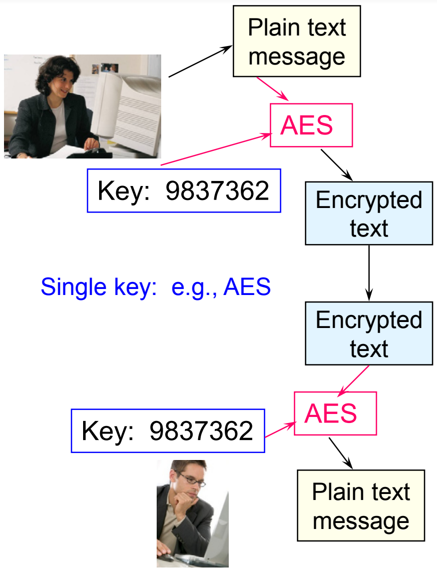


Figure 10.25

Single-key encryption. The same key is used to encrypt and decrypt the message. Distributing and controlling access to keys becomes a major problem when several users are involved.

of the AES encryption method. The primary drawback to AES is that it requires both parties to have the same value of the encryption key. Safely distributing the correct encryption key to each participant is a major problem in protecting data encrypted with a single key.

To solve the key-distribution problem, a second method was created that uses both a **private key** and a **public key**. Whichever key is used to encrypt the message, the other key must be used to decrypt it. The **Rivest-Shamir-Adelman (RSA) algorithm** is an example of a method that uses two keys. RSA protection is available on a variety of computers. RSA encryption works because of the properties of prime numbers. In particular, it is relatively easy for computers to multiply two prime numbers together. Yet it is exceedingly difficult to factor the resulting large number back into its two component parts. The security of the RSA approach relies on using huge numbers (128 digits or more), which would take many years to factor with current technology.

Methods that use two keys have some interesting uses. The trick is that everyone knows your public key, but only you know the private key. Consider the situation in Figure 10.26, where Bob wants to send a database transaction to Alice across the Internet. Bob looks up Alice's public key in a directory. Once the message is encrypted with Alice's public key, only her private key can decrypt it: No one else can read or change the transaction message. The dual-key approach solves the key-distribution problem because anyone can have access to the public key. The tradeoff is that dual-key encryption and decryption is considerably slower than single-key encryption. Consequently, many systems use dual-key encryption to establish a connection and distribute a one-time AES key to use for the session. Also, keep in mind that encryption only prevents people from reading or changing data, someone could still destroy the message before Alice receives it.

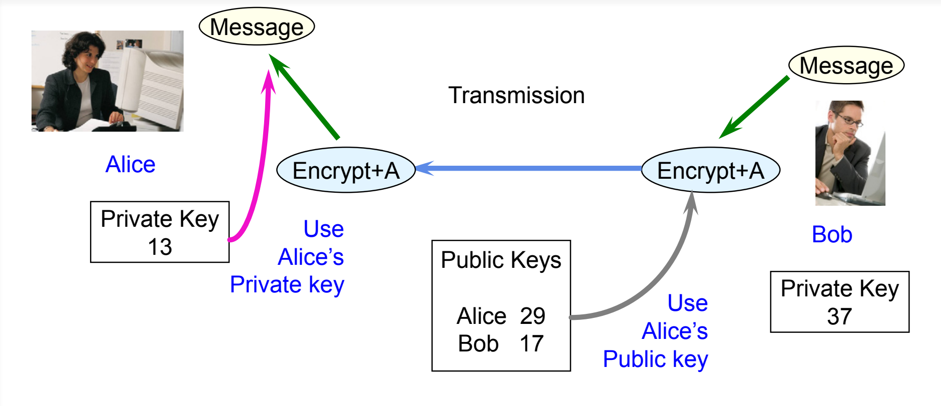


Figure 10.26

Dual-key encryption. Bob sends a message to Alice. Alice publishes here public key, which Bob uses to encrypt the message. Only Alice can read the message, because she is the only one who has access to her matching private key.

There is a second use of dual-key systems called **authentication**. Let's say that Bob wants to send a message to Alice. To make sure that only she can read it, he encrypts it with her public key. However, Bob is worried that someone has been sending false messages to Alice using his name, and he wants to make sure that Alice knows the message came from him. If Bob also encrypts the message with his private key, it can be decrypted only with Bob's public key. When Alice receives the message, she applies her private key and Bob's public key. If the message is readable, then it must have been sent by Bob.

Dual-key encryption systems are useful in all aspects of information communication. They do have one complication: The directory that lists public keys must be accurate. Think about what would happen to authentication if someone impersonated Bob and invented a private and public key for him. This interloper would then be accepted as Bob for any transaction. Hence, the public keys must be maintained by an organization that is trusted. Additionally, this organization must be careful to verify the identity of anyone (individual or corporation) who applies for a key. Several companies have begun to offer these services as a **certificate authority.** One of the leading commercial firms is Verisign, but GoDaddy is cheaper.

For internal use, it is relatively easy to set up a company server that functions as a certificate authority. With this approach, you can generate internal security certificates that will protect transmissions among employees and internal database applications. This approach is significantly less expensive than purchasing annual certificates for every employee. Of course, your certificates will probably not be accepted by anyone outside the company, so you will still need to obtain commercial certificates for applications that deal with external firms and people. Web browsers automatically use dual-key encryption on a **secure sockets layer (SSL)** connection. Check out the options in your browser and you will find a list of known certificate authorities. Certificates issued by these companies will be automatically accepted by the browser.

Encrypting data within a database has one additional twist: Where do you store the decryption key? The purpose of encrypting data within a database is to protect it in case someone gets access to or steals the entire database file. It is particularly useful at protecting sensitive data such as credit card numbers or taxpayer IDs from browsing by employees or other insiders. However, you still have to find a place to store the decryption key so that your application can run, but someone who gets access to the database will not be able to find the decryption key. This problem must be addressed for both types of encryption. Modern operating systems are beginning to offer new options for storing decryption keys. For example, Microsoft systems now provide an encrypted data store specifically for handling encryption keys and certificates. You can find whitepapers on Microsoft's Web site that explain the options. One of the easier-to-use options is to protect the keys using special security certificates that are installed on the server. In SQL Server, you first run CREATE MASTER KEY ENCRYPTION to define a secure area within the database to hold certificates. Then you create certificates that are used to encrypt and decrypt the data. Oracle includes the DBMS_CRYPTO package and sells a Transparent Data Encryption (TDE) system.

Another choice is to store the decryption key in a separate operating system file that has system security rights to allow only a special process to access the file. All of the methods increase the complexity of the application, but they do make it more difficult for someone to read the plaintext data. Over time, operating system vendors will likely add new features that are more secure and easier to use.

The most important rule to follow with encryption is that you should never attempt to create your own encryption method. Always use the tools that are built into the DBMS or operating system. They have been tested and verified by thousands of people.

Sally's Pet Store

What security conditions would be needed at Sally's Pet Store? It is often easier to understand security issues through an example. The first step in assigning security permissions for Sally's Pet Store is to identify the various groups of users. The initial list is shown in Figure 10.27. As the company grows, there will eventually be additional categories of users. Note that these are groups and that several people might be assigned to each category.

Figure 10.27

Initial list of user groups for Sally's Pet Store.

- Management
    - Sally/CEO
- Sales Staff
    - Store manager
    - Salespeople
- Business Alliances
    - Accountant
    - Attorney
    - Suppliers
    - Customers

- Products
    - Sales
    - Purchases
    - Receive products
- Animals
    - Sales
    - Purchases
    - Animal health care
- Employees
    - Hiring/release
    - Hours
    - Paychecks
- Accounts
    - Payments
    - Receipts
    - Management reports

Figure 10.28

Primary operations at Sally's Pet Store. All of these transactions will have forms or reports built into the database.

The second step is to identify the operations that various users will perform. Separate forms will be designed to support each of these activities. Figure 10.28 contains a partial list of the major activities.

The user and group accounts need to be created within the operating system and within the DBMS. After the tables, queries, and forms are created, the DBA should make sure that only the DBA should be able to read or modify data. Go through each operation and identify the queries and tables needed to perform the operation. You should list the permissions for each user group that are required to complete the operation. Figure 10.29 presents the permissions that would be needed to purchase items from suppliers. Notice that only the store managers (and the owner) can order new merchandise. (Insert permission on the MerchandiseOrder and OrderItem tables.) Also note that only the owner can add new suppliers. Remember that a referential integrity constraint is in place that forces the MerchandiseOrder table to use only Suppliers already listed in the Supplier table. Therefore, a store manager will not be able to invent a fictitious supplier. Also note that you would like to permit store managers to add items to the OrderItem table, but they should not be able to alter the order once it has been completed. The DBMS might not support this restriction, and you probably have to give the managers Write permission as well. If available, the distinction would be useful. Otherwise, a manager in charge of receiving products could steal some of the items and change the original order quantity. If Sally has enough managers, this problem can be minimized by dividing the duties and having one manager place orders and another manager record the shipments. Also note that Sally wants to record the identity of the employee who placed the order. For this purpose, you need only read permission on the EmployeeID and Name columns. This privilege can be set by creating a separate EmployeeName query that only retrieves a minimal number of columns from the Employee table. Then you can use this query for purchases instead of the original Employee table.

Purchase	Purchase Query				PurchaseItem Query	
	Merchandise Order	Supplier	Employee	City	Order Item	Merchandise
Sally/CEO	SIUD	SIUD	SIUD	SIUD	SIUD	SIUD
Store Mgr.	SIUD	S*	SIUD	SIUD	I	SIUD
Salespeople	S	S*	S: ID, Name	S	S	S
Accountant	S	S*	S: ID, Name	S	S	S
Attorney	-	-	S: ID, Name	-	-	-
Suppliers	S	S*	-	S	S	S
Customers	-	-	-	-	-	S

S: Select, I: Insert, U: Update, D: Delete
* Basic suppler data: ID, Name, Address, Phone

Figure 10.29

Permissions for purchases. Notice that only the owner can add new suppliers, and only top-level managers can create new orders.

You follow a similar process to identify the access rights for all of the other tables. It is critical that you test all of the security conditions. Best practices dictate that you lock down the permissions so almost no one can access the data. Then, add permissions so that each group can complete assigned tasks. If the actual permissions are altered, you should go record the new values on the charts. The displays make it easier to see if some group has too much access.

Summary

Several steps are involved in managing a database. The DA performs management tasks related to design and planning. Key priorities are establishing standards to facilitate sharing data and integrating applications. The DA also works with users and business managers to identify new applications. In contrast, the DBA is responsible for installing and maintaining the DBMS software, defining databases, ensuring data is backed up, monitoring performance, and assisting developers.

Each stage of application development involves different aspects of database management. Planning entails estimating the size and approximate development costs. Project management skills and teamwork are used in the design stage to split the project and assign it to individual workers. Implementation requires establishing and enforcing development standards, testing procedures, training, and operating plans. Once the application is operational, the DBA monitors performance in terms of space and processing time. Physical storage parameters and other attributes are modified to improve the application's performance.

Backup and recovery are key administrative tasks that must be performed on a regular basis. Backup is more challenging on systems that are running continuously. The DBMS takes a snapshot and saves the data at one point in time. All changes are saved to a journal, which is also backed up on a regular basis. If the system has to be recovered, the DBMS loads the snapshot and then integrates the logged changes.

Security is an important issue in database management. Physical security consists of problems that involve the actual equipment, such as natural disasters or physical theft of hardware. Data backup and disaster planning are the keys to providing physical security. Logical security consists of protecting data from unauthorized disclosure, unauthorized modification, and unauthorized withholding. The first step to providing logical security is to create a system that enables the computer to identify the user. Then application designers and users must determine the access rights that should be assigned to each user. Access rights should be assigned to enforce separation of duties.

Encryption is a tool that is often needed to protect databases. Encryption is particularly useful when the operating system cannot protect the database files. Encryption is also used when data must be transmitted across networks—particularly open networks like the Internet.

A Developer's View

Miranda will quickly see that the tasks of a DBA are different from those of a developer, yet the developer must work closely with the DBA. As a developer, you need to understand the importance of data standards. You also need to work with the DBA in planning, implementing, and maintaining the database application. Before implementing the application, you need to establish the database security rights and controls. For your class project, identify all users and determine their access rights. Use queries to give them access only to the data that they need. Test your work. Also, run any performance monitors or analysis tools.

Key Terms

24-7
advanced encryption standard (AES)
authentication
behavioral security
brute force attack
certificate authority
cold site
computer-aided software engineering (CASE)
data administration
data administrator (DA)
database administration
database administrator (DBA)
disaster plan
encryption
GRANT
hot site
logical security
metadata
network attached storage (NAS)
physical security
private key
public key
redundant array of independent drives (RAID)
REVOKE
Rivest-Shamir-Adelman (RSA) algorithm
role
schema
secure sockets layer (SSL) connection
shell site
snapshot
storage area network (SAN)
tablespaces
transaction log
tuning
virtual machine (VM)
WITH GRANT OPTION

Review Questions

1. What is the role and purpose of a data administrator?
2. What tasks are performed by a database administrator?
3. What tools are available to monitor database performance?
4. ✓ What is metadata and how do DBAs determine the structure of tables in a database?
5. How does the DA facilitate teamwork in developing database applications?
6. What are the primary security threats to a business?
7. How do DBMSs backup data that is constantly changing?
8. ✓ How does hardware provide real-time backup to databases?
9. What are the three problems faced by logical security systems?
10. How does a DBMS identify a user?
11. What are the basic database privileges that can be assigned to users?
12. How do queries provide detailed access controls?
13. How does a good DBMS application provide for division of duties?
14. ✓ How is encryption used to protect data sent to Web sites?
15. How do virtual machines and storage area networks improve administration and security?

Exercises

1. Use online resources to find three job openings for a DBA. Summarize the expected salary and the experience level required for the jobs.
2. You are working on a new project that has the following primary tables with the estimated number of rows. Provide a rough estimate of the size of the database after three years of operation. Pick a DBMS and identify the components needed and try to estimate the cost upfront and annual software licensing costs

Table	Initial size (gigabytes)	Annual Growth rate
Customers	10	10%
Employees (with photo)	12	5%
Item/Package (per year)	625	10%
Tracking (per year)	5,000	10%

3. Briefly describe how you would protect a computer system from the following problems. List steps you will take before the event and after the event has occurred.
   a) Your local electricity provider expanded into wind and solar power and now the weather is bad and the company has insufficient power during peak days so you have rolling brownouts (no power) for up to an hour every three days.
   b) Employees sneaking cigarette breaks behind the building somehow caused a fire that burned down the computer center.
   c) Your CEO managed to anger a couple of hacker groups and your Web servers are now facing a huge distributed denial of service attack so no one can get access.
   d) The North Korean government created a spyware/attack program targeted to your production databases, placed it on a USB drive and dropped a dozen of them in your parking lot. An employee plugged it in an infected your computers, destroying the databases (and some production equipment).
   e) The FBI just appeared at your door and said the person you hired as one of your DBAs a year ago was just caught at the airport with a ticket to China, smuggling a USB drive with what looks to be 80 percent of your main database contents.
   f) As part of a lawsuit against the company, an ex-employee is accusing an HR worker of looking up salaries in the company payroll database and telling other people the numbers.

4. Research the methods of encrypting a database column for a specific DBMS used to connect to a Web application. In particular, describe how the key is stored and comment on what it would take to obtain the key.
5. Research the costs of running SQL Server (or Oracle if you prefer) on a cloud server hosted by a third party. Assume the size of the database is about 50 GB with about 400 GB of network data traffic a month.

6. You are setting up a database for a local government agency to handle its purchases, including electronic payments to suppliers. Define the user groups and access rights to the following tables:

```
Supplier(SupplierID, Name, Address, Phone, City, State,
ZIP, BankAccount)
PurchaseOrder(OrderID, SupplierID, OrderDate, EmployeeID,
DateDue, DatePaid)
OrderItem(OrderID, ItemID, Description, Quantity, Price,
DateReceived)
Payment(PaymentID, OrderID, PaymentDate, BankConfirm,
Amount)
```

7. You are setting up a database for a company that sells products over the Web. Define the access rights to four groups of users: Customers, Shipping Clerks, Marketing Manager, and Customer Service; on the following tables:

```
Customers(CustomerID, Name, Address,  Gender, Phone,
Email, …
Items(ItemID, Description, Photo, ListPrice, Category
Sale(SaleID, SaleDate, CustomerID, IPAddress
SaleItem(SaleID, ItemID, Quantity, SalePrice,
CustomerComments
Returns(ReturnID, ReturnDate, SaleID, ItemID, Quantity,
Reason, Comments
```

8. Employees and other insiders present the greatest security problems to companies. Outline basic policies and procedures that should be implemented to protect the computer systems. (Hint: Research employee hiring procedures.)
9. Research the current status of RAID drives versus SSDs for storing database files. What the benefits and costs to each method?
10. Write a metadata query to retrieve a list of all stored views that use a table named Customer. Hint: Try the SQL Standard, or use your favorite DBMS to test it.

Sally's Pet Store

11. Devise a security plan for Sally's Pet Store. Identify the various classes of users and determine the level of access required by each group. Create any queries necessary to provide the desired security.
12. If it does not already exist, create a sales query that uses data from the Customer, Sales, SaleItems, Employee, and City tables to produce a report of all sales sorted by state. Use a query analyzer to evaluate the query and identify methods to improve its performance.
13. Create a backup and recovery plan that will be used at Sally's Pet Store. Identify the techniques used, who will be in charge, and the frequency of the backups. Explain how the process will change as the store and the database grow larger.

14. What physical security controls will be needed to protect the database and hardware?

15. Assume the database security system has user groups for SalesClerks and Managers. Write the GRANT commands to let sales clerks record a new sale but only managers can make changes to the sale.

16. Assume the Pet Store grows bigger (but still a single store), to include 10 checkout counters including a grooming salon service, handling several hundred customers and thousands of items sold each day. Where would you expect to see bottlenecks in the application? How could performance and security be improved?

17. The Pet Store owner/manager wants to be able to connect to the database while traveling or at home. What security precautions would be needed to keep this connection safe?

18. What data tables or transactions would you want to monitor on a regular basis to help indicate if a crime or attack is begin committed against the system?

Rolling Thunder Bicycles

19. Devise a security plan for Rolling Thunder Bicycles. Identify the various classes of users and determine the level of access required by each group. Create any queries necessary to provide the desired security.

20. Devise a backup and recovery plan for Rolling Thunder Bicycles. Be sure to specify what data should be backed up and how often. Outline a basic disaster plan for the company. Where are security problems likely in the existing application? How should duties be separated to improve security?

21. Use the performance analyzer tools available for your DBMS to evaluate the tables, queries, forms, and reports. Provide an explanation of the top five recommendations.

22. The company is planning to set up a Web site to enable customers to enter and track their orders using the Internet. Explain the additional security procedures that will be needed.

23. The managers want to expand the employee information in the database, such as adding sick days, vacation leave, and some benefits information. For now, these would be simple columns in the Employee table. How can security be assigned so that each employee can see his or her personal data but not that of other employees?

24. Using the existing data, estimate the size and monthly transaction volume/data traffic. Use this data to estimate the price for hosting the entire database on Microsoft's cloud-based SQL Server database.

25. Use the metadata to find all of the queries/views that reference the Customer table.

26. Which of the elements of the Rolling Thunder Bicycle application are most likely to require updates over time—often because of changes in the database software?

27. What data could most benefit through the use of encryption in this company? How can data be encrypted in Microsoft Access?

Corner Med

28. Specify the access permissions needed at Corner Med. Focus on the two main user groups: clerical and medical staff, but also include a managerial group that reviews financial and treatment summary data—without needing access to individual patient data. The plan should address the privacy implications.

29. The company would like to give wireless devices to the medical staff to give them access to the data while talking with patients. Research the potential security issues and describe a solution that would protect the privacy and security, and remain usable.

30. Describe the physical security precautions that need to be taken. How will users be authenticated? Where should the servers and data be stored? How will backups be handled?

31. Identify the elements of the database that should be stored in encrypted form. Pick a DBMS and research the support available to encrypt individual data elements.

32. Use performance monitoring tools to evaluate the database and identify any important suggestions for improving performance.

33. The company is planning to expand to multiple locations, each with a separate manager and staff, but wants to share the database centrally. Which aspects of the database application are likely to need the most work in terms of performance as the load increases?

34. The company wants to move the entire database to a cloud-based platform (pick one). Assume the company maintains about the same patient load per physician and simply adds 19 more locations. Also, assume the company decides to store images (high-definition camera and x-ray) that average two images per patient visit. Because each file is about 10 MB, the images themselves will be stored as files in the cloud with name links stored in the database. Estimate the cost of running this database on a cloud server, storing data for five years.

35. Write the GRANT statements to let each employee see his or her own Employee records and edit simple columns such as name and phone number (but not salary).

Web Site References

http://www.dama.org	Data management organization.
http://www.aitp.org	Association for Information Technology Professionals organization.
http://www.sigsac.org/	Association for Computing Machinery: Special Interest Group on Security, Audit, and Control.
http://www.cert.org	Internet organization tracking security topics.
http://www.databasejournal.com	Database security and other database management topics.
http://www.oracle.com/pls/db112/docindex/	Oracle Administrator Books Online

Additional Reading

Bertino, E., B. Catania, E. Ferrari, P. Perlasca, A Logical Framework for Reasoning about Access Control Models, *ACM Transactions on Information and System Security (TISSEC)*, 6(1), February 2003, 71-127. [A detailed discussion of various security access complications.]

Bryla, B. and K. Loney, *Oracle Database 11g DBA Handbook*, New York: McGraw-Hill/Oracle Press, 2008. [One of many Oracle administrator books from Oracle Press.]

Carpenter, T., *Microsoft SQL Server 2012 Administration: Real-World Skills for MCSA Certification and Beyond*, Indianapolis: Sybex/Wiley, 2013. [One of many books for studying for certification exams.]

Castano, S. (ed.). *Database Security*, Reading, MA: Addison Wesley, 1994. [Collection of articles from the Association of Computing Machinery.]

Natan, R.B., *Implementing Database Security and Auditing*, Burlington, MA: Elsevier Digital Press, 2005.

Shamsudeen, R., "Oracle Database 12c Review: Finally, a True Cloud Database," Infoworld, June 26, 2013. http://www.computerworld.com/s/article/9240354/. [Overview of improvements in Oracle 12c. Almost all of them are related to distributed databases.]

Chapter 11

Distributed Databases

Chapter Outline

What You Will Learn in This Chapter

- Why do you need a distributed database?
- What are distributed databases?
- How is data distributed with client/server systems?
- Can a Web approach solve the data distribution issues?
- How much data can you send to a client form?
- What benefits are provided by cloud computing and data storage?
- How will Sally's employees access the database?

A Developer's View

Ariel: How is the new job going, Miranda?

Miranda: Great! The other developers are really fun to work with.

Ariel: So you're not bored with the job yet?

Miranda: No. I don't think that will ever happen—everything keeps changing. Now they want me to set up a Web site for the sales application. They want a site where customers can check on their order status and maybe even enter new orders.

Ariel: That sounds hard. I know a little about HTML, but I don't have any idea of how you access a database over the Web.

Miranda: Well, there are some nice tools out there now. With SQL and a little programming, it should not be too hard.

Ariel: That sounds like a great opportunity. If you learn how to build Web sites that access databases, you can write your ticket to a job anywhere.

Getting Started

Databases and applications need to be used in multiple locations. You need to decide where to physically place the DBMS and databases and which data to transfer to users. Four primary methods are used to distribute data: (1) Linked databases, (2) Replicated databases, (3) Web applications, and (3) Cloud computing. You need to understand the strengths and weaknesses of each to choose the best method for any application.

Introduction

Why do you need a distributed database? Today even small businesses have more than one computer. At a minimum they have several personal computers. More realistically, most organizations take advantage of networks of computers by installing portions of their database and applications on more than one computer. As companies open offices in new locations, they need to share data across a larger distance. Increasingly, companies are finding it useful and necessary to share data with people around the world. Manufacturing companies need to connect to suppliers, distributors, and customers. Service companies need to share data among employees or partners. All of these situations are examples of distributed databases. Many applications can take advantage of the network capabilities of the Internet and the presentation standards of the World Wide Web. Even applications that seem simple need to consider some distributed issues. For instance, a basic transaction-processing system, such as Sally's Pet Store might need to access the data from sales terminals as well as several management computers. If the company expands to multiple stores, you would need to decide how to handle the data for the individual stores, yet still combine the information for use by management.

Building applications that function over networks and managing distributed databases can be complicated tasks. The goal is to provide location transparency to the users. Users should never have to know where data is stored. This feature requires a good DBMS, a solid network, and considerable database, network, and security management skills. However, a well-designed distributed database application also makes it easier for a company to expand its operations. On the other hand, databases that run on more than one computer significantly complicate transaction processing.

Increasingly, people are carrying tablets and smart phones to run Web applications. Although business applications are behind social networks and other commercial Web sites, it is likely that eventually most applications will need at least some Web-based capabilities. Certainly, most new applications will be built to run in browsers. With Web-based applications, database content and most application processing is handled on central servers, while user interaction is handled on the portable clients using HTML and JavaScript. In a sense, the database becomes centralized while the user applications are decentralized.

The Internet offers other alternatives for distributed databases. Cloud computing and data storage can be purchased from third-party providers such as Amazon and Microsoft. These services offer the ability to store databases online with fast data transfers and network and computer support provided by experts. But, it can be expensive, so you need to understand the tradeoffs.

Higher speeds in network connections are simplifying the issues of distributed databases. When tablets and cell phones have 40 mbps or higher reliable data connections, it becomes easier to store data and applications centrally.

Two-Minute Chapter

Databases and applications become more complex when users need to access the data from different locations. Locations can be relatively close or they can be thousands of miles apart. Networks are used to connect users to the database and application servers. Local networks can be very fast but still create some design issues for large applications. Wide area networks that require paying for data transfers across relatively slow public networks cause bigger problems with sharing data. The primary design decision is where to locate the data. Today, the two main options are (1) store replicated copies locally and synchronize the copies, or (2) keep the data centralized and use Web servers and applications to connect to users.

Using centralized Web server applications is tempting in many situations because it simplifies the management and control of data. But, despite improvements in networks some applications probably need to remain on local systems. For instance, a checkout systems for retail stores would probably be too slow to run as Web applications. And checkout speed is a critical factor for many retail stores. But network and browser technologies continue to improve so developers have to continue to examine the tradeoffs.

Distributed databases consist of database files stored in different locations, under the control of different copies of a DBMS. Developers need to decide if data should be shared instantaneously across all sites or if replicas should be used that periodically synchronize the data changes. The choice depends on whether all sites need to see exactly the same data. Distributed databases also make it harder to deal with key generation, concurrent data access, and distributed transactions.

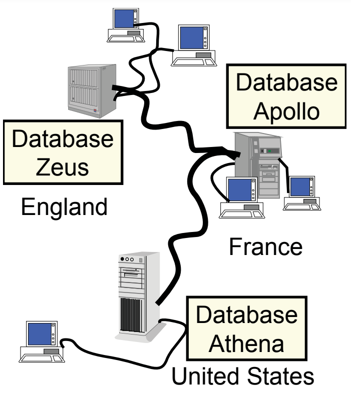


Figure 11.1

Distributed database. Each office has its own hardware and databases. For international projects, workers in different offices can easily share data. The workers do not need to know that the data is stored in different locations.

High-end DBMSs contain mechanisms for handling many of these elements automatically, but they tend to be expensive.

Older client-server systems split applications into the front-end of forms and reports, connected to a back-end database server. Sometimes middleware applications are used to provide business logic and data connectivity between the two layers.

Increasingly, applications are moving to centralized Web servers—perhaps using cloud computing. The data for these sites is centralized but the applications are distributed and available on computers, tablets, and mobile devices through just a Web browser. This approach simplifies the data issues but requires relatively high-speed, and highly-reliable Internet connections to each user.

Distributed Databases

What are distributed databases? ? A **distributed database** system consists of multiple independent databases that operate on two or more computers that are connected and share data over a network. The databases are usually in different physical locations. Each database is controlled by an independent DBMS, which is responsible for maintaining the integrity of its own databases. In extreme situations, the databases might be installed on different hardware, use different operating systems, and could even use DBMS software from different vendors. That last contingency is the hardest one to handle. Most current distributed databases function better if all of the environments are running DBMS software from the same vendor.

In the example shown in Figure 11.1, a company could have offices in three different nations. Each office would have its own computer and database. Much of the data would stay within the individual offices. For example, workers in the United States would rarely need to see the daily schedules of workers in France. On the other hand, workers in France and England could be working on a large international project. The network and distributed database enable them to share data and treat the project as if all the information were in one place.

Distributed databases do not have to be international. They might even exist within the same building. The key part of the definition is that the database runs on different computers—those computers could simply be in different rooms. However, distributed systems where the machines are physically close to each other are much easier to configure. The primary issue in distributed databases is the speed of the network connections. When the machines are physically close, you can easily install high-speed networks at a relatively low cost. When you can transfer huge amounts of data between computers for almost no cost, the distributed issues are easier to solve. On the other hand, you still have to understand the options and plan for potential problems.

The main issue with a distributed database is identifying the data that needs to be shared. Data that is used exclusively within one location, on a single database, is easy to handle—just keep it on the local computer. Data that needs to be accessed from multiple locations is more complex. You have to choose how the data will be shared. For example, you could keep all of the data on one central computer. Or, you could replicate copies to each location. The challenge is to balance the benefits and costs to achieve the performance needed for each application.

Goals and Rules

It is difficult to create a DBMS that can adequately handle distributed databases. (The major issues will be addressed in later sections.) In fact, early systems faced various problems. Consequently, a few writers have created a set of goals or rules that constitute the useful features of a distributed DBMS. C. J. Date, who worked with E. F. Codd to define the relational database approach, lists several rules that he feels are important. This section summarizes Date's rules.

In anyone's definition of a distributed database, the most important rule is that the user should not know or care that the database is distributed. For example, the user should be able to create and run a simple query just as if the database were on one computer. Behind the scenes the DBMS might connect to three different computers, collect data, and format the results. But the user does not know about these steps.

As part of this rule, the data should be stored independently of location. For example, if the business changes, it should be straightforward to move data from one machine and put it in a different office. This move should not crash the entire application, and the applications should run with a few simple changes. The system should not rely on a central computer to coordinate all the others. Instead, each computer should contact the others as needed. This separation improves system performance and enables the other offices to continue operations even if one computer or part of the network goes down.

Some additional goals are more idealistic. The DBMS should be hardware and operating system independent so that when a newer, faster computer is needed, the company could simply transfer the software and data to the new machine and have everything run as it did before. Similarly, it is beneficial if the system runs independently of its network. Most large networks are built from components and software from a variety of companies. A good distributed DBMS should be able to function across different networks. Finally, it is preferable if the distributed application does not rely on using DBMS software from only one vendor. For example, if two companies were to merge, it would be great if they could just install a network connection and have all the applications continue to function—even if the companies have different networks, different hardware, and database software from different vendors. This idealistic scenario does not yet exist.

These features are desirable because they would make it easier for a company to expand or alter its databases and applications without discarding the existing work. By providing for a mix of hardware, software, and network components, these objectives also enable an organization to choose the individual components that best support its needs.

Advantages and Applications

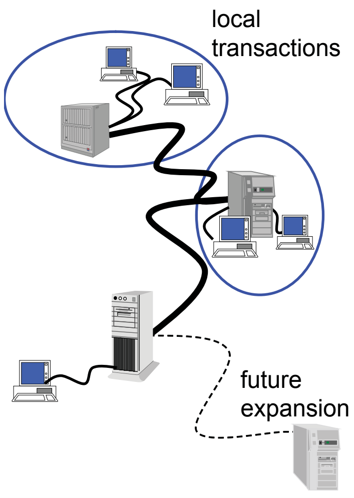
The main strength of the distributed database approach is that it matches the way organizations function. Business operations are often distributed across different locations. For example, work and data are segmented by departments. Workers within each department share most data and communications with other workers within that department. Yet some data needs to be shared with the rest of the company as well. Similarly, larger companies often have offices in different geographical regions. Again, much of the data collected within a region is used within that region; however, some of the data needs to be shared by workers in different regions.

Three basic configurations exist for sharing data: (1) one central computer that collects and processes all data, (2) independent computer systems in each office that do not share data with the others, and (3) a distributed database system.

The first option—a single computer—was the earliest approach to the problem. Interestingly, it is regaining popularity in recent years. Originally, simple terminals were connected to a single, expensive, computer. Today, data can be stored on central servers and accessed via Web browsers from anywhere in the world. Keeping all of the data in one location greatly simplifies coordination and security. Using relatively inexpensive computers as browsers makes it easy to replace them if something breaks.

Figure 11.2

Distributed database strengths. Most data is collected and stored locally. Only data that needs to be shared is transmitted across the network. The system is flexible because it can be expanded in sections as the organization grows.



The second option is a possibility—as long as the offices rarely need to share data. It is still a common approach in many situations. Data that needs to be shared is transmitted via paper reports, e-mail messages, or perhaps text files. Of course, these are ineffective methods for sharing data on a regular basis.

Figure 11.2 illustrates the third option of using a distributed database approach. The main advantage is that distributed systems provide a significant performance advantage through better alignment with the needs of the organization. Most updates and queries are performed locally. Each office retains local control and responsibility for the data in that office. Yet the system enables anyone with the proper authority to retrieve and integrate data from any portion of the company as it is needed.

A second advantage to distributed databases is that, compared to centralized systems, they are easier to expand. Think about what happens if the company is using one large, centralized computer. If the company expands into a new region, requiring more processing capacity, the entire computer might have to be replaced. With a distributed database approach, expanding into a new area would be supported by adding another computer with a database to support the new operations. All existing hardware and applications remain the same. By using smaller computer systems, it is easier and cheaper to match the changing needs of the organization.

Because the distributed database approach can be tailored to match the layout of any company, it has many applications. In a transaction processing system, each region would be responsible for collecting the detailed transaction data that it uses on a daily basis. For instance, a manufacturing plant would have a database to collect and store data on purchases, human relations, and production. Most of this data would be used by the individual plant to manage its operations. Yet as part of the corporate network, summary data could be collected from each plant and sent to headquarters for analysis. As another example, consider a consulting firm with offices in several countries. The workers can store their notes and comments in a local database. If a client in one country needs specialized assistance or encounters a unique problem, the local partners can use the database to search for similar problems and solutions at other offices around the world. The distributed database enables workers within the company to share their knowledge and experiences.

Creating a Distributed Database System

The basic steps to building a distributed database are similar to those for creating any database application. Once you identify the user needs, the developers organize the data through normalization, create queries using SQL, define the user interface, and build the application. However, as shown in Figure 11.3, a distributed database requires some additional steps. In particular, a network must connect the computers from all the locations. Even if the network already exists, it might have to be modified or extended to support the chosen hardware and DBMS software.

Another crucial step is to determine where to store the data. The next section examines some of the issues you will encounter with processing queries on a distributed database. For now, remember that the goal is to store the data as close as possible to the location where it will be used the most. It is also possible to replicate heavily used data so that it can be stored on more than one computer. Of course, then you need to choose and implement a strategy to make sure that each copy is kept up-to-date.

Design administration plan.
Choose hardware, DBMS vendor, and network.
Set up network and DBMS connections.
Choose locations for data.
Choose replication strategy.
Create backup plan and strategy.
Create local views and synonyms.
Perform stress test: loads and failures.

Figure 11.3

Additional steps to creating a distributed database. After the individual systems and network are installed, you must choose where to store the data. Data can also be replicated and stored in more than one location. Local views and synonyms are used to provide transparency and security. Be sure to stress test the applications under heavy loads and to ensure that they handle failures in the network and in remote computers.

Backup and recovery plans are even more critical with a distributed database. Remember that several computers will be operating in different locations. Each system will probably have a different DBA. Yet the entire database must be protected from failures, so every system must have consistent backup and security plans. Developing these plans will probably require negotiation among the administrators—particularly when the systems cross national boundaries and multiple time zones. For example, it would be virtually impossible to back up data everywhere at the same time.

Once the individual systems are installed and operational, each location must create local views, synonyms, and stored procedures that will connect the databases, grant access to the appropriate users, and connect the applications running on each system. Each individual link must be tested, and the final applications must be tested both for connections and for stress under heavy loads. It should also be tested for proper behavior when a network link is broken or a remote computer fails.

Operating and managing a distributed database system is considerably more difficult than a handling single database. Identifying the cause of problems is much more difficult. Basic tasks like backup and recovery require coordination of all DBAs. Some tools exist to make these jobs easier, but they can be improved. Do you remember the rule that a distributed database should be transparent to the user? That same rule does not yet apply to DBAs or to application developers. Coordination among administrators and developers is crucial to making applications more accessible to users.

Network Speeds

The challenge with distributed databases comes down to physics and economics. As illustrated in Figure 11.4, data that is stored on a local disk drive can be transferred to the CPU at transfer rates of 60 to 400 megabytes per second (higher speeds with SSD and RAID drives). Data that is stored on a server attached to a **local area network (LAN)** can be transferred at rates from 10 to 100 megabytes per second (100 to 1,000 megabits per second). Using public transmission lines to connect across a **wide area network (WAN)** provides transfer rates from 0.2 to 300 megabytes per second. To get that 300 megabytes per second (on an OC-

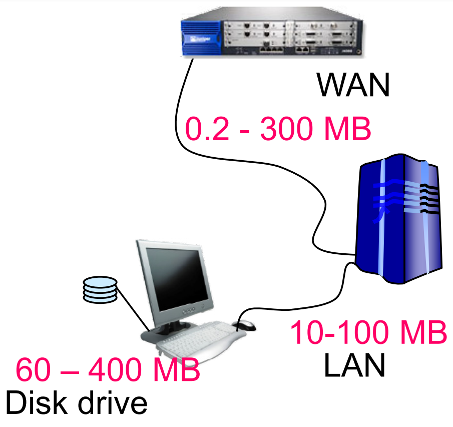


Figure 11.4

Network transfer rates. Network performance is shown here in megabytes per second. Local disks and networks are considerably faster than wide area networks and WAN costs are higher.

48 line at 2488 mbps), your company would probably have to pay over $50,000 a month in network costs. As technology changes these numbers are continually improving. One of the biggest changes in recent years is the performance gains of local area networks. With gigabit performance, it is relatively easy to move data away from the processor and place it on a network attached storage device on a storage area network (SAN). Separating the data from the processor makes it easier to upgrade processors and provide backup facilities—both in terms of backing up the data and replacing servers. Although a SAN offers several benefits to running servers, it does not solve the distributed database problem because the distance is still limited.

Note that most DBMS vendors also sell enterprise versions of the software that can take advantage of a computer cluster. A **cluster** consists of multiple computers that effectively work as a single machine. Because the system uses a single copy of the DBMS, it is not a distributed system. However, it does take advantage of fast LAN speeds and attached storage. The major strengths of clusters are **fault tolerance** and **scalability**. The system automatically balances the load across the servers. If one of the processors fails, the system simply ignores it. The DBA can shut down the failed server and replace it with a new one. Likewise, as the database grows, and you need more processing power, you simply add another machine to the cluster. The system detects the new capabilities and redirects processing needs to the new server.

The real issues of distributed databases arise when you need to connect machines using a wide area network (WAN). Although high-speed WANs are becoming more common, they are still relatively expensive. The goal of distributed processing is to minimize the transfer of data on slower networks and to reduce the costs of network transfers. Part of this goal can be accomplished through design—developers must carefully choose where data should be located. Data should be stored as close as possible to where it will be used the most. However, trade-offs always arise when data is used in several locations.

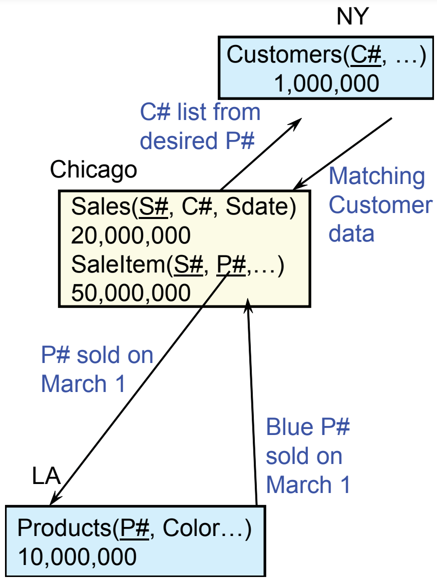


Figure 11.5

Distributed database query example. List customers who bought blue products on March 1. A bad idea is to transfer all data to Chicago. The goal is to restrict each set and transfer the least amount of data.

Query Processing and Data Transfer

Data transfer rates are a key issue in distributed processing. To understand their importance, consider the issue of responding to queries if data is stored in separate locations. If a query needs to retrieve data from several different computers, the time to transfer the data and process the query depends heavily on how much data must be transferred and the speed of the transmission lines. Consequently, the result depends on how the DBMS joins the data from the multiple tables. In some cases the difference can be extreme. One method could produce a result in a few seconds. A different approach to the same query might take several days to process! Ideally, the DBMS should evaluate the query, the databases used, and the transmission times to determine the most efficient way to answer the query.

Figure 11.5 illustrates the basic problem. Consider tables on three different databases: (1) a Customer table in New York with 1 million rows, (2) a Production table in Los Angeles with 10 million rows, and (3) a Sales table in Chicago with 20 million rows. A manager in Chicago wants to run the following query: List customers who bought blue products on March 1. This query could be processed in several ways. First, consider a bad idea. Transfer all of the rows to Chicago; then join the tables and select the rows that match the query. This method results in 11 million rows of data being transferred to Chicago. Even with a relatively fast WAN, anything less than 30 minutes for this query would be fast.

A better idea would be to tell the database in Los Angeles to find all of the blue products and send the resulting rows to Chicago. Assuming only some of the products are blue, this method could significantly cut the number of rows that need to be transmitted. The performance gain will depend on what percentage of rows consists of blue products.

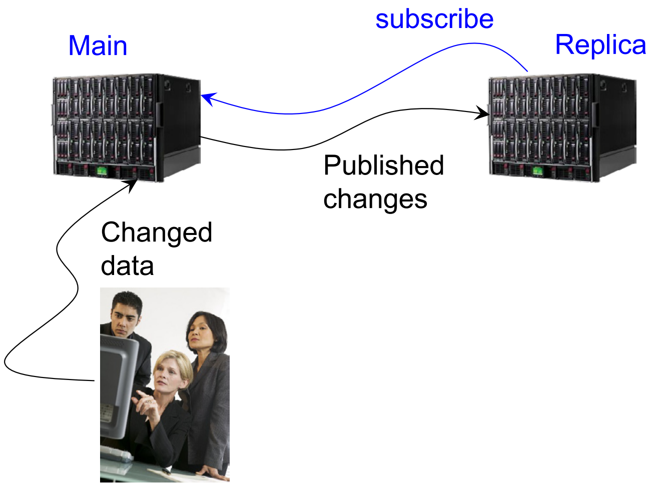


Figure 11.6

Replication with subscribe/publish. The databases are linked by subscribing a replica to the main database. When changes are made on the main, they are published to all of the subscribers.

An even better idea is to get the list of items sold on March 1 from the Chicago table, which requires no transmission cost. Send this list to Los Angeles and have that database determine which of the products are blue. Send the matching CustomerID to the New York database, which returns the corresponding Customer data.

Notice that to optimize the query the DBMS needs to know a little about the data in each table. For example, if there are many blue products in the Los Angeles database and not very many sales on March 1, then the database should send the Sales data from Chicago to Los Angeles. On the other hand, if there are few blue products, it will be more efficient to send the product data from Los Angeles to Chicago. In some cases, the network also needs to know the transfer speed of the network links. A good DBMS contains a query optimizer that checks the database contents and network transfer speeds to choose the best method to answer the query. You still might have to optimize some queries yourself. The basic rule is to transfer the least amount of data possible.

Data Replication

Sometimes there is no good way to optimize a query; or there might be many queries and each requires conflicting optimization methods. When large data sets are needed in several different places, it can be more efficient to **replicate** the tables and store copies in each location. The problem is that the databases involved have to know about each of the copies. If a user updates data in one location, the changes have to be replicated to all the other copies. Two common methods are used to hanlde synchronization: (a) replication mangement, and (2) subscribe/publish connections.

Developers and database administrators can tune the performance by specifying how the database should be replicated. You can control how often the changes

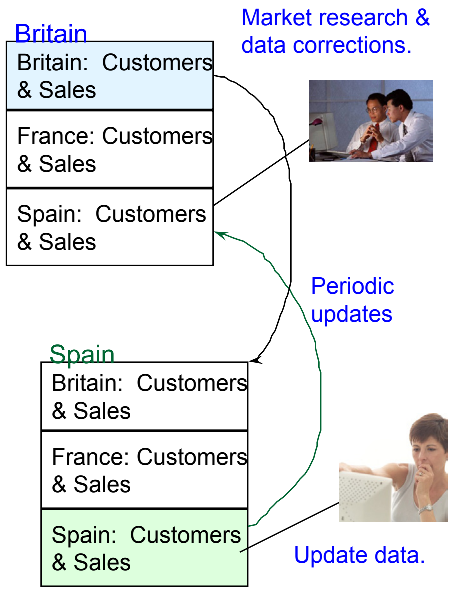

Britain
Britain: Customers & Sales
France: Customers & Sales
Spain: Customers & Sales
Market research & data corrections.
Periodic updates
Spain
Britain: Customers & Sales
France: Customers & Sales
Spain: Customers & Sales
Update data.

Figure 11.7

Replicated databases. If managers do not need immediate data from other nations, the tables can be replicated and updates can be transferred at night when costs are lower.

are distributed and whether they are sent in pieces or as a bulk transfer of the entire table. The biggest difficulty is that sometimes a network link might be unavailable or a server might be down. Then the DBMS has to coordinate the databases to make sure they get the current version of the table and do not lose any changes. Figure 11.6 shows the basic concept of publish/subscribe. Once the subscription connection is established, any changes made to the main database are published to all of the subscribers. In many cases, the changes are sent immediately, which results in a continuous flow of changed data across the network.

With bulk synchronization, most of the database is transferred to a second location, the changes are exchanged, and the new database is returned to the original location. With subscribe/publish, databases that want to be informed of changes create a subscription to a main database. When changes are made on the main copy, they are published and sent to the subscribing databases. In both cases, a replication manager in the DBMS determines which changes should be sent and to handle the updates at each location. Replications can be sent automatically at certain times of the day, sent continuously, or triggered manually when someone feels it is necessary to synchronize the data.

Figure 11.7 illustrates the basic concepts of replication. Marketing offices in each location have copies of Customer and Sales data from Britain, France, and Spain. Managers probably do not need up-to-the-minute data from the other countries, so the tables can be replicated as batch updates during the night. The data will be available to managers in all locations without the managers worrying about transfer time, and the company can minimize international transmission costs by performing transfers at off-peak times.

Transaction processing databases generally record many changes—sometimes hundreds of changes per minute. These applications require fast response times at the point of the transaction. It is generally best to run these systems as distributed databases to improve the performance within the local region. On the other hand, managers from different locations often need to analyze the transaction data. If you give them direct access to the distributed transaction databases, the analysis queries might slow the performance of the transaction system. A popular solution is to replicate the transaction data into a data warehouse. Routines extract data from the transaction processing system and store it in the data warehouse. Managers run applications and build queries to retrieve the data from the warehouse and analyze it to make tactical and strategic decisions. Because the managers rarely make changes to the underlying data, the data warehouse is a good candidate for replication. The underlying transaction processing system retains its speed, and the raw data is not shared. Managers have shared access to the warehouse data.

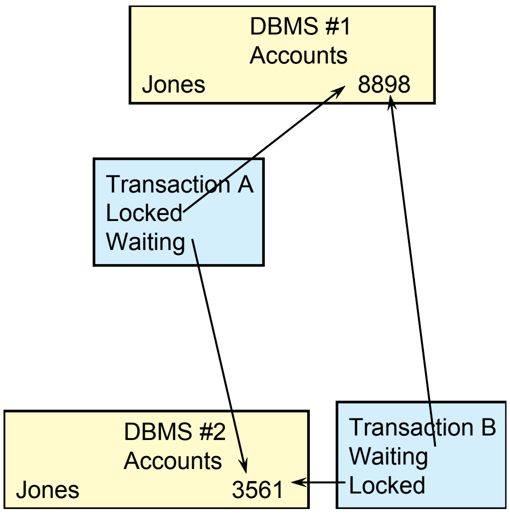
Generating Keys with Replicated Data

Replication seems like an easy solution—each location has a complete copy of the database; performance of local updates and queries is unaffected by the other copies; transactions are completed locally; and data backups are made automatically. In practice, several problems can arise. One problem is the need to generate unique primary key values. A second is the issue of concurrent changes—which is described in the next section.

Automatic key generation is a challenge with replicated databases. What happens if two people in different locations create a new customer? If the key generator is not synchronized, then it is highly likely that both locations will generate the same key, and when the data is updated from the two locations, a collision will occur that must be resolved by hand. Two common methods can be used in dis-

Figure 11.8

Concurrency and deadlock are more complicated in a distributed database. The deadlock can arise across many different databases, making it hard to identify and resolve.



tributed databases to generate keys safely: (1) randomly generated keys, and (2) location-specific keys. Randomly generated keys work if the generator chooses from a sufficiently large number of possible keys. Then there is only a small probability that two keys would ever be generated the same at the same time. To be safe, the generator immediately checks to see if the key just created already exists. The second approach can use either sequential keys or random keys, but it relies on each location being allocated a certain range of values. For instance, one region might be given the range from 1 to 1 million, the next from 1 million to 2 million, and so on. With location-specific generators, you must be careful to isolate the key generation data tables. For example, your key tables would contain the location identifier and the starting or current value of the key. This problem is relatively easy to solve—but you must remember to configure it in your application.

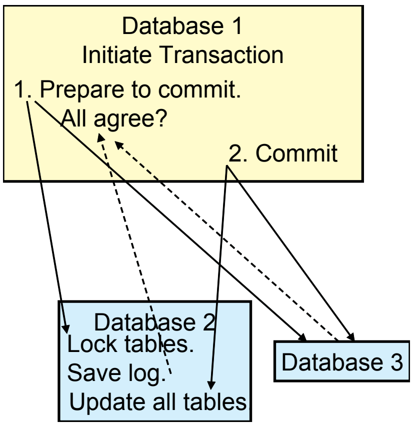
The **globally-unique identifier (GUID)** is often used in distributed databases when a unique value needs to be created. A GUID is essentially a large random number. Microsoft tools use them extensively, and the mechanism for creating them is accessible to most programming tools. The Microsoft algorithm uses the unique ID from the computers' network interface card as part of the GUID, and then adds random digits—for a total of 128 digits. This process ensures that different machines always create different numbers.

Concurrency, Locks, and Transactions

Concurrency and deadlock become complex problems in a distributed database. Remember that concurrency problems arise when two people try to alter the same data at the same time. The situation is prevented by locking a row that is about to be changed. As shown in Figure 11.8, the problem with a distributed database is that the application could create a deadlock that involves different databases on separate computers. One user could hold a lock on a table on one computer and be waiting for a resource on a different computer. Now imagine what happens when there are five databases in five locations. It can be difficult to identify the deadlock

Figure 11.9

Two-phase commit. Each database must agree to save all changes—even if the system crashes. When all systems are prepared, they are asked to commit the changes.

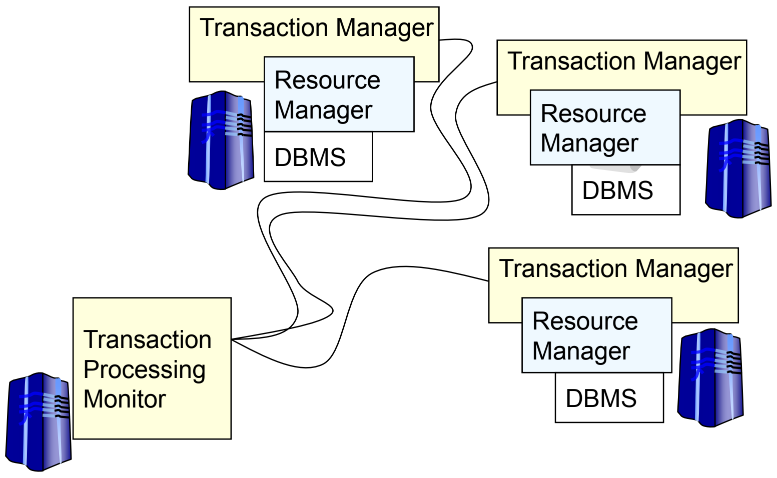


problem. When the locks are on one computer, the DBMS can use a lock graph to catch deadlock problems as they arise. With distributed databases, the DBMS has to monitor the delay while waiting for a resource. If the delay is too long, the system assumes a deadlock has arisen and rolls back the transaction. Of course, the delay might simply be due to a slow network link, so the method is not foolproof. Worse, the time spent waiting is wasted. In a busy system, the DBMS could spend more time waiting than it does processing transactions.

Handling transactions across several databases is also a more complex problem. When changes have to be written to several computers, you still have to be certain that all changes succeed or fail together. To date, the most common mechanism for verifying transactions utilizes a **two-phase commit** process. Figure 11.9 illustrates the process. The database that initiates the transaction becomes a coordinator. In the first phase it sends the updates to the other databases and asks them to prepare the transaction. Each database must then send a reply about its status. Each database must agree to perform the entire transaction or to roll back changes if needed. The local database must agree to make the changes even if a failure occurs. In other words, it writes the changes to a transaction log. Once the log is successfully created, the remote database agrees that it can handle the update. If a database encounters a problem and cannot perform the transaction (perhaps it cannot lock a table), it sends a failure message and the coordinator tells all the databases to roll back their changes. A good DBMS handles the two-phase commit automatically. As a developer, you write standard SQL statements, and the DBMS handles the communication to ensure the transaction is completed successfully. With weaker systems you will have to embed the two-phase commit commands within your program code. If you know that you are building an application that will use many distributed updates, it is generally better to budget for a better DBMS that can handle the two-phase-commit process automatically.

Figure 11.10

Distributed transaction processing monitor. This software handles the transaction decisions and coordinates across the participating systems by communicating with the local transaction managers.



Notice that the two-phase commit system relies on pessimistic locking. Because of transmission delays, it could significantly slow down all of the systems involved in the transaction—as it waits for each machine to lock records. Although optimistic locking might help with some aspects of the transaction, it does not help when a system or communication link fails.

Distributed Transaction Managers

The problem of distributed systems—particularly when database systems are from diverse vendors—is difficult to solve efficiently. One common approach, shown in Figure 11.10, is to use an independent transaction processing monitor or distributed transaction coordinator. This system is a separate piece of software that coordinates all transactions and makes the decision to commit or abort based on interactions with the local transaction managers. This approach is generally provided by the operating system vendor, and the DBMS vendors need to develop interfaces that communicate with the transaction manager. The main transaction manager could run on a separate system, or one of the local transaction managers might be promoted to be the coordinator. For example, Microsoft provides the Distributed Transaction Coordinator, IBM supports Java transactions within its WebSphere Application server, and JBoss Transactions is available independently for UNIX platforms.

Independence is the main strength of the transaction manager. As long as multiple vendors provide support (with the local resource manager software), the system can support diverse products. It is also useful for program-level transactions, where a substantial amount of code is written outside of the databases (e.g., in C++). By relying on the transaction manager, the database system could be changed later if desired—without having to rewrite all of the transaction-processing elements.

Figure 11.11

Design questions. Use these questions to determine whether you should replicate the database, or provide concurrent access to data across the network. Transaction operations are generally run with concurrent access. Decision support systems often use replicated databases. However, the exact choice depends on the use of the data and the needs of the users.

Question	Concurrent	Replication
What level of data consistency is needed?	High	Low – Medium
How expensive is storage?	Medium – High	Low
What are the shared access requirements?	Global	Local
How often are the tables updated?	Often	Seldom
Required speed of updates (transactions)?	Fast	Slow
How important are predictable transaction times?	High	Low
DBMS support for concurrency and locking?	Good – Excellent	Poor
Can shared access be avoided?	No	Yes

Distributed Design Questions

Because of the issues with transmission costs, replication, and concurrency, distributed databases require careful design. As networks gain better transfer rates, database design will eventually become less of a problem. In the meantime you need to analyze your applications to determine how they should be distributed. Figure 11.11 lists some of the questions you need to ask when designing a distributed database. The main point is to determine what portions of the databases should be replicated. If users at all locations require absolute consistency in the database, then replication is probably a bad idea. On the other hand, you might have a weak DBMS that poorly handles locking and concurrency. In this situation it is better to replicate the data, rather than risk destroying the data through incomplete transaction updates.

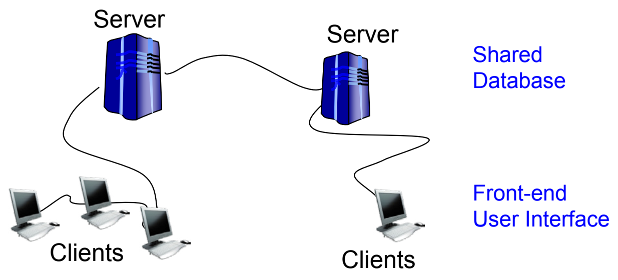
Client/Server Databases

How is data distributed with client/server systems? Many applications run on local area networks using some version of a client/server configuration. This approach is particularly common within retail stores, where checkout registers are based on simple client computers. With this system, the bulk of the data is stored on a centralized server, while the applications run on personal computers. However, some of the data might also be stored on the personal computers, and portions of the application logic might run on middle-tier servers. The client/server approach was driven largely by the limited capabilities of personal computer operating systems. Early operating systems could not support multiple users and provided no security controls. Hence powerful operating systems were installed on servers that handled all the tasks that required sharing data and hardware. The client/server approach is also somewhat easier to manage and control than monitoring hundreds of PCs. Any hardware, software, or data that needs to be shared is stored in a centralized location and controlled by an MIS staff. With the client/server approach, all data that will be shared is first transferred to a server.

As indicated by Figure 11.12, the actual database resides on a server computer. Individual components can be run from client machines, but they store and retrieve data on the servers. The client component is usually a front-end application

Figure 11.12

Client/server system. The client computers run front-end, user interface applications. These applications retrieve and store data in shared databases that are run on the server computers. The network enables clients to access data on any server where they have appropriate permissions.



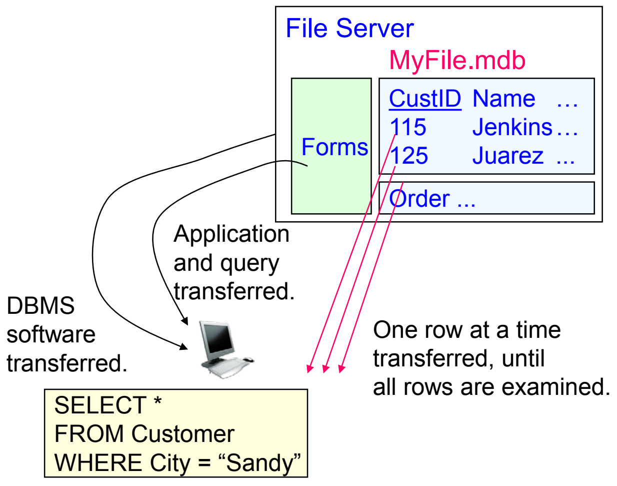


Figure 11.13

File server problems. The file server acts as a large, passive disk drive. The personal computer does all the database processing, so it must retrieve and examine every row of data. For large tables, this process is slow and wastes network bandwidth.

that interacts with the user. For example, a common approach is to store the data tables on a server but run the forms on personal computers. The forms handle user events with a graphical interface, but all data is transferred to the server. You need to understand a few important concepts to design and manage client/server databases. Like any distributed database, where you store the data and how you access it can make a substantial difference in performance. This section also demonstrates some of the tools available to build a client/server database application.

Client/Server versus File Server

To understand the features and power of a client/server database, it is first useful to examine a database application that is not a true client/server database. Initial local area networks were based on file servers. A file server is a centralized computer that can share files with personal computers. However, it does not contain a DBMS. The file server stores files, but to the personal computers it appears as a giant, passive disk drive. The sole purpose of the server is to provide secure shared access to files. The client personal computers do all of the application processing.

Microsoft Access is often used in file server applications. It is relatively easy to split the database and store the data as a file on the server that is accessed by the forms and reports running on personal computers. The file server approach enables you to share a single copy of the data so all users see the same data. However, the file server approach faces some important drawbacks.

Figure 11.13 illustrates the basic problem. The database file is stored on the file server but the DBMS itself runs on the client. Security permissions are set so that each user has read and write permission on the file. The problem arises when your application runs a query. The processing of the query is done on the client computer. That means that the personal computer has to retrieve every row of data from the server, examine it, and decide whether to use it in the computation or display. If the database is small, if the network connection is fast, and if users

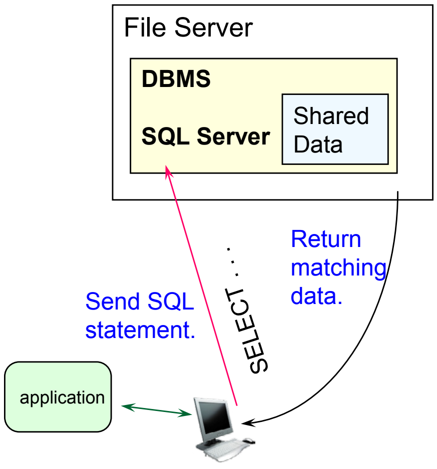


Figure 11.14

Database server. The client computer sends a SQL statement that is processed on the server. Only the result is returned to the client, reducing network traffic.

often want to see the entire table, then this process does not matter. But if the table is large and users need to see only a small portion, then it is a waste of time and network bandwidth to transfer the entire table to the client computer.

The problem is that the file server approach relies on transferring huge amounts of data when the application needs only some of the data. The client/server database approach was designed to solve this problem. With a client/server database, the binary code for the database actually runs on the server. As shown in Figure 11.14, the server database receives SQL statements, processes them, and returns only the results of the query. Notice the reduction in network transfers. The initial SQL statement is small, and only the data needed by the application is transferred over the network. This result is particularly important for decision support systems. The server database might contain millions of rows of data. The manager is analyzing the data and may want summary statistics, such as an average. The server database optimizes the query, computes the result, and transfers a few simple numbers back to the client. Without the server database millions of rows of data would be transferred across the network. Remember that even fast LAN transfer rates are substantially slower than disk drive transfers.

Of course, the drawback to the server database approach is that the server spends more time processing data. Consequently, the server computer has to be configured so that it can efficiently run processes for many users at the same time. Fortunately, processor speeds have historically increased much more rapidly than disk drive and network transfer speeds. The other drawback is that this approach requires the purchase of a powerful DBMS that runs on the server. However, you rarely have a choice. Only small applications used by a few users can be run without a database server.

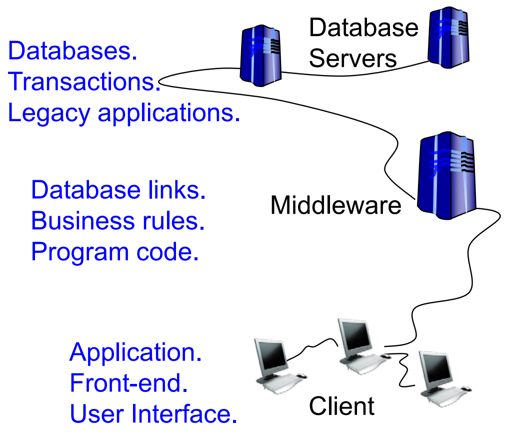


Figure 11.15

Three-tier client/server model. The middle layer separates the business rules and program code from the databases and applications. Independence makes it easier to alter each component without interfering with the other elements.

Three-Tier Client/Server Model

The **three-tier client/server** model has been suggested as an approach that has some advantages over the two-tier model. The three-tier approach adds a layer between the clients and the servers. The three-tier approach is particularly useful for systems having several database servers with many different applications. The method is useful when some of the servers are running legacy applications.

As shown in Figure 11.15, one role of the middle layer is to create links to the databases. If necessary, the middle layer translates SQL requests and retrieves data from legacy COBOL applications. By placing the access links in one location, the server databases can be moved or altered without affecting the client front-end applications. Developers simply change the location pointers, or alter the middleware routines. Some people refer to this approach as n-tier because you can have any number of middle-level computers—each specializing in a particular aspect of the business rules.

Another important role of the middle layer is to host the business rules. For example, creating identification numbers for customers and products should follow a standard process. The routine that generates these numbers should be stored in one location, and all the applications that need it will call that function. Similarly, common application functions can be written once and stored on the middle-layer servers.

This middleware system is well suited to an object-oriented development approach. Common objects that are used for multiple business applications can be written once and stored on the middle servers. Any application can use those objects as needed. As the business rules change or as systems are updated, developers can alter or improve the base objects without interfering with the operations of the applications on the client side. The three-tier approach separates the business rules and program code from the databases and from the applications. The independence makes the system more flexible and easier to expand. Several middleware development tools exist to create and manage objects, but many are proprietary to specific platforms, such as those by Oracle and IBM.

The Back End: Server Databases

Server database systems tend to be considerably more complex and require more administrative tasks than personal computer-based systems. The server environment also provides more options, which makes administration and development more complicated. The DBA must work closely with the system administrator to set up the software, define user accounts, and monitor performance.

Server databases also use trigger procedures to define and enforce business rules. One of the more difficult design questions you must address is whether to store these rules on the back-end database as database triggers and procedures, or move them to a middle-level server using lower-level languages such as C++ or Java. Sometimes you are constrained by the tools and time available. But when possible, you should consider the various alternatives in terms of cost, performance, and expandability.

Placing procedures in the back-end database ensures that all rules are enforced by the DBMS, regardless of how the data is accessed. But, this approach ties you in to a particular DBMS vendor. Because most systems contain proprietary elements, it is difficult to switch to a different DBMS in the future. Placing rules in a middle tier also makes it easier to physically move the database. Generally, the systems are built with reference links to the databases. To move the database, you simply change the reference pointers.

One rule of thumb is to write user-interface code for the client computers and to write data manipulation and control programs to run on the server. Middle-layer programs are used to encode business rules and provide data translation and database independence. The primary objective is to minimize the transfer of data across the network. However, if some computers are substantially slower than others, you will have to accept more data transfers in order to execute the code on faster machines.

The Front End: Windows Clients

Windows-based computers are commonly used as client machines, so Microsoft has created several technologies to provide database connections from the PCs to back-end databases. Various tools and many vendors support the technologies, so they are relatively standardized. The tools have evolved over time as hardware and networks have improved and applications became more complex. Visual Studio is often used as a front-end tool to create the forms and reports. The application is compiled and distributed to user machines, which connect through the Microsoft data components to a back-end server. The PC has a network connection and a database connection that enables it to find the central database on the server. The application code simply selects the appropriate database connection. From that point, your application no longer cares where the data is located—it simply passes SQL requests to the server.

The current Microsoft technology to connect programs to databases is **active data objects (ADO)**. Most DBMS vendors have written ADO connectors, so your application code can retrieve and save data to the most common DBMSs. Whenever you build an application in Visual Studio, you will use ADO to connect to the DBMS. All of the commonly-used commands are embedded in the objects, so you can retrieve data to display it in a form and save changes back to the database with a couple of calls to the object methods. ADO is also used in Microsoft's Web-based applications and the underlying concepts are the same. Java (now supported by Oracle) uses JDBC to connect the language to backend databases.

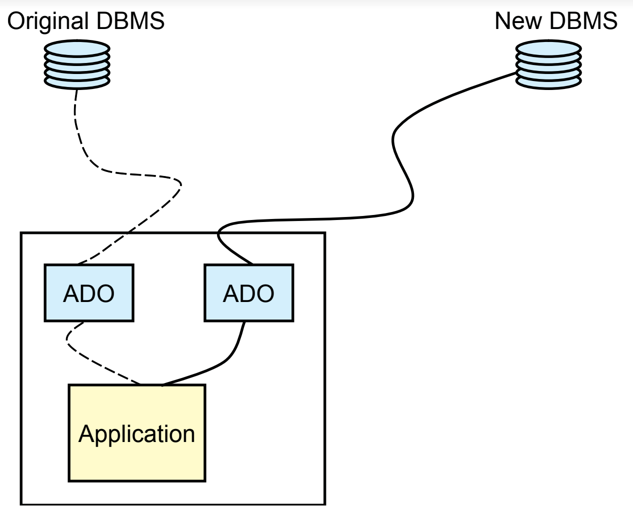


Figure 11.16

Database independence. ADO is a useful buffer between the application and the DBMS. Changing the connection makes it relatively easy to switch the back-end DBMS.

Maintaining Database Independence in the Client

One of the trickiest aspects of distributed databases is the issue of maintaining database independence. When you first build an application, it is often created to run with a single, specified database on the back end. Consequently, it is tempting to simply build the application assuming that the same database will always be there, and use the tools and shortcuts available for that particular system. But what happens later when someone wants to change the back-end database? In extreme cases, the entire application will have to be rewritten. As a developer pressed for time, you might ask why it matters. If someone wants to change the database at a later date, then should they be willing to pay the costs at that time? Yet, with only a little extra effort up front, the application can support most common database systems on the back end; making it easy to change later.

The database connection is one issue in building a generic application. Using a standard such as ADO makes it easier to change databases. Figure 11.16 shows that by changing the connection, your application can connect to a different DBMS. Of course, it is never quite that simple, but the ADO buffer is an important element. In many cases, you can specify the ADO connection string dynamically, making it easy for the application to connect to a different DBMS without rewriting the code. If you are careful, you can build the application so that it can switch DBMS connections at any time. You can build the system using one DBMS and run the production system against a different one.

It is important that you understand that the connection is only one element in making an application DBMS independent. In most situations, the actual SQL commands are a much bigger issue. DBMS vendors tend to provide different levels of support for the SQL standard. They also add proprietary options and commands that are enticing. In particular, vendors offer many variations within the SELECT command. For example, string and date operations are notoriously non-

```
Generic application query:
SELECT SaleID, SaleDate, CustomerID, CustomerName
FROM SaleCustomer

Saved Oracle query:
SELECT SaleID, SaleDate, CustomerID,
          LastName || ‘, ‘ || FirstName AS CustomerName
FROM Sale, Customer
WHERE Sale.CustomerID=Customer.CustomerID

Saved SQL Server query:
SELECT SaleID, SaleDate, CustomerID,
          LastName + ‘, ‘ + FirstName AS CustomerName
FROM Sale INNER JOIN Customer
ON Sale.CustomerID = Customer.CustomerID
```

Figure 11.17

Database query independence. The application contains only simple queries that do not use vendor-specific functions. All detail queries are created and saved within the DBMS.

standardized across vendors. And, if you are using an older version of Oracle or SQL Server, you will not be able to use the INNER JOIN syntax. Because of these differences, a key step in making an application DBMS independent is to move all queries to the DBMS and save them as views. Then your application only contains simple SELECT (or INSERT/UPDATE/DELETE) queries that pull data from the saved view. These simple queries should use only basic standard SQL elements. All of the vendor-specific functions are coded into the query that is saved in the DBMS. To transfer to a new DBMS, you just recreate the queries on the new DBMS using that vendor’s specific tools and syntax. This technique is particularly important for applications that might begin small and grow. At a small size, you might be able to use a small, inexpensive DBMS, but as the number of users grows and demand on the system increases, you will have to scale up to a larger DBMS. If the application queries were carefully built to remain independent, it will be relatively easy to transfer to a new DBMS. Figure 11.17 shows an example of using simple queries to maintain DBMS vendor independence.

Trigger functions are a more complex issue. Some systems do not support triggers at all, and those that do generally provide different functionality. At this point in time, the only method to guarantee compatibility across vendors is to avoid database triggers. Instead, write the same functionality into middleware code that also relies on generic queries.

Centralizing with a Web Server

Can a Web approach solve the data distribution issues? The **World Wide Web** was designed to enable people (initially physicists and researchers) to share information with their colleagues. The fundamental problem was that everyone used different hardware and software. The solution was to define a set of standards. These evolving standards are the heart of the Web. They define how computers can connect, how data can be transferred, and how data can be found. Additional standards define how data should be stored and how it can be displayed. As long

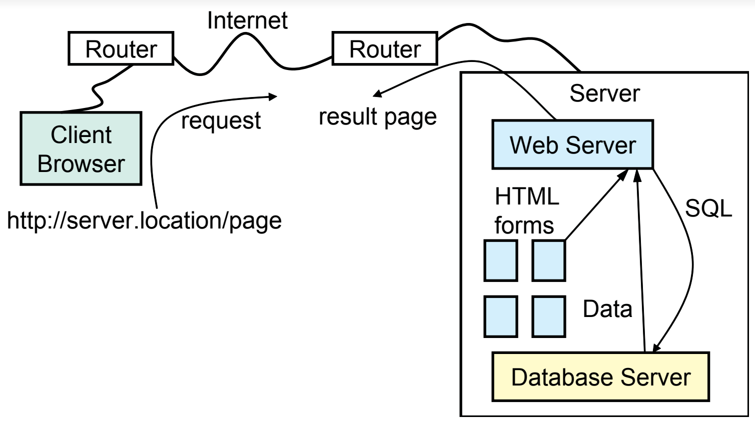


Figure 11.18

Web servers and client browsers. Browsers are standardized display platforms. Servers are accessible from any browser.

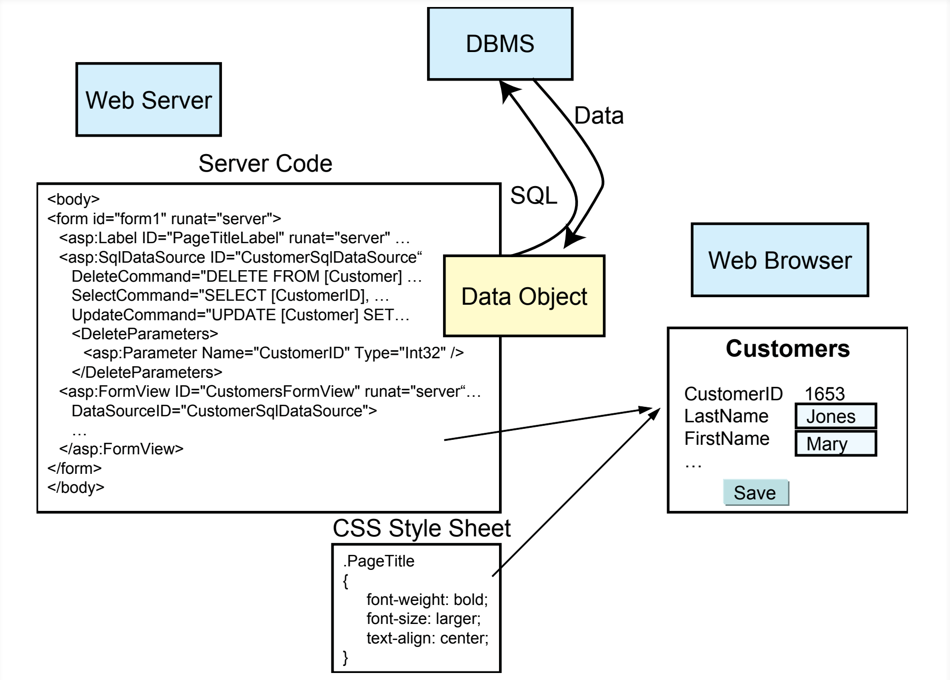
as a computer runs **browser** software that receives and displays data files, it can access and interact with data stored on Web servers. The servers run Web server software that can do almost anything—as long as it formats the data for standard browsers. Both the servers and browsers are becoming more sophisticated, but the essence of the method is presented in Figure 11.18.

An interesting consequence of the rapid acceptance of the Web approach is that it encourages a return to a centralized database. All of the data and applications are stored in one location. Users can be located anywhere in the world—as long as they have a Web browser and an Internet connection. The issues of concurrency and security are simplified again, since everything is handled by one DBMS. The issues of distributed data are minimized, since the data is now stored in one place, and all users share the same data. Placing the data in one location does not remove the issue of data transfer speeds. Users with slow Internet connections might complain about sluggish performance. However, most of the bulk data transfers should take place at the server itself on high-speed lines. The majority of the communication with users can be reduced to simple pages consisting of input screens or simple data results.

The Web-based approach does not yet solve all distributed problems. For instance, a retail chain that has stores in multiple locations will probably want to keep most data locally. Each store would use a server to handle local transactions. This data can be transferred in bulk to the headquarters server a couple of times a day, where it can be made available for analysis over the Web. Even though the data eventually winds up on a central Web server, local server are still needed to improve transaction performance and reliability at each store.

Web Server Database Fundamentals

There is no standard mechanism for connecting databases to the Web server. Consequently, the method you use depends on the specific software (Web server and DBMS) that you install. Most of the methods follow a similar structure but vary in the details. Several tools exist to help you build forms in a graphical designer, supported by a programming language to process the data. These tools then gen-



```
<body>
<form id="form1" runat="server">
  <asp:Label ID="PageTitleLabel" runat="server" …
  <asp:SqlDataSource ID="CustomerSqlDataSource"
    DeleteCommand="DELETE FROM [Customer] …
    SelectCommand="SELECT [CustomerID], …
    UpdateCommand="UPDATE [Customer] SET…
    <DeleteParameters>
      <asp:Parameter Name="CustomerID" Type="Int32" />
    </DeleteParameters>
  <asp:FormView ID="CustomersFormView" runat="server"…
    DataSourceID="CustomerSqlDataSource">
    …
  </asp:FormView>
</form>
</body>
```

```
.PageTitle
{
    font-weight: bold;
    font-size: larger;
    text-align: center;
}
```

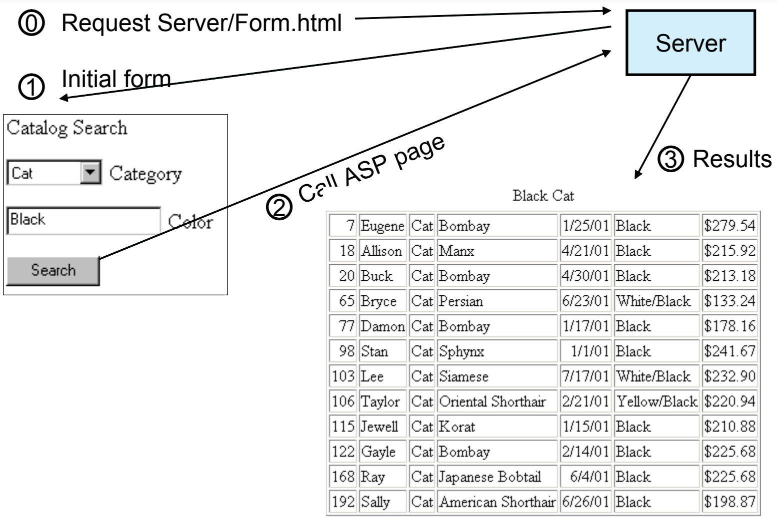
Figure 11.19

Web server database fundamentals. The server executes a script or code page that utilizes a data object to pass SQL commands to the DBMS and receive data and results. The page generates HTML and browser script that is sent to the browser along with a style sheet to establish the page data and layout.

erate the HTML files that are sent to the client browsers. One issue to watch for when selecting tools is that some of them require users to download special add-in software for the browsers. Most users are wary of downloading nonstandard components. When you are dealing with users outside of the main company, it is best to stick with tools that use only standard browser features. At the moment, the three leading tools are: (1) Microsoft ASP .NET, (2) Java, such as Oracle's JDeveloper and IBM's WebSphere, (3) UNIX-based scripting, particularly LAMP (Linux, Apache Web Server, MySql, and PHP or Python).

Figure 11.19 shows the basic process of connecting a DBMS to a Web server. As the developer, you create a code file on the server. The file contains data objects that pass SQL commands to the DBMS and receive results (and error codes). The page code can contain complex SQL statements and conditions. Ultimately, it generates the HTML code and browser script that is sent to the client's Web browser. Most systems also use a **cascading style sheet (CSS)** to establish the overall page design elements. The syntax of the code varies enormously, depending on the server. Pages written for one server system generally cannot be translated to run on a different system. Consequently, the choice of server and tools is a critical first step in building data-driven Web applications.

Systems that separate the code from the HTML and the styles have a substantial advantage. In particular, most organizations hire Web graphics designers and usability experts to design the final Web page that is seen by users. These people



Black Cat

7	Eugene	Cat	Bombay	1/25/01	Black	$279.54
18	Allison	Cat	Manx	4/21/01	Black	$215.92
20	Buck	Cat	Bombay	4/30/01	Black	$213.18
65	Bryce	Cat	Persian	6/23/01	White/Black	$133.24
77	Damon	Cat	Bombay	1/17/01	Black	$178.16
98	Stan	Cat	Sphynx	1/1/01	Black	$241.67
103	Lee	Cat	Siamese	7/17/01	White/Black	$232.90
106	Taylor	Cat	Oriental Shorthair	2/21/01	Yellow/Black	$220.94
115	Jewell	Cat	Korat	1/15/01	Black	$210.88
122	Gayle	Cat	Bombay	2/14/01	Black	$225.68
168	Ray	Cat	Japanese Bobtail	6/4/01	Black	$225.68
192	Sally	Cat	American Shorthair	6/26/01	Black	$198.87

Figure 11.20

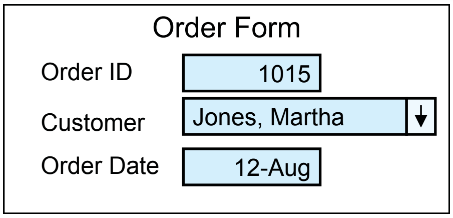
Client perspective. The client enters data into a form. Clicking the Search button sends the data to a server page. The server page retrieves the matching data from the DBMS and formats a new HTML page. This table is returned to the user, along with additional choices.

rarely know how to write database code. If the system mixes code with layout and design elements, it is difficult for the designers to alter the server files. By separating the three elements (code, layout, and style), the designers can fine tune their work without affecting any of the underlying procedural code. Similarly, the style sheet is easily modified by designers. If the style sheet is used for every page item, it is possible to change the overall look and feel of a Web site simply by editing this one file.

Browser and Server Perspectives

On the client browser the user will see a simple sequence like the forms shown in Figure 11.19. Once someone chooses a Search option, the AnimalSearch form is displayed on his or her browser. The user chooses a category and enters a color. When the Search button is clicked, the choices are sent to a new page on the server. This page retrieves the data and formats a new page. The data is generally stored in a table, similar to the one shown in Figure 11.20. The user never needs to know anything about the DBMS: Users see only forms and new pages. Each new page should provide additional choices and links to other pages.

Vendors are busy creating and refining tools to create Web-based forms that can interact easily with the database. The details vary enormously depending on which tool you use. However, the basic process remains similar. The server takes the values from the client form, validates them for rules you specify, and writes SQL queries to insert them or update existing rows in the database. You should review all of the issues discussed in Chapter 6 for building forms and reports. The main difference with the Web is that today's tools generally require more programming and individualized attention. On the other hand, Web-based applications create some potentially difficult problems that must be addressed. Data transmission,



Order Form

Order ID	1015
Customer	Jones, Martha
Order Date	12-Aug

Figure 11.21

Data transfer in forms. What if there are 10,000 customers? How long will it take to load the selection box? How long will it take to refresh a page with several selection boxes? How can a user possibly read and scroll all 10,000 entries?

concurrency, and server loads are significant issues that arise in Web-based applications. In fact, some of the most important differences in vendor tools can be found in how these problems are solved.

Data Transmission Issues in Applications

How much data can you send to a client form? At first glance, it seems straightforward to build a client/server or Web-based application. You simply move the database to a central server and use the network connections to handle the data transfer. However, these applications can present some challenging issues for data transfer and usability of forms. One of the most difficult issues is the use of drop down list boxes on a form.

Consider the main section of a standard order form shown in Figure 11.21. If you build this form in Access or Visual Studio and run it locally over a fast network, it will run fine. But what happens if there are 10,000 customers or the form runs at a remote location with a slow network? For simplicity, assume those names and identification numbers average 20 Unicode characters for each customer. So the selection box needs 10,000 times 20 times 2 or 400,000 bytes of data. At 8 bits per byte that is 3.2 million bits of data to transfer. Even at 3 mbps it would take a little over 3 seconds just to transfer the data for that one lookup box. If the network is slower, such as 300 kbps on a cell hone, it would take 30 seconds. Anytime you need to refresh the form or reload the drop down list box, it takes another 30 seconds. If your form has several selection boxes, the form takes even longer to load. Most users will be unhappy with the performance if forms take more than a couple seconds to load.

So why not just remove the drop down list box? To understand the issues, you need to remember why the selection box is useful. In a relational database, data is stored in separate tables that are joined through key data. In this case, the Order table contains the CustomerID. Theoretically, to place an order you simply need the customer's ID. (Eventually you will also need individual identification numbers for products as well.) You cannot expect your customers or clerks to memorize ID numbers, so the order form uses the drop down list box to look up customers alphabetically and return the matching ID number for the selected customer. If you remove the list box, you need to rethink the usability and find another method for customers and clerks to enter data.

Even without the data transfer issue, a selection box might not be the best solution when it has thousands of entries. Some boxes try to automatically find a

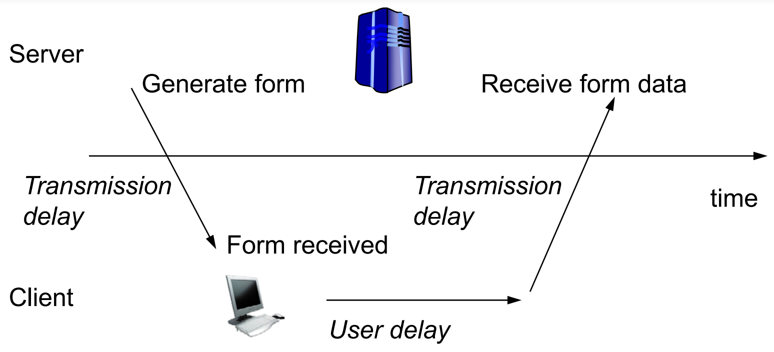


Figure 11.22

Latency. Transmission delays and user delays can create long latency for Web forms. Avoid pessimistic locking and carefully test for changes in the underlying database.

matching entry as a user enters the first few characters of a name or product, but this method still requires the user to know the first few characters. Hence, a potentially better solution is to create a more detailed search mechanism. Instead of a selection box containing all customers, the user could enter the first few characters of a customer's last name, click a button, and receive a small list of matching names.

ItemIDs present similar problems. Two common solutions exist: (1) For in-store sales, attach the product ID numbers to the individual items (e.g., bar codes); or (2) for Web sales, let the customer search for items and keep a collection of the selected ID numbers in a shopping cart.

For situations where you still want to use selection boxes, you need to be more creative with programming. For example, Oracle recommends that you do not use selection boxes for lists with more than 30 items. Instead, Oracle suggests the use of a standard text box, along with a **list of values (LOV)**. A list of values can be defined as a query. When the user enters the text box, the item can be selected from the list of values. How is this approach different from a selection box? The main difference lies beneath the surface. The LOV retrieves data in chunks instead of trying to transfer the entire set at one time. To the user, the list appears continuous, but by transferring only the currently displayed section of the list, the LOV reduces transmission time. It also transfers data while the user is reading, so the delay is less noticeable. This approach can be used, even if your tool or vendor does not support it directly. In Web forms, simply create a second form that holds the list of items on multiple pages with a search function. When users find the specific item, you can write a function to transfer the selected item to the main form. On the Web, this approach requires some slightly tricky Javascript (or ECMA script) coding.

For similar data transmission reasons, concurrency is also a problem with Web forms. In engineering terms, **latency** is a time delay in a system. In the context of forms, latency is the time between generating the form and receiving a response from the user. With a long latency, there is a greater opportunity for someone else to modify the same data elements, so concurrency is a greater problem. As shown in Figure 11.22, latency is typically longer on Web forms because of slow trans-

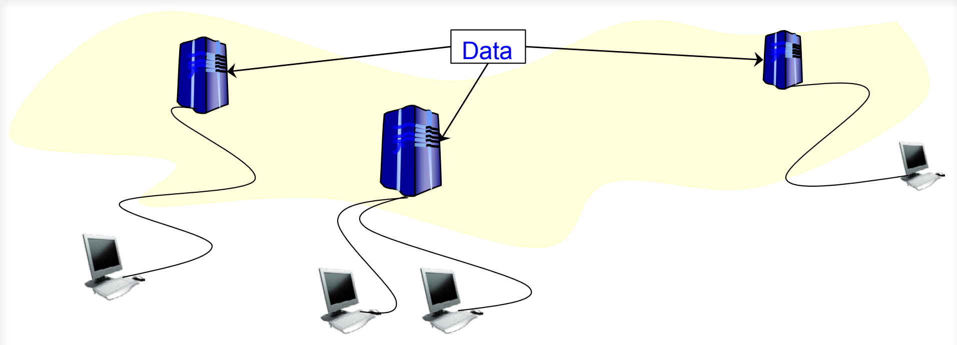


Figure 11.23

Cloud computing. Data is replicated to multiple, connected servers in the Internet cloud. Client requests are filled from the nearest available server, spreading the bandwidth and processor demands across the network.

mission lines and because users may be casual browsers who wander off and do other tasks before submitting the form. Consequently, Web-based applications should avoid pessimistic locking so that the data is available to more people at the same time. As a result, your application has to test and handle optimistic concurrency issues when the data has been changed by another process.

Cloud Databases

What benefits are provided by cloud computing and data storage? As the Internet has expanded, some of the structures have changed. In particular, a few large companies have become leaders in developing tools and in establishing multiple high-speed connections. These companies are led by Amazon, Microsoft, and possibly Google. All of them have huge data centers with very high-capacity Internet connections. Beyond their underlying businesses, they offer other firms the ability to use their computing and network facilities—for a fee of course.

Think about Web-based servers for a second. Typically, you think about one big server and its Internet connection. An expensive Internet connection might be capable of handling 155 megabits per second, which sounds like a lot of capacity. But, what happens if your server (and database) needs to deliver content to a million users—at the same time? The network becomes congested and you are averaging only 155 bits per second per user—way too slow for transferring large amounts of data. OK, perhaps a million simultaneous hits is high, but similar problems quickly arise with a smaller number of users when the Web server is delivering complex content such as images or video. As shown in Figure 11.23, some companies—notably Akamai—help reduce this problem by creating hundreds of data centers around the globe. Your data gets duplicated and distributed. Users interacting with your server might actually receive data from a nearby data center—reducing the overall processing and transmission load on any one server.

Cloud Computing Basics

Cloud computing arises by locating multiple servers and databases on the Internet. You can purchase time, space, and network data transfers by renting a virtual machine and virtual storage space on these distributed servers. At the simplest level, you can store files on a virtual Web folder (e.g., Amazon S3). You are charged a monthly storage fee along with monthly fees based on the amount of data transferred. This approach is typically used to store large files (books, music, videos, and so on) that need to be downloaded by people around the world. The service provider has high-speed data connections that can support a huge number of users, but you only pay for the actual usage rates.

In more complex scenarios, you can rent SQL Server data storage and query processing (e.g., Microsoft). The data is actually stored in Microsoft's cloud, and that company runs the network, hardware, and backup facilities. Your Web site issues queries and stores data, but the data is stored on these distributed servers.

It is even possible to rent virtual machine servers (e.g., Amazon EC2). These computers can be configured in a variety of sizes and with almost any operating system or software you can find. You pay a monthly fee based on the relative size/performance needed. Again, the service provider ensures that the computers keep running and provides network access. One of the key features of these virtual machines and cloud databases is that they can be expanded (or contracted) at almost any time. Also, initial fixed costs are low.

Think about the problem from the perspective of an Internet entrepreneur with a new idea. You could build your own data center, hire a staff, and buy hardware, software, and network access for hundreds of thousands of dollars. But you really do not know how much capacity you need. Alternatively, you can rent all of the initial capacity you need from a cloud provider for a monthly fee with little or no startup cost. If your company takes off and you get a huge increase in customers, you simply scale up the processing and storage with the cloud vendor.

However, you must continually re-evaluate your costs. The monthly rates charged by the cloud providers will ultimately be higher than those that you might achieve on your own. Once a company reaches a relatively steady-state, it might be cheaper to install your own computers or lease servers directly from a hosting company. But the decision depends on how much your demand fluctuates, and whether you want to hire people to configure and run your own servers.

Data Storage in the Cloud

Web-based data is interesting. It might or might not resemble traditional transaction data. For example, images, files, audio, and video are common on the Web. These items could be stored in a relational DBMS, but it is generally easier to store them as separate files—perhaps storing the file location in a relational table. Cloud vendors have initially concentrated on tools to handle these types of object data. For example, Amazon's S3 and Google's Bigtable are designed to store large chunks of object data. They are not relational databases and support only rudimentary query operations. However, they are distributed and have the ability to deliver data quickly—to anywhere in the world. For the most part, think of them as keyed storage—you provide a key value and the database returns the matching item stored at that location. (The systems, particularly Bigtable, are more flexible than that but it is a good starting point.)

When you need more traditional relational database services, you can also rent those—such as Microsoft's SQL Server offerings through its Azure platform.

In all cases, the data storage is most commonly used in combination with Web applications. But, Web applications are often used for in-house systems as well as public data, so this usage is not a limitation. Essentially, you treat the cloud server as a basic database server. The Web server (which could also be in the cloud), sends key values or queries to the cloud database, then formats and displays the returned data to place it into an HTML page.

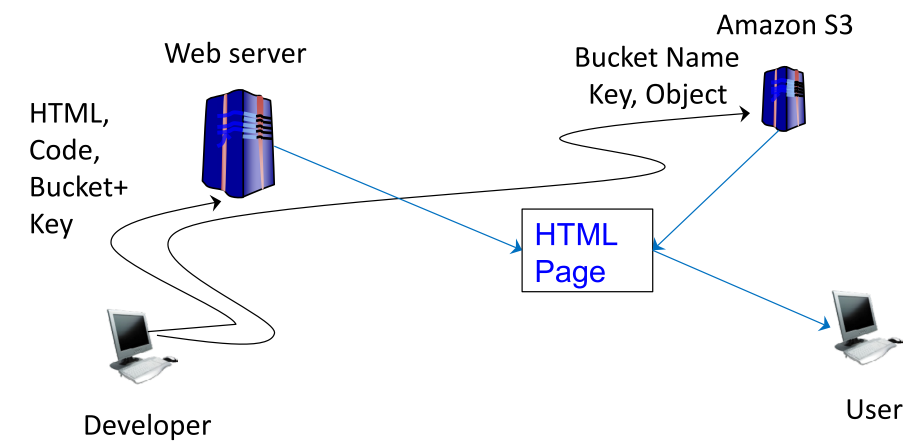


Figure 11.24

Amazon S3 process. Developer uploads object to an Amazon S3 account into a named bucket and gives the object a key value. The Web server code includes the HTML and the link to the S3 service with the bucket name and key. The Web.server delivers the HTML page and the nearest S3 server delivers the referenced content.

Figure 11.24 shows the basic process for the Amazon S3 service. Developers define buckets in Amazon, which are similar to file folders. Objects are uploaded to a bucket and assigned a key value. The developer creates pages on the Web server (which does not have to be at Amazon), and includes the bucket and key names to tell the browser to insert objects from the S3 service. The bucket and key names might actually be stored in a relational database and assigned based on some actions or choices by the user. For example, a page request might retrieve the bucket name and ID value to display a photo of an item being purchased. The Web server encodes the bucket name and key into a URL such as http://s3.amazonaws.com/mybucket/mykey. When the browser sees that link, it retrieves the specified object from the nearest available S3 server. Objects are generally uploaded to the S3 service using a simple transfer tool such as the add-in for the Fire Fox browser.

Microsoft's Azure SQL service is even easier. You interact with the service by writing SQL statements and upload data into tables. Then you write the Web page code using a database connection that specifies the Azure SQL service as the database provider. Your Web server runs the SQL statements and sends them to the Azure server which returns the values, and you insert the results into the Web page. The details are straightforward with most current Web programming platforms including Microsoft .NET and Java's JDBC.

Sally's Pet Store

How will Sally's employees access the database? Even with relatively simple applications, you need to think about how employees and managers will access the database. As a retail store, Sally's employees will need access to terminals or personal computers to record sales transactions and purchases or lookup customer or item data. Managers will have to print reports and browse for trends over time. You still need to determine where these computers will be located, how they will be connected, and where the data will be stored. Some of the answers depend on which DBMS you are using and the physical layout of the store. For example, will Sally's have a single checkout counter near the entrance with one terminal, or will there be multiple terminals scattered throughout the store? If you use Microsoft Access as the DBMS with multiple transaction computers, you will need to split the database to store the shared data tables on a file server. Or, you might transfer the tables to SQL Server and just use Access for the front-end forms and reports. With Oracle, a single server would hold all of the data, and the front-end tools would run on basic Web browsers. Since the machines are all contained within the store, you can install a relatively high-speed network and not worry about transfer speeds. If Sally wants access to the database from home, you will have to experiment with options.

If Sally wants to expand and add a second store, the decisions become more complex. She is also pushing for creation of a Web site, so that customers can order products, check on adopting animals, and get some help in caring for their pets.

Sally's request to expand the database to a second store raises many questions. Does she need "instant" access to the sales data from both stores all the time? Do the stores need to share data with each other? For example, if a product is out of stock at one store, does Sally want the system to automatically check the other store? Will the stores operate somewhat independently—so that sales and financial data are maintained separately for each store—or will the data always be merged into one entity? How up-to-date does data need to be? Is it acceptable to have inventory data from yesterday, or does it need to be up-to-the-minute?

The answers to these questions determine some crucial design aspects. In particular, the primary design question to answer is whether one central database should handle all sales or separate, distributed databases should handle each store. The answer depends on how the stores are managed, the type of data needed, the network capabilities and costs, and the capabilities of the DBMS.

In many ways, initially the cheapest solution is to keep the second store completely independent. Then there is no need to share data except for some basic financial information at the end of each accounting period. A second advantage of this approach is that it is easy to expand since each new store is independent. Similarly, if something goes wrong with the computer system at one store, it will not affect the other stores.

However, at some point Sally will probably want a tighter integration of the data. For example, the ability to check inventory at other local stores can be a useful feature to customers, which means that the application will need to retrieve data from several databases, located in different stores. These distributed databases must be networked through a telecommunications channel. There are many ways to physically link computers, and you should take a telecommunications course to understand the various options. Once the computers are physically linked, you need to deal with some additional issues in terms of creating and managing the distributed databases.

Summary

As organizations grow, distributed databases become useful. Distributed databases enable the company to expand individual departments without directly affecting everyone else. Distributed databases also give individual departments increased control and responsibility for their data. However, distributed databases, with independent database engines running in different locations, increase the complexity of developing and managing applications. One of the primary goals is to make the location of the data transparent to the user. To accomplish this goal, developers and DBAs need to carefully define the databases, networks, and applications.

Some of the major complications generated by distributed databases are query optimization; data replication questions; and support for transactions, concurrency controls, and deadlock resolution. These issues become even more complex when multiple databases are involved. Network transfers of data are substantially slower than transfers from local disk drives. Transfers over wide area networks can be slow and costly. These factors imply that developers must carefully design the applications and the data distribution strategy. The applications also have to be tested and monitored for performance and cost.

One of the major strategies in designing and controlling distributed databases is to replicate data. Instead of maintaining one source, it is often more efficient to replicate data that is heavily used in multiple locations. Of course, replication requires additional disk space, along with periodic updates and transfers of the data changes to each copy. Replication saves time by providing local access to data. It reduces costs by reducing the need for a full-time high-speed connection. Instead, bulk data is transferred at regular intervals—preferably at off-peak communication rates.

Client/server networks and client/server databases are a common means to design applications and distribute databases. Clients usually run applications on personal computers, and most of their power is devoted to the user interface. The data is maintained on a limited number of database servers, which are more efficient than simple file-server transfers. With a server database, the client sends an SQL query, and the server processes the request and returns only the desired data. With a file server, the client computer performs all the processing and must retrieve and examine all the data.

Larger, object-oriented applications are being built using a three-tier client/server architecture. The additional layer is in the middle and consists of business rules and program code (business objects) that execute on servers. The middle layer is also responsible for pulling data from the database servers and reformatting it for use by the client applications. Separating the three layers makes it easier to modify each component without interfering with the other elements.

The World Wide Web is becoming a popular mechanism to centralize applications and solve some of the distributed database issues. Web browsers have limited capabilities, but standards make it easier for everyone to get access to the applications and data. Keeping the data in one location simplifies security and concurrency issues. You still have to think about reducing the data transferred to client computers, such as avoiding huge drop down lists. You also have to deal with transferring data to external suppliers and customers. XML is a standard method of transferring bulk data. XML stores data in hierarchical files and XQuery provides searches of those files.

A Developer's View

Like Miranda, most developers understand the importance of the Web. The client standards make it easier to distribute data and connect with users around the world. Additionally, as applications expand, it becomes necessary to create distributed databases to improve performance and to support different regions. Distributed databases can significantly complicate application development. First be sure the application runs on one computer. Then get the best software you can afford. As much as possible, let the server databases perform the data manipulation and computation tasks. Use the client computers to display the results. Learn as much as you can about the Internet: It changes constantly, but will become increasingly important in your applications. For your class project, you should identify where the company might expand and where you would position distributed computers to support it. Explain how the database design would change in a distributed environment.

Key Terms

active data objects (ADO)
browser
cascading style sheet (CSS)
cloud computing
cluster
distributed database
fault tolerance
globally-unique identifier (GUID)
latency
list of values (LOV)
local area network (LAN)
replicate
replication manager
scalability
three-tier client/server
two-phase commit
wide area network (WAN)
World Wide Web

Review Questions

1. What are the strengths and weaknesses of distributed databases?
2. Why might a query on a distributed database take a long time to run?
3. When would you want to replicate data in a distributed database?
4. Why is concurrency a bigger problem with distributed databases than with stand-alone databases?
5. How does the two-phase commit process work?
6. Why is a client/server database more efficient than a database on a simple file server?
7. What are the advantages of the three-tier client/server approach?
8. How does a central Web site reduce problems with distributed databases?
9. How do you reduce transmission delays within data-driven Web sites?
10. What benefits are provided by databases run on the Internet as cloud computing?

Exercises

1. In each of the following situations, identify the best method of structuring the databases.
   - A. A single retail store with 7 checkout lanes, a manager's office, and an owner who wants to review reports from home.
   - B. Three retail stores with the same owner located in different cities about 50 miles apart. Each with 5 checkout lanes.
   - C. An engineering firm with computers used for design work where engineers spend considerable time at production sites.
   - D. A large marketing firm with major offices in Los Angeles and New York, where each office mostly works with local clients, but some work is shared.
   - E. A large agricultural firm with a custom application used to collect and analyze production data with headquarters in one state and farms located in at least three other states.
2. Assuming your DBMS cannot generate distributed safe keys automatically write a procedure to generate key values based on a location.
3. Compare the cost of using Amazon RDS versus Microsoft SQL Server cloud services. Assume the database content is about 10 GB with monthly data transfer rates around 1 GB per month.
4. You have the following distributed databases:

Location	Link Speed	Tables	Sizes

London	53 kbps	Contact(ContactID, Name, ClientID, Title, Phone)
		Employee(EmployeeID, Name, Phone, Title)
		WorkHours(WorkID, EmployeeID, ClientID, Date, Hours)
			5,000 rows
			2,300 rows
			3,000,000 rows

Paris	1.544 mbps	Contact(ContactID, Name, ClientID, Title, Phone)
		Employee(EmployeeID, Name, Phone, Title)
		WorkHours(WorkID, EmployeeID, ClientID, Date, Hours)
			10,000 rows
			1,000 rows
			20,000,000 rows

Frankfort	128 kbps	Contact(ContactID, Name, ClientID, Title, Phone)
		Employee(EmployeeID, Name, Phone, Title)
		WorkHours(WorkID, EmployeeID, ClientID, Date, Hours)
			7,000 rows
			3,500 rows
			30,000,000 rows

Madrid (HQ)	local	Client(ClientID, Name, Lead contact, Main city)
		Employee(EmployeeID, Name, Phone, Title)
		Project(ProjectID, ClientID, StartDate, Topic)
			20,000 rows
			10,000 rows
			1,000,000 rows

You are working for an accounting firm with headquarters in Madrid and major offices in London, Paris, and Frankfort. Many of the client companies have offices in three or four of these cities. Some clients are smaller and work with a single office. The accounting teams in the various offices need to share documents with teams in the other offices when they are working for the same client. Each office maintains a database of working papers, spreadsheets, questions, answers, and workflow data for the team. It also tracks billable hours for each employee and client. You need to get a list of all employees who have worked for a particular client in the last month; along with hours worked. Based on the communication speeds and table sizes, design the best performing query to answer this question. Could the database and query performance be improved by changing the distributed design?

5. A company has a database and an application where managers often generate and read a report that consists of 5 pages of dense text and numbers plus a chart on each page. If 100 people routinely view this report (with different data) each hour, and if the company wants to run the report on a central Web server, how much bandwidth capacity is needed? If the application is converted to tablets using cell-phone connections, how much will it cost in monthly cell phone bills?

6. You are working for a company that has two offices and is using a replicated database approach. Both offices have copies of the database of about 100 GB. Assuming that the entire database has to be transferred (both ways) between the two offices to handle synchronization, estimate the time required to handle the updates using at least three common network speeds.

7. A company wants managers to use mobile devices (phones, tablets, laptops) to access its database-driven applications. Briefly explain the different ways there are to develop the client (tablet/phone) applications. Which method would you select?

8. Find a development tool that can be used to create database-driven application on a Web server. Briefly describe the commands used to connect a server page to a database and explain what needs to be changed if the underlying DBMS is changed.

9. A company is building a Web-based application with SQL Server (or Oracle) as the backend DBMS. The middle tier uses a server-based programming language to generate HTML pages and insert data retrieved from the DBMS. The project manager wants on section of the application to implement pessimistic locking but does not think the application software can adequately handle it. Over half of the large application has already been written. What can you do to address the issue?

Sally's Pet Store

10. Sally is planning to add a second store. Write a plan that describes how the data will be shared. How will you control and monitor the new system? Which tools will you add?

11. Sally wants to connect to suppliers so that she can get information on orders and shipments electronically. The data needs to be imported into her database and matched to the orders. Describe a system that can handle these tasks.

12. Sally is thinking about implementing a Web site to sell pet products. Estimate the costs of storing the data for the site on Microsoft Azure SQL servers. The database would need to store the tables for Customer, Merchandise, Sale, and SaleItem. As an initial estimate, assume there will be 3,000 merchandise items, 10,000 customers with each customer placing an average of one purchase a month consisting of an average of three items on each order.

13. Use the tables for a different DBMS, or create them if necessary. Try to connect your primary DBMS to the new tables. Write a query that pulls data from both DBMSs.

14. Build a front-end application that handles the Sales form and connects to a database server.

15. Briefly explain the steps required (and estimate the time) to build an application so that store clerks could carry tablets on the floor to look up inventory or take special orders from customers.

Rolling Thunder Bicycles

16. Rolling Thunder is planning to expand to a second location across the country. How should the database be distributed? Where should each table be stored? Which tables should be replicated, and how should the data changes be reconciled?

17. Rolling Thunder is planning to expand by sending sales representatives around the country to various bike shops. They will use portable devices and a Web interface to configure bicycles and take new orders. The system should at least be able to run on an Apple iPad browser, and perhaps even a cell-phone browser. Describe how this system will work. What security provisions will be needed?

18. Describe a method to create a Web application that enables customers to check on the progress of their bicycle orders.

19. Create a second copy of your database running on a second computer. Create a link from the first database to the copy. Write a query that combines data from at least one table in each database.

20. Assume the owners want to convert the entire application to the cloud (such as Amazon RDS or Microsoft SQL Server/cloud). In the process, they want to increase marketing to increase sales and production to ten times the level in 2012. Estimate the new database size. Assuming customers place their own orders over the Web, estimate the cloud hosting costs.

21. Using the DBMS tools available to you, create a replica of the database; make changes to data in both copies and then synchronize the database to see the changes. Test what happens if you change the same data (such as customer phone number) in both copies before synchronizing.

22. Assume the managers want to temporarily connect a SQL Server of the database to an Oracle database with other tables. How can you build a link between SQL Server and Oracle that lets you run Oracle queries inside SQL Server?

Corner Med

23. The company owners basically want to franchise the operations. The headquarters will run database operations for all of the local clinics. Describe how you will configure the database to support this operational process. List any potential problems you might encounter.

24. Using one DBMS, research the capabilities for replicating data. Build at least one replica of a table, make changes to both copies, and synchronize the copies. Describe how the system handles generated keys.

25. If Corner Med decides to franchise and move the database to cloud computing, what security and privacy problems might arise and how could they be managed?

26. Assume the company has several offices, but physicians (and sometimes patients) move among the offices during the month. So the company builds the application on a centralized server with access through Web-based forms and reports. Would you add pessimistic locking to any of the tables or forms?

27. Related to distributed databases, assume that insurance companies send you payment data by downloading data files. The CSV files contain a line for each payment and list: Customer SSN, LastName, FirstName, VisitDate, PaymentAmount, and AmountDenied (not paid). Create an application to read this data and update the insurance company payment data. You might need to add more columns to existing tables, and you should create a sample file to test your application.

28. Create a second copy of your database running on a second computer. Create a link from the first database to the copy. Write a query that combines data from at least one table in each database.

Web Site References

http://www.w3.org/	Web standards body.
http://msdn.microsoft.com	Search for Microsoft's .NET framework and documentation.
http://www.oracle.com/technetwork/java/index.html	Java and JDBC documentation and references.

Additional Reading

R. Burns, D. Long, and R. Rees, Consistency and Locking For Distributing Updates to Web Servers using a File System, *ACM SIGMETRICS Performance Evaluation Review*, 28(2) September 2000, 15-21. [Performance issues in replicated databases.]

Date, C. J., *An Introduction to Database Systems*, 8th ed. Reading, MA: Addison-Wesley, 2003. [In-depth discussion of distributed databases.]

Fisher, M., J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference/3e*, Boston: Addison-Wesley, 2003. [Using JDBC and Java to connect to database.]

Chapter 12

Physical Database Design

Chapter Outline

What You Will Learn in This Chapter

- How does a DBMS store data for efficient retrieval?
- How does a DBMS interact with the file system?
- What are the common database operations?
- What options does a DBMS have for storing tables?
- How is one data row stored?
- How can you improve performance by specifying where data is stored?
- How does a DBA control file storage?
- What performance issues might arise at Sally's Pet Store?

A Developer's View

Ariel: How is the new job going, Miranda?

Miranda: Great! The other developers are really fun to work with.

Ariel: So you're not bored with the job yet?

Miranda: No. I don't think that will ever happen—everything keeps changing. Now they want me to set up a Web site for the sales application. They want a site where customers can check on their order status and maybe even enter new orders.

Ariel: That sounds hard. I know a little about HTML, but I don't have any idea of how you access a database over the Web.

Miranda: Well, there are some nice tools out there now. With SQL and a little programming, it should not be too hard.

Ariel: That sounds like a great opportunity. If you learn how to build Web sites that access databases, you can write your ticket to a job anywhere.

Getting Started

A DBMS sometimes provides options on how to physically store data. Most enable you to add indexes to improve query performance. Some systems enable you to select hashed or direct storage for data that needs immediate access. You can also use data clustering and partitioning to handle large data tables more efficiently.

Introduction

How does a DBMS store data for efficient retrieval? Any database application is created through the basic steps described in Chapters 2 through 9. You get the user requirements, design the database through normalization, create the queries using SQL, build forms and reports and then add the details to create a complete application. However, with large applications, one more step is critical to the success of your application. You must analyze its performance. Performance is largely controlled by telling the DBMS how to physically store and retrieve the data.

If computers were fast enough, how the DBMS physically stored the data for each table might not matter. Today, for small applications, this situation is probably true. The default storage method provides acceptable levels of performance, and you could skip this chapter. However, as databases and applications become larger or contain specialized types of data, physical storage becomes an important issue in the performance of your application. Large business applications routinely hold millions or even trillions of rows of data in tables. Proper configuration is essential—otherwise, even simple queries could take minutes or hours to run.

Two basic questions must be answered to store data tables: (1) How should each row of data be stored and accessed? and (2) How should individual columns be stored? The first question is more difficult to answer and is determined largely by how the data is used. Hence we must first examine the possible uses of the database. The answer to the second question depends largely on the type of data being stored. For traditional business data (numbers and small text), the answers are straightforward. If your application stores more complex data objects, the second question becomes more critical.

Two-Minute Chapter

Up to this point in the book, the features of relational databases have been discussed without the need to understand how data is actually stored and retrieved by the DBMS. In fact, that is one of the key features of a DBMS—it is free to optimize the storage of data without affecting the overall application design. However, sometimes it becomes useful to understand some of the underlying data storage techniques. When databases get huge and you need to find ways to improve performance, some DBMSs provide storage options that can make a big difference in usability.

Indexes are the most common method of improving performance for data retrieval. Most systems use linked lists and a B+-tree approach to storing indexed data. These topics are routinely covered in a second programming course in computer science disciplines (data structures). Some examples are given here but programming details are left for CS courses.

As pointed out in Chapter 9, adding too many indexes to a table can degrade performance when inserting or updating data in the table. So two other primary methods of storing data are sometimes available: simple sequential and hashed-key tables. It might seem strange, but sometimes sequential storage can be the fastest approach to handling big tables—as long as the data rarely changes and is generally retrieved as a large batch. Removing all other overhead items can substantially improve the raw transfer of the data. Hashed-key tables are trickier and not always available. They are useful when the data always has its own key value and you need rapid access to an individual item. For example, a bar-code number can be used as a key, or a transponder value from an RFID toll device (FasTrak in California or E-ZPass on the East Coast). The number provided is hashed (simplified) and directly converted into a physical location in the database file. So individual items can be retrieved or updated almost instantaneously.

Another approach that is used for huge databases is to partition the data or cluster items together. Remember the common Sales form that leads to separate tables for Sales and SaleItems. The related data from these tables (linked by SaleID) is almost always retrieved together. So some systems provide methods to store the related data together—making it faster to retrieve from typical disk drives.

Most applications work well with the standard B+-tree indexes, and as disk drive performance improves (such as using solid-state drives), this approach can be fast enough for most common business data. But for some specialized situations, performance can be dramatically improved with different storage approaches. Your job is to recognize when those tools are needed.

Physical Data Storage

How does a DBMS interact with the file system? Developers see database storage in terms of tables, but these tables ultimately need to be stored in files on the operating system. The main job of the database engine is to translate the concept of tables and rows into physical storage on the computer's disk drives. In computer science classes (particularly the data structures class), you will spend a lot of time coding different ways to store this data. This chapter simply introduces the basic concepts. Figure 12.1 shows how the operating system is responsible

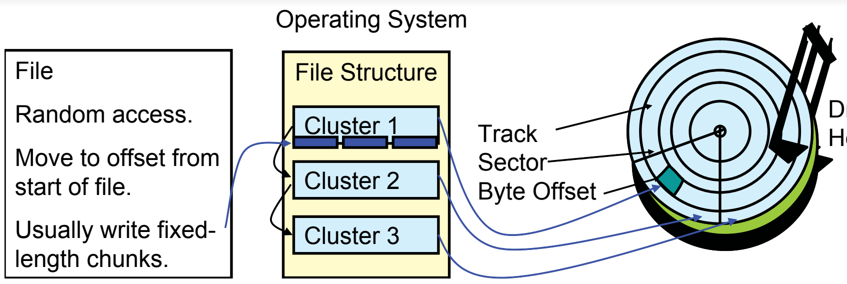


Figure 12.1

Physical data storage. The operating system breaks files into clusters and writes the clusters onto the physical disk drive. It uses internal pointers to retrieve the data sequentially, or randomly based on an offset from the start of the file. The DBMS uses file read/write commands to store chunks of data.

for translating files into physical storage on the disk drive. The file system (such as NTFS for Windows), breaks a file into clusters and uses internal pointers to record the physical location of each cluster. In most cases, the DBMS ignores the direct disk drive issues and lets the operating system handle the details. Instead, the DBMS uses the operating system's file read and write commands. For the main data storage, the DBMS creates a file and reserves a specified amount of disk space. The DBMS extends the allocated file space as the data grows. The DBMS can write a chunk of data anywhere within the allocated space by using the standard write command. The DBMS keeps track of which portions of the space are used by recording the **offset** (count of the number of bytes) from the start of the file. If you are familiar with programming, you should recognize the role of the standard fopen, fseek, fread, and fwrite commands available in stdio in C (and similar languages), or the fstream objects with open, seekg, read, and write methods in C++.

The challenge for programmers who create the DBMS is to translate the concepts of tables and rows into this file structure—so that the data can be stored efficiently and retrieved quickly. Several common storage methods have been developed over the past few years. One of them (B+tree) is commonly used for general data access and is useful in most situations. However, for huge databases, you might need more control over how the table rows are stored. Some DBMSs give you more choices and even if you do not intend to become a DBMS programmer, you need to understand their strengths and weaknesses.

Table Operations

What are the common database operations? To understand the differences between storage methods, you must first understand how the DBMS will use the data. Then by evaluating how each storage method affects the various table operations, you can choose the best method for your particular application. As shown in Figure 12.2, three major categories of operations affect tables: (1) retrieving data, (2) storing data, and (3) reorganizing the database. Each category contains more detailed tasks that are described in the following sections. Every application will perform all of the operations within the categories. As a developer you need to examine the application and identify the operations that are affecting performance.

```
Retrieve data
        Read entire table.
        Read next row.
        Read arbitrary row.
Store data
        Insert a row.
        Delete a row.
        Modify a row.
Reorganize database
        Remove deleted rows.
        Recover unused space.
```

Figure 12.2

Table operations. Every application must perform these operations. The key is to determine which operation is causing delays.

Retrieve Data

Retrieving data constitutes some of the most common activities in a database application. These operations also present the best opportunity to improve performance. Applications commonly perform three types of data retrieval. They read the entire table, read the next row in a sequence, and find and retrieve an arbitrary row.

Reading the entire table, or large portions of it, might not seem like a common operation, but it does occur relatively often when printing reports. For the example in Figure 12.3, to print weekly paychecks, the application will have to read every row in the employee table. But what if hourly workers are paid weekly, but managers are paid monthly? In most companies the managers represent only

Figure 12.3

Read a table sequentially. Sequential retrieval requires the data to be sorted; for example, this customer data is sorted alphabetically by LastName and FirstName. Fortunately, sort methods are so fast that they do not generally affect the application performance.

LastName	FirstName	Phone
Adams	Kimberly	(406) 987-9338
Adkins	Inga	(706) 977-4337
Albright	Searoba	(619) 281-2485
Anderson	Charlotte	(701) 384-5623
Baez	Bessie	(606) 661-2765
Baez	Lou Ann	(502) 029-3909
Bailey	Gayle	(360) 649-9754
Bell	Luther	(717) 244-3484
Carter	Phillip	(219) 263-2040
Cartwright	Glen	(502) 595-1052
Carver	Bernice	(804) 020-5842
Craig	Melinda	(502) 691-7565

a small percentage of the total workers, and retrieving 90 percent of a table is no different in performance than retrieving 100 percent.

Reading the next row in a sequence is related to retrieving all the data in a table. When an application needs to read an entire table, it is generally retrieved in some order or sequence. For example, paychecks might be printed in alphabetical order by employee name, department name, or postal code.

The more challenging retrieval operation is the ability to retrieve any arbitrary row. It is sometimes called random access because the database does not know which record might be requested. For example, any customer could place an order at random, and the database would have to retrieve the matching data for that customer.

This lookup process is one of the most critical elements to affect the performance of your application. It is easy to spot in situations like the customer example. The clerk enters a customer name or number, and the database has to retrieve the matching data. Clearly, you want to keep the lookup time as short as possible to avoid delays for the customer and the clerks.

Yet there is a more critical problem involving lookups. Any time you build a query, two types of random lookups come into play. First, joining two tables requires the database to match the values in one table with those in a second table. Second, any time you impose a condition with the WHERE statement, you are asking the DBMS to find rows that match that condition. So query performance is directly related to how fast the database can perform lookups and match the data requested. These lookups are critical because they are so numerous. Joining two tables could require thousands or millions of lookups—depending on the number of rows in the two tables. Remember that many tasks throughout the application use queries. Sequential lookups that retrieve large portions of the table require minimal optimization, because you have to read the entire table. The random retrievals and random lookups require more thought about optimization. However, storing data sequentially causes other problems when you need to insert, delete, or modify rows.

Store Data

A DBMS has to perform three basic operations involved with storing data: inserting a new row, deleting a row, or modifying the data in a row. Most systems implement a fast delete operation—they do not actually remove the deleted data. As shown in Figure 12.4, it is much faster to just mark the row as deleted. Then when the database wants to retrieve an item, the DBMS first checks to see whether the item has been deleted. If so, the DBMS ignores that row. Similarly, a good DBMS attempts to store data in fixed block lengths, so that if a row is modified, the DBMS can simply overwrite the data. With highly variable-length data, this operation is not always possible, so the DBMS must perform a delete and an insert operation.

In terms of performance, the biggest issue with delete operations involves storage space instead of speed. Although a row has been deleted, it still takes up physical space. Sometimes the DBMS can overwrite the old data, but after a while, there can be millions of bytes of unused fragments.

Inserting a new row of data is one of the more challenging aspects in a database management system. Next to random lookups, it is the source of the most performance problems. In fact, there is generally a trade-off between the two issues. If a system is good at random lookups, it is not as efficient at storing new data rows.

LastName	FirstName	Phone
Adams	Kimberly	(406) 987-9338
Adkins	Inga	(706) 977-4337
Albright	Searoba	(619) 281-2485
Anderson	Charlotte	(701) 384-5623
Baez	Bessie	(606) 661-2765
xBaez	Lou Ann	(502) 029-3909
Bailey	Gayle	(360) 649-9754
Bell	Luther	(717) 244-3484
Carter	Phillip	(219) 263-2040
Cartwright	Glen	(502) 595-1052
Carver	Bernice	(804) 020-5842
Craig	Melinda	(502) 691-7565

Figure 12.4

Delete a row. Deletion is fast because the DBMS just marks the row as deleted. It does not actually remove the data.

That is, the techniques used to improve random lookups often require significantly more time to add data rows.

The performance issues of adding new data are somewhat technical and will be explained in more detail in the section on data storage methods. For now, examine your application to identify which tables will add new data on a regular basis and which tables might add data only occasionally. For example, a firm might add only a few new items a year to the Products table. However, thousands of new rows could be added to the Order table every day.

Reorganize the Database

Largely because of the deletion method, a database can become disorganized over time. Data that is flagged as deleted is still hiding in the table space. Empty holes of storage space are too small to hold new data and data rows that are used together are no longer stored near each other.

These problems are particularly challenging with relational databases. In a relational database the system data is also stored in tables. For example, the form layout that you redesigned 20 times is stored as rows in a table. Each time you redesigned it, the database flagged the old version as deleted and saved the new version. Complex forms could take up several thousand bytes of storage.

Most systems have an administrative command to reorganize or **pack** the database. This command causes the DBMS to go through the data and rewrite each table—clearing up the storage space. A major challenge to database administration is to determine how often to run this command. Two complications exist. First, it can take several hours for this command to process large databases. Second, a few systems require that all users be logged off the DBMS before the administrator can run this command. You want to avoid database systems with the second requirement. It prevents you from providing 24-hour access to the database. However, even if other people can still use the system, database reorganization can affect the overall performance of the application, so the process generally needs to be performed during slow periods (e.g., at night).

On the flip side, if you forget to periodically reorganize the database, it can rapidly fill with wasted space. It is not uncommon for even a small Access database to grow from under 1 megabyte to 5 or 6 megabytes of storage space during development. Be sure to use the database utilities to compact the database. Doing so will make it much easier and faster to back up and copy the data files.

Identifying Problems

During the database design stage, you should be able to identify potential problems. You need to analyze the database usage and volume statistics collected in Chapter 3. In particular, look for large tables; heavily used tables; transaction tables requiring fast database responses; and queries with multiple joins, complex criteria, or detailed subqueries. You should also perform tests during the development of the applications. Generate large sample tables and test the performance of the queries, forms, and reports. Once the database application is operational, you can use the performance monitoring tools described in Chapter 11 to locate bottlenecks.

Once you identify the form, report, or query that is causing delays; you need to determine the cause of the problem: data retrieval, data storage, or data reorganization. You can use the programming debug feature to step through code that utilizes many different operations. By timing procedures and loops, you can determine which section is causing the longest delays. You can also use the Timer function to record the times of various operations.

Once you have identified the location of the delays, you can test various strategies for improving performance. If the delays involve your program, explore different ways to reorganize your code to improve performance. If delays are due to data retrieval or storage, think about ways to perform data operations in larger blocks. For example, your program might run faster if it writes individual changes to a temporary table and then uses SQL statements to transfer the changes to the primary tables in one large operation.

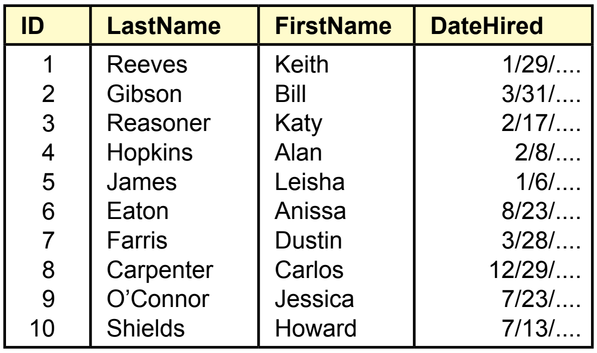
A second method to improve performance is to alter the way the data is stored. Each DBMS provides different controls over data storage. The following sections summarize the most common techniques.

Data Storage Methods

What options does a DBMS have for storing tables? Three primary methods are used to store data tables—each with several variations. The simplest method is sequential storage—putting the data into tables in the order in which it is most commonly accessed. To provide faster access, particularly for random lookups, a second approach is to create indexes of the data. A third approach known as direct or hashed-key storage is radically different and is designed to optimize random lookup at all costs.

Sequential storage is relatively easy to understand, but probably the least useful. Hashed storage methods are also straightforward, but have their own limitations. Indexed tables are by far the most common means of storing and accessing data today. They are complex and have many variations. To choose the best storage method, you sometimes have to understand the differences between the variations.

Pointers and linked lists are key topics in understanding how indexes work. You might have heard computer science students discussing these topics. Do not panic. You do not need to know how to program routines using pointers and linked lists.



ID	LastName	FirstName	DateHired
1	Reeves	Keith	1/29/....
2	Gibson	Bill	3/31/....
3	Reasoner	Katy	2/17/....
4	Hopkins	Alan	2/8/....
5	James	Leisha	1/6/....
6	Eaton	Anissa	8/23/....
7	Farris	Dustin	3/28/....
8	Carpenter	Carlos	12/29/....
9	O'Connor	Jessica	7/23/....
10	Shields	Howard	7/13/....

Figure 12.5

Sequential file. Each row is stored in some predefined order. Sequential storage is used primarily for backup or for transferring data to a different database.

To understand their strengths and weaknesses, you just need to be able to draw some basic diagrams.

Sequential Storage

Sequential files are the simplest method of storing data. Each row is stored in a predefined order as shown in Figure 12.5. As long as the data is retrieved in the order specified, access is fast and storage space is used efficiently. The real problems arise when data is added or when users need to retrieve data in several different sequences.

Uses

Sequential storage is useful when data is always retrieved in a fixed order. It is also useful when the file contains a lot of common data. For example, if most customers have the same ZIP code, you might as well leave the ZIP code data in simple sequential storage.

Another use of sequential files is for backup or transporting data to a different system. Each database system stores data in a proprietary internal format. To transfer data from one system to another generally requires exporting the data to a common format, moving the data, and importing it into the new database. A sequential ASCII file is a popular export/import format that most database systems support.

Drawbacks

To understand the drawbacks to sequential storage, consider the steps involved in performing the basic database operations listed in Figure 12.2. Reading the entire table and retrieving the next sequential row are easy. Finding an arbitrary row is much slower. If the rows can hold different lengths of data, the only way to find an item is to search from the start of the table until the desired row is found. With N rows of data, the expected number of retrievals required to find a random row is (N + 1)/2, or a table with 1,000,000 rows would require 500,000 lookups on average to find a matching row. Obviously a bad idea.

Another major drawback can be seen by examining the data storage operations. As with every method, flagged deletion is fast and relatively efficient. The real

ID	LastName	FirstName	DateHired
8	Carpenter	Carlos	12/29/....
6	Eaton	Anissa	8/23/....
7	Farris	Dustin	3/28/....
2	Gibson	Bill	3/31/....
11	**Inez**	**Maria**	**1/15/....**
4	Hopkins	Alan	2/8/....
5	James	Leisha	1/6/....
9	O'Connor	Jessica	7/23/....
3	Reasoner	Katy	2/17/....
1	Reeves	Keith	1/29/....
10	Shields	Howard	7/13/....

Figure 12.6

Insert into a sequential table. Copy the top of the table to a new table. Store the new data row (Inez). Copy the rest of the data. The system must read every row in the table.

problems arise when you want to insert a new row. Examine Figure 12.5 and decide how you would insert data for a new employee with the last name of Inez. The basic steps are shown in Figure 12.6. If you had to write a program to insert a row, the most efficient method is to follow four steps: (1) Read each row. (2) Decide if this row comes before the new row. If so, store it in a new table. (3) When you reach the insertion point, save the new row of data. (4) Append the rest of the data to the end of the new table. The main drawback to this approach is that any time you want to add a row of data, the database has to retrieve (and probably rewrite) every row in the table.

Pointers and Indexes

The most common solution to the problems of sequential tables is to store each row separately and use pointers to find a row. This approach also uses indexes to establish the sequential retrieval of data and to improve searches. Separating rows of data means that each row is stored as an independent group. (Actually, you can break rows into smaller chunks, but for now, think of each row stored independently.) When a row of data is stored, it is stored at some location. This location is called an **address**, and a variable that holds this address is called a **pointer**. With most file systems, the address (and pointer value) is a number that represents the offset in bytes from the start of the file.

Figure 12.7 illustrates how the data is separated. It also shows how an index is used to retrieve the data. The data is linked to the index via the address pointers. To retrieve the data sequentially, the DBMS simply loops through the index and follows the pointers to retrieve the data. The data rows can be stored in any order in the file structure.

An **index** is the most common method used to provide faster access to data. An index sorts and stores the key values from the original table along with a pointer to the rest of the data in each row. Figure 12.8 illustrates the concept. Notice that a table can have many indexes. Indexes can also be based on several columns of data. The ability to create multiple indexes in a table indicates their first strength.

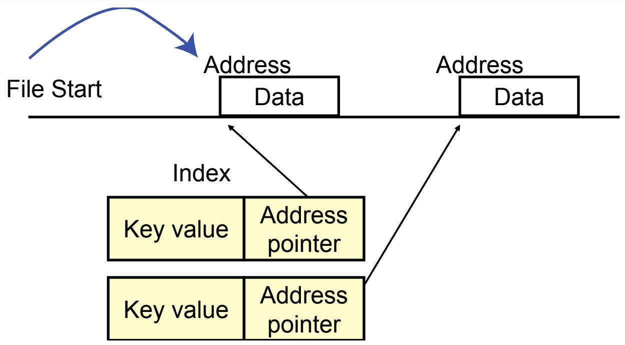
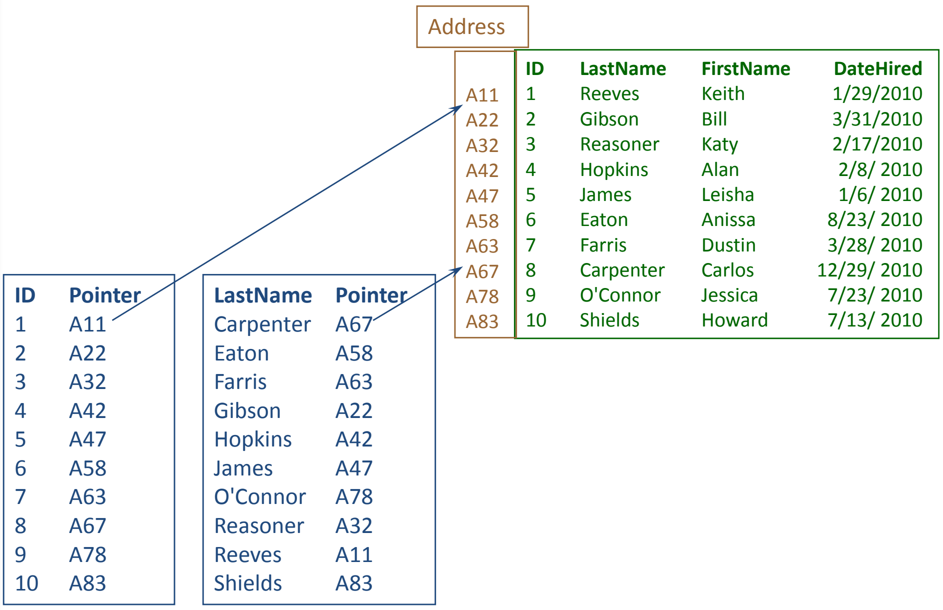


Figure 12.7

Use of pointers. The database searches the key values. When it finds the appropriate key, it follows the pointer to retrieve the associated data stored on the disk.

Figure 12.8

Indexes. An index sorts and stores a key value along with a pointer to the rest of the data. Indexes can be built for any column or combination of columns in the table. The two separate indexes provide different sorts and searches for one table.



Address	ID	LastName	FirstName	DateHired
A11	1	Reeves	Keith	1/29/2010
A22	2	Gibson	Bill	3/31/2010
A32	3	Reasoner	Katy	2/17/2010
A42	4	Hopkins	Alan	2/8/ 2010
A47	5	James	Leisha	1/6/ 2010
A58	6	Eaton	Anissa	8/23/ 2010
A63	7	Farris	Dustin	3/28/ 2010
A67	8	Carpenter	Carlos	12/29/ 2010
A78	9	O'Connor	Jessica	7/23/ 2010
A83	10	Shields	Howard	7/13/ 2010

ID	Pointer
1	A11
2	A22
3	A32
4	A42
5	A47
6	A58
7	A63
8	A67
9	A78
10	A83

LastName	Pointer
Carpenter	A67
Eaton	A58
Farris	A63
Gibson	A22
Hopkins	A42
James	A47
O'Connor	A78
Reasoner	A32
Reeves	A11
Shields	A83

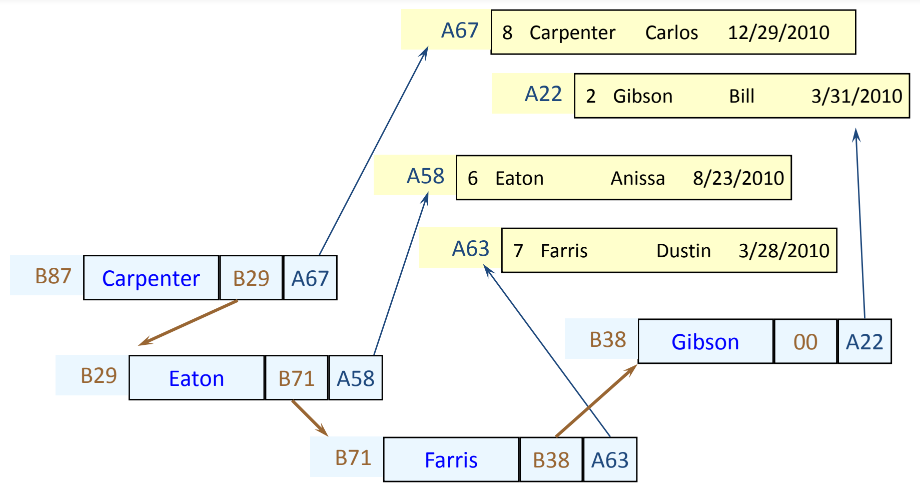


Figure 12.9

Linked list. The index is split into separate index elements. Each element contains a key value (LastName), a pointer to the next index element, and a pointer to the rest of the data for that row. To retrieve data sequentially, start at the first element (Carpenter) and follow the link (pointer) to the next element (Eaton).

They enable relatively fast, sorted access to a table based on any criteria. Indexes generally provide a clear advantage over straight sequential files because they support high-speed access to any data columns.

The astute reader will recognize that the index has not really solved all of the problems—it has simply transferred them to the index file. That is, to store and retrieve data, you face the same problems in building the index. On the plus side, the index is smaller and easier to manipulate. It is also possible to create multiple indexes for any table, so it can be searched or retrieved using different key columns. But, it would be nice to find a better way to handle the index itself.

Linked Lists

To solve the insert problem, indexes are generally based on linked lists instead of sequential lists. A linked list is a technique that splits data even further than a sequential index. With a linked list, any index element can be stored separately. A pointer is then used to link to the next index item. Figure 12.9 illustrates the basic concepts. In this example each row of data is stored separately. Then an index is created that is keyed on LastName. However, each element of the index is stored separately. An index element consists of three parts: the key value, a pointer to the associated data element, and a pointer to the next index element.

To retrieve data sequentially, start at the first element for Carpenter. Follow the pointer to the next element (B29 points to Eaton). Each element of the index is found by following the link (pointer) to the next element. The data pointer in each index element provides the link to the entire data row for that key value (A67 points to the Carpenter row).

The strength of a linked list lies in its ability to easily and rapidly insert and delete data. Remember the difficulty in inserting data with a sequential table. Even with a sequential index, inserting a new row generally results in copying half the index (or more). For large tables this approach is clearly inefficient.

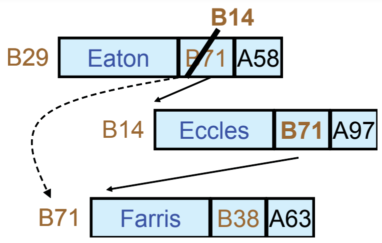


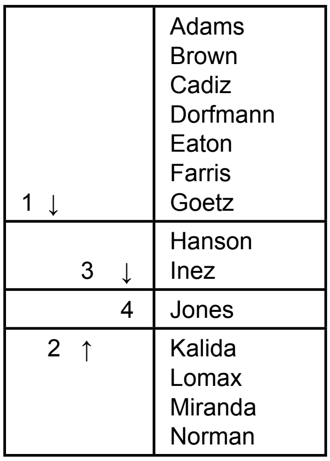
Figure 12.10

Insert into a linked list. To add the index element for Eccles: store the new data element, keep the address (B14); find the sort location—between Eaton and Farris; move the link pointer from Eaton into Eccles (B71); store the pointer for Eccles (B14) in Eaton.

On the other hand, as shown in Figure 12.10, inserting a new key row into a linked list requires three basic steps. (1) Store the data and store the index element—keeping the address of each. (2) Find the point in the index to insert the new row using a binary search. In the example, Eccles comes between Eaton and Farris. (3) Change the link pointers. The link in Eaton should point to Eccles (change B71 to B14) and the link in Eccles should point to Farris (insert the B71). Those are the only steps needed. No copying of data keys and no complicated code.

Figure 12.11

Binary search. A sorted index can be searched rapidly using a binary search. To find the entry for Jones, find the middle of the list (Goetz). Jones is past Goetz, so split the second half in half (Kalida). Keep splitting the remainder in half until you find the entry.



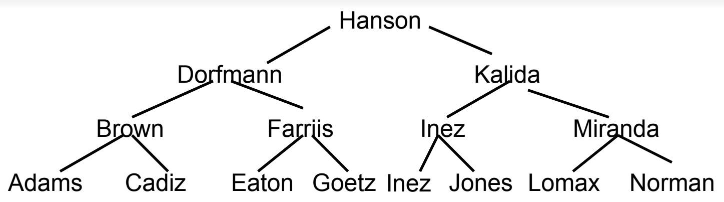


Figure 12.12

Simple tree. Each node element has a key value, a pointer to data for that key, and two link pointers. One pointer is for values less than the key. One is for values greater than or equal to the key.

Linked lists have substantial advantages for most of the standard table operations. In particular, they are the most efficient way to insert and change data because the code simply edits the link pointers to add or delete something from the list. But, how can linked lists improve searching for and retrieving random items in the list?

B+Trees

As noted in Chapter 9, sorted lists like indexes provide a relatively efficient method to search for data. A binary search can take advantage of the sorted data by cutting the search in half at each step. Figure 12.11 shows the search process. Recall that a binary search can find any specific entry with no more than log2(N) retrievals. In this example with 14 entries, log2(14) is 3.8, or a maximum of 4 lookups. The example specifically uses Jones because it requires all 4 retrievals.

It is clear that binary searches are efficient, but how does that help with linked lists on indexes. First, recognize that indexes are sorted, so it should be possible to use a similar approach. Second, think about the list for a few minutes, and you can see that it can be reorganized. Instead of trying to store it sequentially, grab the middle entry (the starting point for any search), and build a tree structure. In many ways, a tree is just a more complex way of storing a linked list. Instead of linear, it contains multiple links.

One version of a tree is shown in Figure 12.12. Only the key values are shown in this figure. In practice, each **node** or element on the tree would contain an index element much like those in Figure 12.10. That is, each element would contain the key value, a pointer to the rest of the data, and two link pointers. For the particular tree in Figure 12.12, each element has at most two links. One link (the line to the left) points to elements that have lower values. The other link (line to the right) points to elements that have a value greater than or equal to the value in the node. The **root** is the highest node on the tree. The bottom nodes are called **leaves** because they are at the end of the tree branches.

The power of the tree lies in its ability to find a data element. To find the data for Jones, start at the top of the tree (Hanson). Jones is alphabetically greater than Hanson, so go to the right side. Track down the tree depending on the key value until you reach the bottom element for Jones. Notice that every element requires at most four searches because there are only four levels in the tree. Notice that the search was exactly the same as the binary search. The number of searches is given by the **depth** of the tree, which is the number of nodes between the root and the leaves. Notice that if you compress a B+tree down to one level, each element

- Set the degree (m)
  - m >= 3
  - Usually an odd number.
- Every node (except the root) must have between m/2 and m children.
- All leaves are at the same level/depth.
- All key values are displayed on the bottom leaves.
- A nonleaf node with n children will contain n-1 key values.
- Leaves are connected by pointers (sequential access).

Figure 12.13

B+tree rules. These rules will generate a tree structure that provides good database performance under a variety of conditions.

would be in one long key row. In other words, you would end up with indexed sequential access.

The power of a B^+tree for searching is clear, but what if you want to retrieve the data sequentially? The answer is that the leaves or bottom nodes contain a link to the next item. When the DBMS reads to the leftmost leaf (Adams), it can follow points to the right to retrieve each final item in sequence.

B^+Tree Definition

On examining Figure 12.12, it quickly becomes clear that there are many ways to organize a tree. For example, why is Brown listed beneath Cadiz instead of beneath Adams? There is no good answer to this first question. Minor positional choices like this one are arbitrary and do not affect the tree. But the question shows that there is some flexibility in the final tree. Bigger questions do affect the tree significantly, such as why is the tree approximately symmetrical—that is, why not let one side reach lower than the other side? Why does each node split into two branches—why not three or more?

Answers to each of these questions will affect the layout of the tree. As the layout changes, so does the performance. Computer scientists have studied these structures in detail. For database purposes, they have determined that the best overall performance is provided by a B^+tree, which follows the six basic rules shown in Figure 12.13. The rules are not as complicated as they may first appear.

First, you have to choose the degree of the tree. The **degree** represents the maximum number of children that can fall below any node. Choosing the degree determines how fast the database can find any particular item. In Figure 12.12 the degree was 2, which produced a binary search. Higher degrees result in trees that are broader, requiring even fewer searches to find any item. Two rules that give the B^+tree its power are that each node must have at least m/2 children (and no more than m children) and that all leaves must be at the same depth. In other words, the tree cannot be lopsided, but must be balanced so that data is distributed relatively evenly across the tree.

Figure 12.14 shows a small B^+tree of degree 3. With a degree of 3, a node can point to three different children. If it does, the node must have two key values, such as (458, 792). To understand why, search the tree to find key value 692. Start at the top and note that 692 is greater than 315, so go to the right branch. Now 692 falls between 458 and 792, so branch to the middle child and then drop down to find the entry on the bottom leaf, which contains a pointer to the rest of the data. A node with three children must have two keys. Any value lower than the left-most

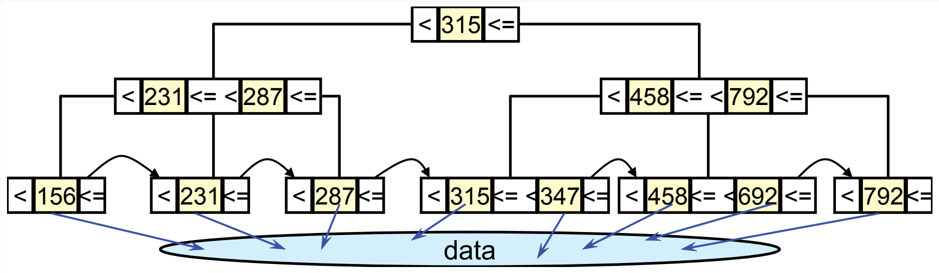


Figure 12.14

Sample B+tree of degree 3. Start at the top to find the value 692. It is larger than 315, so go to the right branch. It is between 458 and 792 so go down the middle. The bottom leaf points to the rest of the data. The bottom leaves also contain links to provide sequential access.

key goes to the left. A value greater than the right-most key goes to the right. Anything between the keys follows the middle path.

Uses

The main strength of the B$^+$tree is that it provides a guaranteed level of performance for access to the data. Every element can be found in the same number of searches—which is determined by the depth of the tree. The tree also provides fast sequential retrieval. The other power comes from the ability to add or delete elements from the tree. As in a linked list, adding new items to a tree is relatively easy. The process is a little more complicated in a tree, because the rules require the tree to be rearranged periodically as data is added. You can study the details of programming a B$^+$tree in a computer science class. The basic operations are straightforward, but somewhat tedious. You can also buy software to create and manipulate B$^+$trees. However, adding items to a tree is still relatively fast and efficient.

Overall, the B$^+$tree approach provides the best general access to data. If you do not know anything useful about the data or how it will be used, you should always choose the B$^+$tree method to store a table. It provides the best overall performance for typical data—for sequential retrieval, random lookup, and for changes to the data.

Drawbacks

The drawbacks to B$^+$tree storage are relatively minor. It has been shown to be the best general purpose storage method, and most DBMSs use it as the main storage method. One criticism has been that the coding is relatively complex, but standard algorithms have been developed for several years, so it is not really an issue. The bigger problem is that for large tables that involve constant changes, it takes time to reorganize the index for every change. The problem is worse when you create multiple B$^+$tree indexes on a table. Inserting a row could trigger changes in several indexes and result in restructuring millions of items in each index. Many systems recommend that if you are going to bulk insert thousands of rows of data, you should turn off all indexing, insert the data, then index the table one time. The only other solution is to use indexes sparingly on tables that have heavy transaction changes.

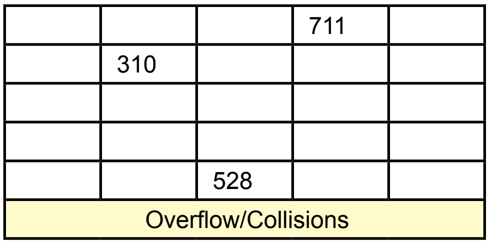


Figure 12.15

Hashed-key access. The key value (528) is converted directly into a storage location by dividing by a prime number (101). The remainder (23) is used to identify the storage position. If two keys have the same remainder, one is stored in an overflow location.

Direct or Hashed Access

Some situations require super fast random access to data. For example, in transaction situations you might need virtually instantaneous retrieval of some data items. When a grocery store clerk scans an item, the DBMS must retrieve the price immediately. A delay of even 5 seconds would be incredibly annoying and costly given the huge number of items that are scanned every day. In this example, the computer is given a unique bar-code number and needs to retrieve the matching data. It makes sense to optimize the search for this situation.

A **direct access** or **hashed-key** storage method solves this problem better than any other approach. The method works by first setting aside enough space to store all the key values you might need in numbered storage locations. Then the key value (bar code) is converted to a storage location number. Computer researchers have determined that a prime modulus function usually provides the best conversion. For example, you might have 100 elements with key values ranging from 100 to 9911. You choose a prime number approximately equal to the number of elements. For this case 101 is a good prime number. Then you divide each key value by the prime number and look at the remainder. As shown in Figure 12.15, a key value of 528 has a remainder (or modulus) of 23. Hence data for that key will be stored in location number 23. There is one catch—some keys might have the same modulus. The system sets aside an overflow area for these collisions, which it searches sequentially.

Uses

The hashed-key approach is extremely fast for finding and storing random data. The key's value is immediately converted into a storage location, and data can be retrieved in one pass to the disk. This method works best for transaction operations that require instantaneous retrieval of small amounts of data.

The hashed-key storage method requires you to know approximately how many items will be stored in the table. It also works best if the data does not change very often. It is acceptable to set aside enough space to add a few items. The method begins to deteriorate if key values are constantly being added to the table.

Drawbacks

One drawback to the hashed-key storage method is that it has little or no provision for sequential retrieval of data. It is possible to retrieve the data and sort it. Some

Operation	Sequential	B+Tree	Hashed
Read one	••	••••	•••••
Read next	•••••	••••	•••
Read all	•••••	••••	•••
Insert	•	••••	••••
Delete	•	••••	••••
Modify	•	••••	••••
Reorganize	••	••••	•••

Figure 12.16

Comparison of access methods. The B+tree is the best overall method to store and retrieve data. Sequential is useful for large tables that do not change often and need only sequential access. Hashed is useful for rapid access to individual items.

order-preserving hash functions exist to keep the keys in a predefined order. However, sequential retrieval will be slower than with a B$^+$tree index.

A second drawback is that the method sets aside storage space for the data, so you have to know how much space will be needed before you collect the data. If you add items to the table, they tend to end up in overflow storage, which is substantially slower. Performance can be improved by reorganizing the table—which creates more space and uses a new prime number. However, it takes time to reorganize the table, which should be done when the data is not being heavily used.

Bitmap Index

Some vendors (e.g., Oracle) provide highly compressed bitmap indexes for large tables. With a **bitmap index** each data key is encoded down to a small set of bits. The bitmap (binary) image of the entire index is usually small enough to fit in RAM. High-speed bit operations are used to make comparisons and search for key values. Hence the bitmap indexes are extremely fast. Bitmap indexes are particularly useful for columns like secondary keys that contain large amounts of repeating data. They should not be used for a column that contains all unique values. For example, in a typical SaleItem(<u>SaleID</u>, <u>ItemID</u>, Quantity) table, you could consider using a bitmap index for the SaleID and ItemID columns. But you would not want to use a bitmap index for the SaleID column in the Sale table. In Oracle, you use the CREATE BITMAP INDEX to generate a new bitmap index.

Comparison of Access Methods

All of these access methods are critical to computer scientists who create the DBMS. As an application developer, you do not need to know the gory technical details of the various methods. However, you do need to understand the strengths, weaknesses, and best uses of the methods. A good DBMS will let you choose how you want to store each table. At a minimum the DBMS will provide the ability to specify indexes for various columns. To determine which method should be used to store and retrieve data, you need to know two things: the primary operations that will be performed on the table and which method best supports those operations. Figure 12.16 answers the second question by summarizing the comments from the previous sections.

In practice, you have only three choices. First, the B$^+$tree is the best overall method to store and retrieve data. In almost any table the primary-key columns should be stored in a B$^+$tree index to speed the join operations in queries. Second, hashed access should be used for tables that do not change often and the application requires fast retrieval or storage of data based on a key value. Third, sequential storage can be used if a table almost never changes and the application always retrieves data sequentially and in large chunks. Generally, your choice comes down to B$^+$tree or hashed access. If you have tables that change often, you should consider removing indexes—which creates a sequential table. Most modern databases use some version of B$^+$tree storage. Primary keys are almost always indexed this way by default.

Storing Data Columns

How is one data row stored? The previous section explored the various methods of storing and retrieving individual rows of data. The second issue in storing data is how to store individual columns of data within a single row. For basic business data consisting of numbers and short text, it rarely matters how individual columns are stored. However, applications are being developed that need to store more complex data such as large amounts of text, graphics, sound, and even video clips. This data is relatively complex and requires significantly more storage space. Despite the declining cost of storage space, some of these objects are so large that you must be careful in how the database allocates storage for each item.

Text and Numbers

Fixed-width or positional storage is the simplest means of storing a row of data as shown in Figure 12.17. Each column is allocated a fixed number of bytes, and the data is stored in a set position. When the DBMS retrieves a row, it can find each column because the table definition lists the starting position of each column. The biggest drawback to this method is that at the start you must decide on the width of each column. Any data that does not fit into the assigned width will be truncated. This decision causes problems. For example, how much space should you set aside for a customer name? If you pick a small number, you risk throwing away part of a customer's name. If you pick a large number, the database sets aside that much space for every row of data—wasting space for most situations. This type of storage is used when you specify the domain as numeric or a CHAR column with a fixed width.

Figure 12.17

Fixed-width or positional-column storage. If data widths do not vary much, this method is a fast, efficient means to store columns. If descriptions can be short or very long, then you will have to allocate space for the longest possible description, which wastes space for the short descriptions.

ID	Price	QOH	Description
4	110.00		Dog Kennel-Extra Large
18	1.00	1874	Cat Food-Can-Premium
29	6.00	240	Flea Collar-Cat

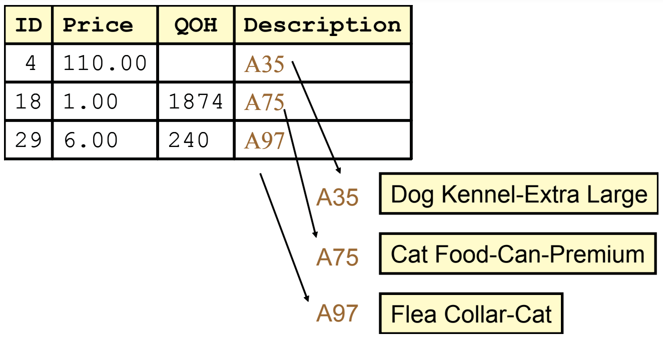


Figure 12.18

Variable length columns. Text columns that can be variable should be stored as variable width (varchar). The DBMS stores a pointer to the data that is stored in a pool.

The problem of deciding how much text space to allocate is common. Hence, a solution was developed to accommodate text data that is highly variable in length. For example, descriptions, comments, and memos can be long or short. In these situations the best storage method to use is the **variable length** method shown in Figure 12.18. In this case only a pointer is held in the actual row of the table. The data is stored in a separate pool. In SQL databases you specify this type of storage by selecting the **VARCHAR** column type. Some databases also provide a memo or comment data type to implement this type of storage. For example, Access provides a Memo type, which can hold large chunks of text. The Memo type can hold up to 64,000 characters, whereas text columns are limited to 255. For most systems you should always use the VARCHAR instead of fixed-width CHAR to store a text column. The exception is that small text columns, such as a two-letter state code, will be slightly more efficient if you use fixed width.
Note that numeric data is almost never stored as characters. Instead, it is stored in binary format to save space. The numbers used in these figures are just for illustration. You rarely have to worry about the width of numeric columns; they typically use either 4 or 8 bytes of storage.

One of the more challenging problems is storing variable-length string data, particularly when the lengths can vary widely, such as comments. If the system allocates a fixed amount of space, every row would be at the maximum value, and most of the space would be wasted. On the other hand, if the system allocates space for each row dynamically, then some rows will be shorter than others. This approach saves space, but makes it more difficult to handle modifications of the data. If the new data is longer than the old row, the system cannot just overwrite the old row. Some systems (e.g., Oracle) solve this dilemma by allocating data blocks to hold a group of rows. Each block contains a certain amount of free space. The DBMS uses this free space to store modified data that is longer than the existing row. In Oracle, you can control the amount of free space through two parameters: PCTFREE and PCTUSED. If the current data block is fuller than the PCTFREE value, no new rows are added to the block. The remaining space is kept for expansion of existing rows. See the *Oracle Server Administrator's Guide* for details and suggestions on values for these parameters.

Image and Binary Data

Most DBMSs provide the ability to store binary data within the database itself. For example, you can create a column to hold a picture for each row. Unfortunately, no standards exist for defining these columns or using this data. Hence, if you use these features, it is difficult to convert your database to another vendor's format. The data type varies depending on the DBMS: Access uses an OLE Object column, Oracle uses LONG RAW or BLOB (binary large object), SQL Server uses image. More importantly, the internal data format, and the storage and retrieval methods are different for each vendor. It is relatively easy to create and store data in these formats. The only problem is if you need to transfer data from one DBMS to another. Usually, the only answer is to retrieve each object one at a time, return it to its native format, and then store it in the new DBMS. It is a relatively painful process that you want to avoid.

From a performance standpoint, you will have to experiment with each application to decide if it is worthwhile to use these binary data types. The advantage of storing binary data within the DBMS is that you gain the use of the concurrency protection and database backup facilities. The main drawback is the difficulty in accessing the binary data using other software. Most software applications (e.g., drawing packages) do not know how to store and retrieve data from your DBMS, so you need to create an intermediate program to handle the exchange.

The alternative to storing binary files within the DBMS is to store them in a separate subdirectory, and then store only the file name within a text column in the database. This method is commonly used for Web-based applications. The Pet Store example uses this method to provide support with different databases.

Transferring Data with Delimited Files

If you need to transfer data to a different database or a different application, you often have to use a delimited file. It is often called a delimited file because the table is converted to standard text characters (no binary numbers). As shown in Figure 12.19, each column is separated by a specific character or delimiter. A comma is a common delimiter. Because text data might contain commas or other special characters, text columns are enclosed in quotation marks. Spaces are eliminated unless they are in quoted text columns. Missing data is simply not displayed, so if a column is missing, the data row would have two adjacent commas (e.g., 110,,"Dog …"). This technique is not very useful for permanent use within a database. Every time it retrieves a row, the DBMS has to search for the commas and interpret the quotes to find a particular column. However, it is a good way to transfer data between different systems. It is also good at saving space—particularly when many columns are missing.

Figure 12.19

Delimited files. Each column is separated by a special delimiter character (,). Text columns are quoted to protect spaces and hide special characters like commas. This method is often used to transfer files to different databases or other applications.

```
4, 110, , "Dog Kennel-Extra Large"
18, 1, 1874, "Cat Food-Can-Premium"
29, 6, 240, "Flea Collar-Cat"
```

Data Clustering and Partitioning

How can you improve performance by specifying where data is stored? Another way to improve database performance is to control the location of individual components of the table. For example, some parts of your application may always be retrieved together, so performance might improve if the two sets of data are retrieved together. On the other hand, sometimes you collect data that might not be accessed very often. It is still worthwhile to keep the data, but it might be better to store it on cheaper, slower drives. A third technique exists to speed up access to data by spreading it across several disk drives. All three situations are related in that they involve partitioning data and controlling where it is stored to improve performance. The key to understanding these methods is to remember that mechanical disk drives are slow. Every access to the disk that can be avoided will improve the application's speed.

Data Clustering

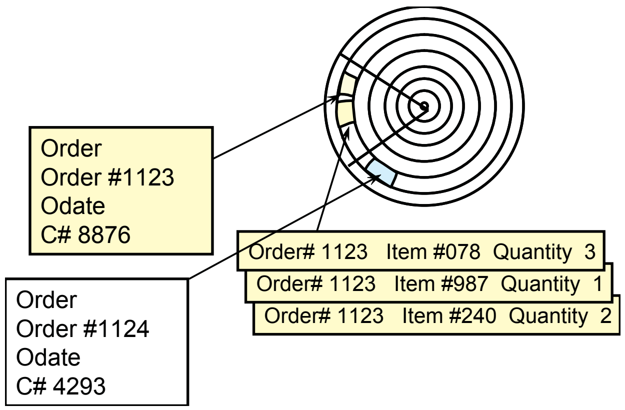
To improve general system performance, most computers retrieve data in chunks. They try to anticipate the next demand and read ahead of the current request. If the system guesses correctly, the next data request can be filled from RAM, which is substantially faster than waiting for the drive to spin around again.

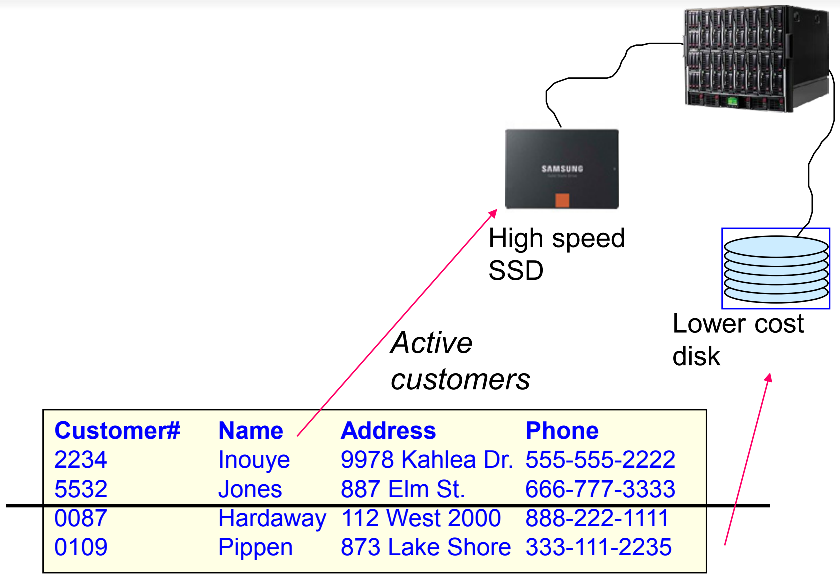
Database systems designers have used this concept to improve performance of database applications. Some parts of an application are generally used at the same time. Consider the example presented in Figure 12.20. Generally, when users look at order items, they also want to see the related data stored in the order table. By storing all the data for Order 1123 in the same data block, the data can be retrieved in one pass. The application will run faster because it avoids a second trip to the disk drive.

If you are using a DBMS that supports data clustering, you can improve performance by identifying data that is commonly accessed together. To create a cluster, you need to specify the tables involved and the key columns that link those tables.

Figure 12.20

Data clustering. Order and OrderItem data are usually needed at the same time. By storing them close to each other, the computer can retrieve them in one pass. Clustering the data improves application speed by reducing the number of disk accesses.


Order
Order #1123
Odate
C# 8876
Order
Order #1124
Odate
C# 4293
Order# 1123 Item #078 Quantity 3
Order# 1123 Item #987 Quantity 1
Order# 1123 Item #240 Quantity 2



Customer#	Name	Address	Phone
2234	Inouye	9978 Kahlea Dr.	555-555-2222
5532	Jones	887 Elm St.	666-777-3333
0087	Hardaway	112 West 2000	888-222-1111
0109	Pippen	873 Lake Shore	333-111-2235

Figure 12.21

Horizontal partition. Data for currently active customers is stored on high-speed drives. Older data is moved to cheaper, slower drives. The user does not need to know about the split because the DBMS automatically retrieves the data.

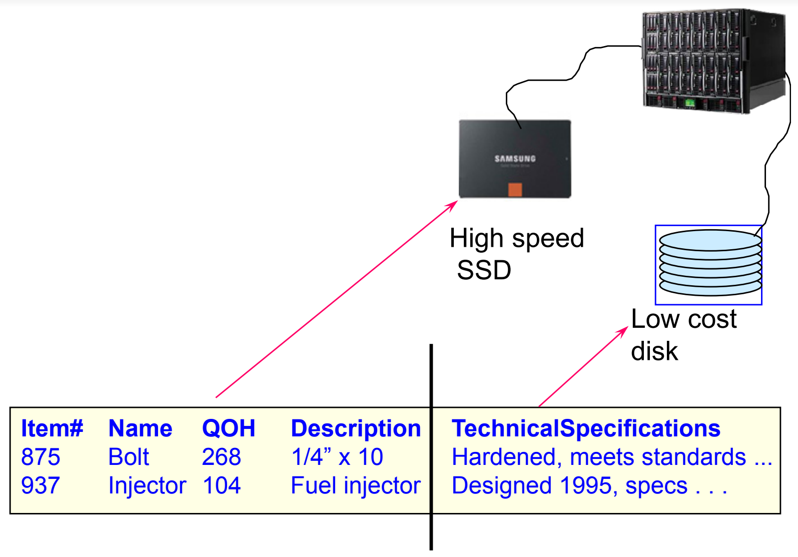
The DBMS then automatically stores and retrieves the related data in the same cluster. Only some of the large transaction-oriented database systems support clustering. For example, Oracle has a CREATE CLUSTER command to define the tables and key columns.

Data Partitioning

Another situation that commonly arises in business applications is that some data is used more frequently than other data. Even in the same table, you might collect data that is used only occasionally. For example, a basic customer table could contain information on customers who have not placed orders for several years that the marketing department wants to keep. Because the data is rarely used, it would be nice to move it to a cheaper storage location.

As shown in Figure 12.21, this situation would involve a **horizontal partition**. Some of the rows (currently active customers) will be stored in one location, and other rows (inactive customers) will be stored in a different location. The active data will be stored on high-speed disk drives. In extreme situations, some of this data could be stored on solid-state RAM drives, which hold all data in semiconductor RAM. On the other hand, the less-used data can be placed on slower-speed optical drives. The optical drives can hold huge amounts of data at a low cost; however, their access speeds are somewhat slower.

The key to making this approach work is that after you set it up, a good DBMS automatically retrieves the data from the appropriate drive. The user does not have to know that the data is stored on different drives. A single SQL query will retrieve the data—wherever it is stored. The high-end DBMSs provide several methods for determining the partition. Common methods include range partitioning (e.g.,



Item#	Name	QOH	Description	TechnicalSpecifications
875	Bolt	268	1/4” x 10	Hardened, meets standards ...
937	Injector	104	Fuel injector	Designed 1995, specs . . .

Figure 12.22

Vertical partition. Technical data and images that are not accessed very often can be stored on a high-capacity, low-cost, but slower hard drive or even an optical drive.

specify a range of ID values) and list partitioning (e.g., list the key values that fall into each partition).

Vertical partitioning uses the same logic. The only difference is that with vertical partitioning, some columns of data are stored on a faster drive, whereas others are moved to cheaper and slower drives. Figure 12.22 shows how a product table might be split into two pieces. Basic business data used in transactions is stored on a high-speed disk. Detailed technical specifications and images are stored on high-capacity optical disks. Most day-to-day operations will use the basic data stored on the high-speed drive. However, the detailed data is readily available to anyone who needs it. The only difference is users will wait a little longer to retrieve the data on the slower drive.

In theory, data can be partitioned using any DBMS. Simply define two tables that can be joined by a common key. Then store each table on the appropriate drive. The difficulty with this approach is that anyone who wants to use the data will have to know that it is stored in different tables. You can circumvent this issue by building a query that automatically combines the tables. Then users can pull data from the query without having to know where each piece is stored.

In practice, horizontal partitioning is often used to split data so that it can be stored in locations where it will be used the most. For instance, you might split a customer table so that each regional office has the set of customers that it deals with the most.

On the other hand, vertical partitioning is useful for limiting the amount of data that you need to read into memory. If some columns are rarely used, they can be stored in a separate table. Overall performance will improve because the DBMS will be able to retrieve more of the smaller rows.

Managing Tablespaces

How does a DBA control file storage? Each vendor provides different methods to monitor and control database performance. These tools are a major selling point for each vendor. Smaller systems like Microsoft Access provide only limited control over the physical storage of data. System developers generally use the storage methods that are appropriate for the most general situations (B^+tree). You control column storage by the data type you assign.

Larger systems like Oracle provide a variety of tools to help evaluate and manage the performance of the database. For example, Oracle sets clustering and provides hashed access with the CREATE CLUSTER command. Indexed files can also be partitioned and clustered. Oracle database performance can also be tuned with various parameters. For example, the PCTFREE and PCTUSED options specify how tightly the data should be packed into the defined space. Various STORAGE parameters specify how the database should be expanded as it grows. Tables and indexes are stored in tablespaces, which are areas that the database administrator allocates on a drive. By specifying the location of the tablespaces, you can allocate data on specific drives. You can improve performance by storing each element in a tablespace on a different drive. For example, large databases should store transaction and recovery logs and main data on different drives.

Sally's Pet Store

What performance issues might arise at Sally's Pet Store? At the start the Pet Store database should have few performance problems. Beginning in one store, an ambitious system might store the database on a central computer, which is connected to three or four other computers in the store. Reasonably up-to-date personal computers should be able to handle the initial database. As accounting functions are added, or if the system needs to expand beyond a single store, then the system would have to be reevaluated.

At the current time, there should be few concerns about performance tuning. However, to improve performance, all primary keys should be indexed. Microsoft Access generally defines these indexes by default, but you should examine each table to be sure. Be careful when assigning indexes to columns that are part of a concatenated key. The index on a partial key must allow duplicates.

One potential area for problems is the City table. This table currently holds basic data on cities throughout the United States. Performance could be improved by reducing the number of cities—on the assumption that most customers would come from the surrounding communities. However, if you choose to keep the data, you can improve performance by thinking about how the table will be accessed. In particular, it is often searched by ZIP code. Similarly, because users often want a sorted list of the cities, it would be useful to index the City column. Are there too many indexes for one table? You could test the performance of retrievals before and after adding the indexes. However, note that the City table is predominantly used for retrieval and rarely used to add data. Hence building additional indexes makes sense.

The same situation probably exists for the Merchandise table. Most applications and users will retrieve data from the Merchandise table, with few updates, deletions, or insertions. Hence you might build additional indexes on that table.

For now, partitioning and clustering are not warranted. Over time, as the business expands, you might want to move some of the older data to less expensive storage devices. For instance, data on animals sold more than 5 or 10 years ago will probably not be used often and could be placed on slower CD-ROM drives. Similarly, inactive customer data, and older order data can be moved from the primary tables. The exact dates will depend on the cost of storage, discussions with Sally, observation of retrieval patterns, and legal needs.

Summary

Large application databases sometimes need to be fine-tuned to improve their performance. Some systems provide control over how the data is stored and retrieved. Three basic types of controls can be used to determine (1) how table rows are stored and retrieved, (2) how individual columns are stored, and (3) how data is clustered or partitioned.

The primary choices for storing rows of data are B$^+$tree indexes, hashed-key access, and sequential files. The method depends on how the data is used in terms of the standard database operations. The most challenging operations are searching for random entries and adding new data to the table. The B$^+$tree approach is the most common because it provides the best overall access for a variety of situations. In particular, it provides reasonably fast random access, good sequential retrieval, and good performance for inserting and deleting rows of data. In contrast, the hashed-key approach provides high-speed random access to any data element, but it is poor at retrieving data sequentially. Sequential files are rarely used, because although they use a minimum of space, they provide weak access to random rows of data.

Most DBMSs provide some control over how individual columns can be stored. The most common feature enables developers to control the storage of text data. Large text columns should be stored in varying-character columns instead of fixed-width columns. You should also be familiar with using delimited files for transferring data to different systems.

Some systems can cluster data in common locations on the disk drive. This approach improves performance by enabling the disk drive to retrieve related data in one pass. Another useful technique is to partition data so that data that is used less often can be moved to less expensive, slower disk drives. RAID systems provide another performance gain by splitting data and storing it on independent disk drives within the same system. The RAID drives can store and retrieve data substantially faster than a single disk drive can. RAID drives can also provide automatic backup by storing each component on two different drives.

Be careful when attempting to improve the performance of an application. Changes that help one area can adversely affect other operations. This trade-off is important when creating indexes for columns in a table. Indexes tend to improve data retrieval but slow down the processing when data is added to the table.

A Developer's View

As Miranda's problems indicate, database performance can become an important issue. Performance problems should be anticipated and solved as early as possible in design and development. You do not have to be intimately familiar with how the DBMS stores data. However, you do need to know which options are available to you. With many systems, the most important control you have is in choosing which columns to index. Sometimes you can choose the exact storage method. You need to understand the strengths and weaknesses of the various methods so that you can choose the method that best fits your application's needs. For your class project, you should identify the columns that should be indexed. You might have to generate sample data and compare processing time for various operations.

Key Terms

address
bitmap index
degree
depth
direct access
fixed-width
hashed-key
horizontal partition
index
leaves
node
offset
pack
pointer
root
VARCHAR
variable length
vertical partition

Review Questions

1. What basic data operations are performed on tables?
2. What are the primary data storage methods for tables?
3. What are the strengths and weaknesses of sequential storage?
4. How do linked lists solve insert and delete problems?
5. What are the strengths and weaknesses of indexed (B^+tree) storage?
6. What are the strengths and weaknesses of hashed (direct access) data storage?
7. How does data clustering improve database performance?
8. How does data partitioning improve database performance?
9. How is storage different for CHAR versus VARCHAR data types?

Exercises

1. Using the documentation for one DBMS, write the commands to create a table using a hashed-key index on an integer primary key column.
2. Based on the sample data in Figure 12.10, write the logic for the code to insert a new element in a linked list.
3. Research the documentation, DBAs, magazine, or Internet sources and find two methods or tricks that can be used to improve performance of your DBMS. Identify the specific problem the hint is designed to solve.
4. Create a B^+tree (degree 3). Show each final tree.
   a) The base tree holds the following key values: 1038, 1164, 2314, 3678, 4164, 5931, 6104, 7368, 7547, 8442, 8556, 8777, and 9114.
   b) Add the key value 8655.
   c) Add the key value 2715.
   d) Add the key value 10911.
   e) Add the key value 2941.
   f) Delete the key value 9114.
5. Draw a linked list.
   a) Start with the following key values: 341, 492, 561, 678, 781, and 856.
   b) Show how to insert the key value 603.
   c) Show how to delete the key 781.
6. Create a hashed storage example. Use a prime number of 53. Show the storage of the following numbers: 781, 467, 198, 435, 351, 782, and 149.
7. Write the commands to partition a Customer table based on the CustomerID. Older data has lower values for CustomerID, so split the table into three partitions based on values of 10,000 and 20,000.

Sally's Pet Store

8. The basic version of the database is relatively small and there should not be any current performance problems. However, if the company expands into several cities with multiple stores, performance could become more important. Outline a plan for how you could expand the database to handle this situation. Identify the DBMS software you would choose.
9. Go through the list of tables and classify them into two groups: (1) Transaction tables that receive many updates, and (2) Lookup or analytical tables that are used in transactions but are seldom updated, so they can include more indexes.
10. Copy the City table and remove all of the indexes from the copy. Create a query that counts the number of customers from each state using the original City table. Create a second copy of the query that uses the copy of the City table. Run both queries and comment on the performance of the two queries.

Rolling Thunder Bicycles

11. Make a copy of the Rolling Thunder database. Write SQL statements to perform the following operations on the Bicycle table: (a) add a row, (b) delete a row, (c) select all rows, and (d) write a program to change one value in every row. Write four short programs to perform these operations in a loop that repeats at least 100 times. Run the programs and record the time it takes to perform the operations. Next, index every column in the Bicycle table and rerun your tests. Record and analyze your results.
12. Examine the tables and the usage of each table in the Rolling Thunder application. Identify the primary uses of each table in terms of the table operations described in this chapter. Use this list to identify desired indexes and appropriate storage methods for each table if the database becomes large.
13. Examine the tables in Rolling Thunder and identify which tables should be clustered. Which tables could gain from partitioning? If the application is expanded, what new data could be added that might gain from partitioning?

Corner Med

14. Examine the tables and the usage of each table in the Corner Med database. Assume the database is going to become relatively large when it is used at multiple locations. Identify the tables that are primarily transaction based versus the lookup and analysis tables. Use these lists to specify additional indexes that might be added (or removed) to improve performance. What other options could be used to improve performance?
15. Assuming the company has operated for five years, how would you partition the data to reduce storage needs and improve performance?
16. If the company decides to digitize other medical records (x-rays, photos, lab results, prescriptions, and so on), what performance problems can be expected? How will you minimize these issues?

Web Site References

http://www.sql-server-performance.com	Hints on improving performance for SQL Server.
http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm	Oracle performance tuning.
http://www.bluerwhite.org/btree/	General B-tree information and coding.

Additional Reading

Baeza-Yates, R. and Ribeiro-Neto, B. *Modern Information Retrieval*, Reading, MA: Addison-Wesley, 1991. [Computer science approach to storing and retrieving data, includes Web access and multimedia.]

Goetz, G., Modern B-Tree Techniques, Boston: Now Publishers, 2011. [Basic textbook on B-tree processing.]

Korfhage, R., *Information Storage and Retrieval*, New York: Wiley & Sons, 1997. [Summary of data storage methods.]

Loomis, M. *Data Management and File Processing*, Upper Saddle River, NJ: Prentice-Hall, 1983. [In-depth treatment of data storage issues such as B-trees.]

Dunham, J. *Database Performance Tuning Handbook*, Berkeley: McGraw-Hill, 1997. [In-depth treatment of improving your application's performance.]

Chapter 13

Non-Relational Databases

Chapter Outline

What You Will Learn in This Chapter

- Why would anyone need a non-relational database?
- What are the main features of non-relational databases?
- How are databases designed and queried using Cassandra?
- How does cloud computing benefit key-value pair databases?

A Developer's View

Ariel: Why the puzzled look, Miranda?

Miranda: Well, my company built this great Web site that lets customers post comments, rate products, and interact with each other…

Ariel: Yes, that sounds fairly standard today.

Miranda: But, there are millions of customers! We ran some tests with sample data and it runs really slowly. Plus, it looks like we would need a huge server, and the DBMS license fees will be enormous—just to provide a free service to customers.

Ariel: Wow! That does sound like a problem. Are there any other options.

Miranda: Yes, that is the confusing part. Big companies like Facebook have developed non-relational DBMSs that emphasize scalability and speed. Some are even open-source so we don't have to pay license fees for each copy. All of which is great, but these things are really new and they keep changing, so it is difficult to figure out how to structure the data and write the code.

Ariel: Hmm. That does sound tricky. But it sounds like it is useful to learn the basics to help you decide when to use the tools.

Getting Started

Web applications tend to handle data in a relatively unique pattern: Most data is exchanged as key-value pairs. Starting with data on forms passed to a Web server, the data is coded with a key (such as the textbox name) and the corresponding value. Large Web sites also struggle with handling data for millions of users. So new systems have been defined to store and retrieve these individual pieces of data as quickly as possible. The Cassandra project is one of the most popular. It focuses on the ability to store data across multiple servers; both for performance and to minimize disruptions if one node fails. The data design for these systems is not normalized, and Joins are not supported. This chapter presents the basic elements of design and data retrieval in non-normal databases.

Introduction

Why would anyone need a non-relational database? The importance of the Web has three major impacts on technology systems: (1) The need to run 24-7 (24-hours a day, 7-days a week); (2) The ability to handle relatively complex object data (pictures, long text, and so on); and (3) Handling data for millions of users. In terms of Web 2.0 (Web services and social interaction), the other key aspect is that companies provide these services with minimal or no fees on customers. The large scale of the applications causes several problems with performance, reliability, and cost. This last element has led large Web firms to develop open source tools to handle many computing aspects in an attempt to hold down licensing costs. Along the way, they decided to use a different data model to improve performance and reliability. These newer systems do not use relational data storage. The early versions had minimal support for data storage and retrieval, so some people referred to them as **NoSQL** databases. But it is better to refer to them as **non-relational**, because the major differences lie in the data model and storage, not in the query language. Of course, as Chapter 1 points out, non-relational systems have been around longer than relational, so how are these new systems different from the old tools? The answer to that question revolves around two key features: distributed databases and key-value data storage.

The new DBMSs are evolving rapidly. Most of the tools were originally developed by specific companies for their particular needs (such as BigQuery (BigTable) and MapReduce by Google, Hadoop (HBase) by Yahoo, and DynamoDB by Amazon). The open-source community then developed variations on the tools. Some similarities exist across the tools, but in the end they are all different. As of 2013, some Web sites list 150 variations of NoSQL tools. To illustrate the concepts, this chapter explains the general concepts but focuses on one DBMS: Cassandra. Cassandra is in the top 3 list for non-relational DBMSs in terms of popularity or usage. It is also one of the best performing and has ongoing development with installation files for several operating systems (including Amazon EC2). It also has an interactive query language which makes it possible to explore the data without detailed programming. However, any real-world application would require programming, which is also supported through several languages.

Chapter 11 introduces how distributed databases can improve reliability and scalability. Data storage in the non-relational systems was built from the ground up to run as distributed systems. In particular, Cassandra operates as **peer-to-peer** distributed nodes. The system is designed to be installed on multiple servers, in multiple clusters, and across multiple data centers—which could be located anywhere in the world. Any piece of data is replicated across multiple servers, and typically each server holds only a portion of the data. If any node fails, the data is available from other nodes. As the database grows, more server nodes can be added to the clusters to scale the operations linearly.

The other defining aspect to the new systems is that data is stored as **key-value** pairs. The key can be any type of data, but must be unique. The data value can be any data element and might be a complex object or collection of items. The key-value concept is important in Web-based applications. For example, each text box on a Web-browser form has a unique ID. When the user submits the form to the server, the browser packages the data and sends it as pairs of the form: ID=value. The browser and Web server programming tools are designed to handle these pairs of data. So, it made sense to build a DBMS that uses this same concept to store and retrieve the data from files. The key-value concept is similar to the primary key in a relational DBMS, but most of the new systems are far more flexible. In particular, the new systems routinely violate the definition of first normal form (storing atomic, single-valued data in one cell). The data stored can be a single item (such as a last name), but it can also contain repeating items such as multiple e-mail addresses.

Retrieving data is quite different from the SQL approach. The most important limitation is that the query systems do not support any type of table JOIN—which is the main reason for the NoSQL name. Data can be retrieved from only one table at a time—by providing the key data. Some additional queries can be supported by predefining indexes on the desired search columns. But queries are limited to improve performance.

This chapter explores these fundamental differences between Cassandra and traditional systems. It explains how the database model design is different and how to handle common situations. A small example of the Pet Store is used to illustrate data storage for a simple Web application. You can download and install a copy of Cassandra from the Web (www.DataStax.com is recommended), and then download and install the sample Pet Store Web database to test the basic concepts.

Two-Minute Chapter

Data storage is the defining difference between traditional relational DBMSs and the new non-relational systems. To improve performance, the new systems require that indexes be defined for every item that needs to be queried later. So the data storage model must be defined in terms of the queries that will be used. The strongest limitation is that the query systems do not support any type of JOIN statement. Data is retrieved from one table at a time. Retrieving matching data from a second table requires writing code to extract a key for the second table and then writing another query to obtain the data from the second table. For even faster performance, data is often duplicated. For example, to avoid a lookup for Customer Name, many designers would store the Customer Name column along with the transaction data so it can be retrieved immediately. The assumption is that disk space is cheap, but Web response delays are expensive because people will leave a slow site.

The DBMSs do not provide referential integrity, so data entered in one table might not exist or match in a second table (unless application code is written to maintain integrity). Similarly, there is no guarantee that each node in the system has the exact same data for each item. A node or connection might fail or updates might be slow so a node might have older data. The goal of the new systems is to emphasize performance over strict data integrity. In their defense, an argument has been put forward that it is probably impossible to guarantee data integrity in a distributed system—without severe performance issues. Interestingly, Cassandra provides the ability to specify the level of consistency desired with each query.

The issue of handling one-to-many relationships is still important, and in many cases it makes sense to create separate tables to handle them. But the new systems often encourage storing multi-valued items in a column or cell (which violates the first normalization rule). Cassandra supports the definition of sets and lists within a single column. Again, the point is that anything stored with the original row key will be retrieved immediately. So any data that is used together should be stored together.

In Cassandra, a keyspace is similar to a schema in that it holds all of the tables for a single application. A Table holds a collection of rows, and each table must have a primary key—preferably based on a single column. One-to-many relationships are handled by defining two columns in a compound primary key. A non-key column can hold single-valued data, or sets, lists, and maps can be used to store repeating data.

Database design is more flexible than the relational model, with few strict rules. Start by normalizing the data into tables and then decide how to duplicate or combine data to improve performance. Ultimately, the design is based on the queries that will be needed by the application.

Cassandra uses the CQL query language to define the database structure and retrieve data. Additional programming (CLI) tools are available but CQL provides commands that are similar to SQL—without the JOINs and with severe restrictions on retrieving data. The CREATE TABLE command is similar to SQL (with different options). The SELECT command is used to specify columns and the table name. A WHERE condition can be used but it can only include columns that are in the primary key or supported by a secondary index.

Non-Relational Databases: Background

What are the main features of non-relational databases? Technically, a non-relational database could be any data storage method that does not support the data normalization rules. In fact, the earliest data storage methods were flat files and hierarchical databases, which were then expanded into network databases (which had nothing to do with LANs or distributed databases). In some ways, the resurgence of non-relational systems is a continuation of the arguments for those earlier data storage methods. A primary argument was that data stored in non-relational systems could be retrieved faster. And in some cases, that answer was true. What Codd successfully argued is that the relational system separated the data so that it was stored more efficiently and provided support for **ad hoc queries**. The relational model also provides tools to ensure data integrity and consistency. So overall, it provides the best performance across a wide range of uses.

However, the relational approach is not necessarily the absolute fastest way to store and retrieve individual pieces of data. In particular, if data is always stored and retrieved in a specific way, it can be considerably faster to optimize the data storage to match the application needs. For instance, Chapter 12 explains the hashed or direct key access method. Given a unique key, its value can be hashed or converted into a specific location address and the data associated with that key can be stored and retrieved almost instantly. But, the application has to always store and retrieve the data using the key. The current non-relational systems utilize this hashed-key approach to store and retrieve data—with a few additional twists.

In a Web application environment data is often collected and transferred in key-value pairs, so it makes sense to store data using the key. Data related to individuals or to specific items is also easy to identify with unique keys. Data storage and retrieval can be optimized for these transactions. The data storage will fail if a manager wants to do a more complex search; but data could be extracted and stored in a data warehouse for purposes of data mining or complex searches.

The problem with Web applications is that users expect instantaneous results. In the early years, Google asked users if they would prefer 10 results or 30. Most people opted for the larger number. Until Google ran actual tests and found that people were dissatisfied with larger results page—as much as a 20 percent drop in usage. The reason: it took a half-second less time to generate the smaller page. Marissa Mayer (then at Google) gave a couple of talks with summaries on the Web such as http://www.zdnet.com/blog/btl/googles-marissa-mayer-speed-wins/3925. The point is that timing can be critical on Web pages, and as the number of users and amount of data increase, delay times can increase exponentially. It is far better to scale up the servers linearly as the number of users increases.

To improve speed, non-relational systems emphasize hashed-key data access, and storing data on distributed servers. Multiple servers are important for scale—adding new servers should improve the overall performance of the data storage. The Web also has geographic implications because of the location of users and bandwidth constraints. Distributed systems are useful because the data can be placed around the world where local sites can respond faster to user requests.

CID	LastName	FirstName	Email
101	Brown	Bobby	BBrown@gmail.com
102	Jones	Jackie	JJackie@live.com
103	Piste	Paula	SkiFast@yahoo.com

Relational table: Primary key (with index).
Atomic cell data, JOINs to other tables.
Fixed columns, all columns searchable.

Key	Value
91e83b31...	LN=Brown, FN=Bobby, E=BBrown@gmail.com
4f763ab4...	LN=Jones, FN=Jackie, E=JJackie@live.com
754d4a...	LN=Piste, FN=Paula, E=SkiFast@yahoo.com

Key-value pairs. Row key is unique and defines storage partition.
Row key is the default way to retrieve a row.
Searching by other columns requires a secondary index.
Data value can be almost anything.
Columns are treated as more key-value pairs and are flexible by row.

Figure 13.1

Relational v. Key-value pair table. Both use indexed primary keys to locate a row, but columns in a relational table are fixed and hold single, atomic values. With key values, rows are retrieved with a row key and the value can be almost anything; including more key-value pairs that are essentially columns.

The challenge with distributed systems lies with maintaining data integrity. Specifically, how can the DBMS ensure that all nodes in the system have the same up-to-date copy of data? Relational systems emphasize the importance of data integrity. Most use locking mechanisms and transaction logs to ensure that data is always accurate across the entire system. But these mechanisms add delays to processing data—particularly storing or updating values. And, in the end, it is still hard to guarantee that every node will always maintain consistent data—particularly if network connections fail.

So, the simple difference with non-relational systems is that they focus on performance and worry less about data consistency. To improve performance, they also limit the ways in which the data can be queried. Consequently, the database design ultimately must be based on identifying exactly how the data will be used and queried. Figure 13.1 shows the main conceptual differences between a table in a relational database and in a key-value pair database. Relational tables have fixed columns with atomic, single-valued cells. Data for rows is retrieved via an indexed primary key, but keys can use several columns. For key-value tables, a primary key is almost always a single column and data is primarily retrieved only by a specific key. The rest of the row value can hold almost anything. In table terms, the columns are treated as another set of key-value pairs. So new columns can be added to any row at any time, simply by adding new key-value pairs to the row data. But, searching for data by any column other than the key requires creating a new index on that column.

Some non-relational systems also provide more flexibility in defining tables—particularly columns. Primary keys identify rows of data, but with some tools it is possible to put anything into a row. Which means that each row might hold different columns and even different types of data. Early proponents of the non-relational approach argued that this flexibility made it easy to expand the database to add new columns and new data later. In a world of Web applications that start small and then add features with each new version, there is some appeal to this flexibility. On the other hand, putting different columns and data into every row is a programmer's nightmare because the code has to continually check to see what data exists for each row. In most cases, it is safer to simply add new columns to tables and ensure that each row is at least somewhat consistent.

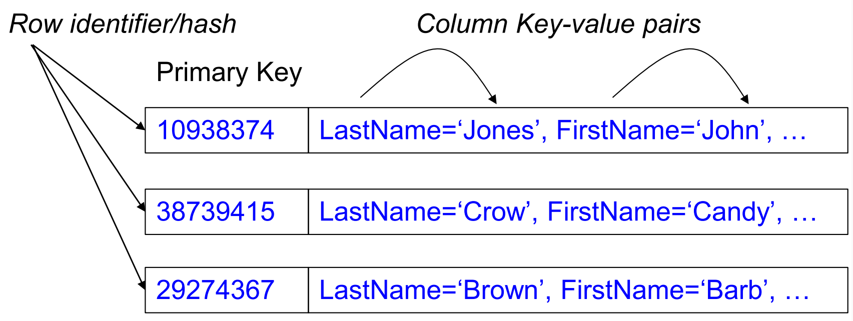
Key-Value Pairs

A key-value pair is probably the simplest data structure available for storing and locating data. Each item to be stored is identified with a key and the key is the only data needed to find the value item. The key could be any data type. A long integer is probably best to ensure that the values are unique, but text values can be converted into numbers through a hashing algorithm. As noted, Web forms make heavy use of key-value pairs, so most Web servers and other tools also use them. For instance, a basic customer Web form would send pairs of data to the server based on the text boxes, of the form: LastName='Jones', Email='John@Jones.com', and Category='Student'.

For storing data in a database, the concept is similar; but a database table has rows and columns. So each row needs to have a unique key entry. In the case of Customer data, it would make sense to invent a CustomerID to use as the **primary key**. So a data row for a specific customer might be stored with a value for CustomerID of 10938374. The remaining "columns" of associated data (FirstName, LastName, Email, and so on) would be stored in the space identified by the primary key value. Figure 13.2 gives an example of the key-value concepts. The primary key is similar to most other lookups, where the key is converted into a storage location. Some tools use indexes, others might use a hashed-key conversion directly to the physical address.

Figure 13.2

Key-value pairs for identifying rows and for extracting column data within a row. The row primary key is just a direct/hashed-key lookup. The column storage shows how key mappings are used to support flexible rows that can hold different data.


Row identifier/hash
Column Key-value pairs
Primary Key
10938374
LastName='Jones', FirstName='John', …
38739415
LastName='Crow', FirstName='Candy', …
29274367
LastName='Brown', FirstName='Barb', …

The interesting twist with non-relational systems is that they might also store the column or cell data using key values where the key is the name of the column. Instead of allocating space for each column, the data storage consists of a mapping array that retrieves a value based on the key. With the sample data, data might be retrieved by specifying CustomerID=10938374. Then the application requests the value on that row associated with the key of 'LastName' or 'FirstName'.

Obviously, the row keys need to hold unique values. The column names also have to be unique, but in most cases those are predefined. The challenge with row keys becomes more difficult in the distributed environment of most non-relational systems. Think about the challenge of inventing a key number that has to be unique across all of the servers. A relational DBMS probably has a key-generation method that creates incremental values. It might store the latest ID values in a table and then generate the next value on demand. However, this approach requires that all servers have access to the same consistent table data. The non-relational approach avoids enforcing consistency, so a different method is needed to create ID values. The most common approach is to use a number known as a **universally unique identifier (uuid)**. These numbers have been used for several years for similar purposes, so software exists to generate them reliably on almost any device. Microsoft has used a variation known as globally unique identifier or GUID, but an ISO standard now exists. By the standard, a uuid is a 128-bit number represented by 32 hexadecimal digits and written in standard form with hyphens, such as:

```
71c1da88-88af-4217-aa41-332ea3d33ae9
```

Several methods have been defined to generate uuid values. The earliest ones (type: Version 1) used the MAC address of the computer's network card and a measure of time. Because each network card is assigned a unique MAC address by the vendor, the UUID generated is known to be different from one generated by any other computer. Other methods also exist, including purely random numbers (Version 4), which could result in duplicates with a tiny probability; or (Version 5) numbers generated with security hash algorithms. In any case, primary programming languages all have algorithms to generate uuid values. The drawback to uuids is that they are a pain to type if you want to manually test a query. But they are necessary in distributed computing environments, and most of the time a programming language generates the value or retrieves it from an existing table.

Sparse Data and Flexible Columns

The second part of Figure 13.2 hints at how columns in non-relational systems are different from those in relational tables. In a relational DBMS, each table has a fixed set of columns and each row/column cell has exactly one value. Once the relational rules are discarded, it is easier to think of a row as just a collection of bytes. The row key retrieves those bytes, but the application can store or retrieve almost anything in that space. Most systems define the column space as a data map which is just another set of key-value pairs.

One benefit to the mapping approach arises for data with many missing entries, or **sparse** tables. Each row only stores the data columns that exist so if much of the data is missing no space is wasted. For instance, one Customer entry might have values only for LastName and FirstName, so the map contains only those two entries. Another row might have several items, including a photograph. Application do have to be slightly cautious and test for missing values when requesting items within a row.

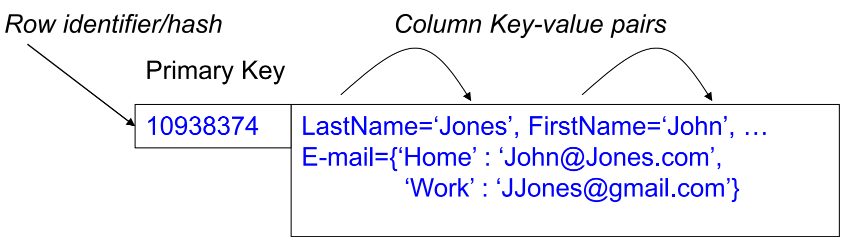

Row identifier/hash
Column Key-value pairs
Primary Key
10938374
LastName='Jones', FirstName='John', …
E-mail={'Home' : 'John@Jones.com',
'Work' : 'JJones@gmail.com'}

Figure 13.3

Data collection map for e-mail addresses. Multiple values can be stored in the E-mail column, but the key-values are defined and handled by the application. Using collections is similar to using XML. It is more flexible but requires more application code to handle the data storage and retrieval details which increases the programming difficulty and probability of introducing errors.

Some systems and some developers take this approach to the extreme and claim that the flexibility enables them to store different key items in every row. For example, one Customer row might contain entries for LastName, FirstName, and Phone, while another could hold data for FamilyName, Nickname, and Skype address. Even if a system does support this level of flexibility, it should be avoided. Changing column names/keys means that every application needs to know all of the possible values and test for them within the code. Making data storage more "flexible" at the cost of making program code harder to write, read, and test is a bad tradeoff in most situations.

Another approach to flexible column data is the ability to store complex data within a single cell (or column). For example, Cassandra supports set, list, and map data types. A **set** is an unordered collection, a **list** has an index order (1, 2, …), and a **map** is a collection of key-value pairs. Each of these can be used to store multiple entries in a single column for one row of data. They should be used only for small lists because the query system will retrieve every value at the same time.

Figure 13.3 shows an example where a set or list might be used to enable customers to enter multiple e-mail addresses or multiple phone numbers. For instance, to store two e-mail addresses, a set could define

```
E-mail = {'John@Jones.com', 'JJones@gmail.com'}
```

A map uses a key to define the difference between the two addresses such as

```
E-mail = {'Home' : 'John@Jones.com', 'Work' : 'JJones@gmail.com'}.
```

Collections support any data type and the map can contain any key definitions needed by the application. But, the application programmers need to remember all of the keys and handle them within the program, so the programming can become more complex and subject to more errors. In some ways, the collection types are not radically different from the relational DBMS. Most relational systems today support an XML data type which makes it possible to store complex data, including lists and collections in a single column. And the same warnings apply to using the XML data types—they make the application programming more complex and harder to test and debug.

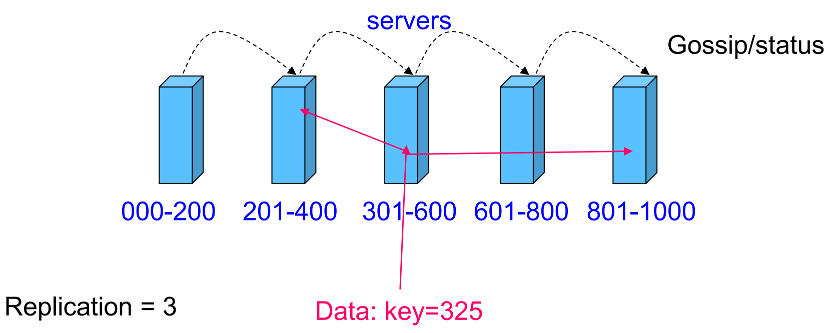


Figure 13.4

Cassandra data storage overview. Servers are configured as (virtual) nodes. They communicate with each other via gossip for status (every second). A data partitioner assigns data to an initial server based on key value. The replication parameter specifies the number of copies.

Distributed Data

The non-relational systems are built from the ground up to handle highly distributed data. Cassandra has a particularly interesting version because it is peer-to-peer instead of using a central server approach. Every server node in the Cassandra network is independent and shares data directly with other nodes. No single node coordinates or controls the others. Figure 13.4 shows the basic elements of the Cassandra network. A data partitioner defines a range of key values for each node. When the database is created, the designer specifies a replication level—3 in this example. Data is then written to the appropriate server based on the value of the key and replicated to the specified number of servers. Each node communicates with the others via a **gossip** channel to share status information. If a server fails to respond, it is moved from the active list and others pick up the lost key range. Similarly, when a new server is added, the key ranges are redefined and gossip is used to synchronize the data updates across the new server.

Distributed data in Cassandra is actually much more sophisticated, but most of the details are not important to the design or queries so they are not covered in this chapter. Basically, servers can support virtual nodes which simplifies replication assignments. More interestingly, it is possible to incorporate the physical layout of the servers into the replication design. For example, servers located in the same rack are connected by high-speed networks and can quickly share data; but they are more at risk for collective failure (e.g., power or network). Nodes in a different data center might be a different geographical location, so data is more protected if spread across centers, but updates are slower. Cassandra data models support defining these characteristics and the data partitioner optimizes the data replication.

Other systems have different features, and some rely on a central server to coordinate data storage and status messages. With Cassandra, data queries and write operations can be connected to any node and the system will function the same way each time, even if one node crashes or becomes inaccessible. Most of the non-relational systems use some form of distributed data storage; both to provide data protection through replication and to improve performance by having multiple servers and multiple drives handling the data.

In terms of physical computers, the server processing is important, but the data storage methods are more important. Because Cassandra automatically handles replication of the data, RAID 1+0 drives are not recommended. RAID 1+0 drives make physical replicas of data being written so if one drive fails the others can rebuild the content. But Cassandra already handles the replication so using RAID 1+0 just wastes space. RAID 0 drives are still useful because they multiply the access speeds with physically independent disks. But high-end Cassandra implementations still recommend even faster **solid state drives (SSDs)** for all data storage.

Other than performance and backup, the nice feature of the distributed systems is that they are invisible to the application. The application (writing and retrieving data) just issues queries and the DBMS handles all of the details automatically. Of course, setting up and monitoring the distributed network takes additional time. But, application transparency is important because the data storage can be rescaled at any time without altering the application.

Consistency and Integrity

Largely because of the distributed structure, one of the key aspects of non-relational databases is the limitations on consistency and integrity. A key strength of traditional relational systems are the built-in controls to ensure data consistency; which makes them valuable for business transactions. The problem is that absolute consistency is difficult to guarantee in a widely distributed system. It would require that all nodes maintain communication during all updates. Worse, strict consistency can require that some transactions (reads and writes) be delayed until all nodes are consistent. But the point of a distributed system is that it should be able to handle short-term failures in some nodes and connections; and maintain high performance even under heavy load.

Figure 13.5

Cassandra tunable consistency. Developers can choose a consistency level for any write (or read) operation. The lowest level (ANY) has the least delays. The ALL level requires all replicas to be updated before continuing.

Level	Nodes	Description
ANY (lowest)	1	Write will still succeed if a hinted handoff has been written.
ONE, TWO, THREE	1, 2, or 3	Write must be logged and committed to the specified number of replica nodes.
QUORUM	Replication/2 + 1	Write logged and committed to at least half the replication nodes.
LOCAL_QUORUM	Same data center	Same as quorum within the local data center.
EACH_QUORUM	All data centers	Same as quorum within all data centers.
ALL (highest)	All replicas	Write must be logged and committed to all replicas.

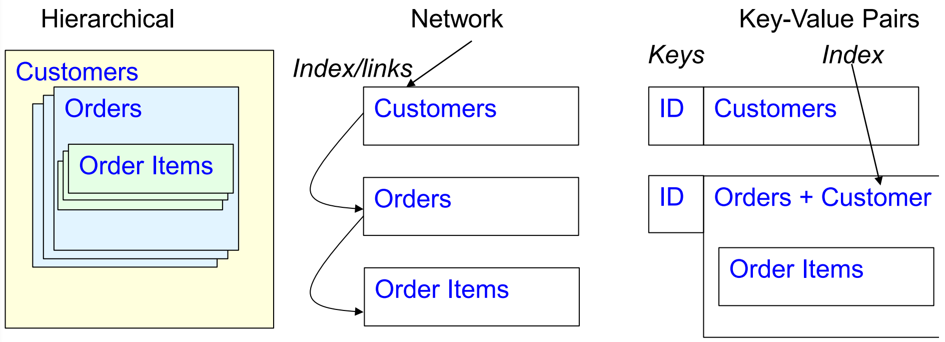
Non-relational systems relax the constraints on absolute consistency and allow nodes to be inconsistent—at least for a short period of time. Actually, in terms of read and write transactions, Cassandra provides the ability for developers to specify the desired level of consistency, calling it **tunable consistency**. As shown in Figure 13.5, write consistency specifies the number of replicas that need to return an acknowledgement of success. The lowest level (ANY) operates with the least delays. The highest level of consistency (ALL) requires all replicas to be updated and committed before continuing. It is similar to the consistency requirements in traditional relational systems.

However, several consistency issues exist beyond read and write transactions. First, non-relational systems do not support referential integrity. The DBMS does not have a method to verify that foreign keys are valid. For example, a CustomerID entered into a Sale table could be wrong. Similarly, when a row is deleted from the Customer table, there is no automatic mechanism to delete the corresponding data in tables that use that data (cascade delete). So the programmer is responsible for maintaining data integrity.

A second consistency issue arises because non-relational designs often duplicate data to improve performance. Consider the standard Sale and Customer tables again. Instead of relying on the CustomerID to look up the customer name in the Customer table, many designers will duplicate and store the customer name in the Sale table. That way the name (and other data) can be retrieved at the same time the sale data is read, without requiring an additional lookup in the Customer table. But again, the DBMS has no method to ensure consistency of data. Changing the address in one location does not affect the others. This action might have some use—the sales data could contain different addresses for a customer depending on when the sale was made. Many Web databases rely on keeping different values of data at different points in time. But users do need to remember that the data can be inconsistent at times.

Figure 13.6

Non-relational storage affects how data can be retrieved. Hierarchical systems stored and located data by starting at the top level and working down. Network allowed more flexibility by separating the tables and linking them through indexes that had to be built to support queries. Key-Value combines elements of both by using indexes on keys to locate individual rows. Any other searches require additional indexes.



Optimizing Data Storage for Queries

Figure 13.6 shows that the original non-relational DBMSs (hierarchical and network) were relatively rigid in the way data was retrieved. Hierarchical models adopted features of paper filing cabinets. A cabinet (database) would hold folders of Customers stored alphabetically. Each folder would contain the individual orders, and the orders would contain the detailed items purchased. As long as you only wanted to retrieve data by Customer and then find individual orders, the system was relatively fast. But, if you wanted to find customers who ordered a specific product, the system would have to start at the top and go through every customer and every order. The Network model attempted to support these additional searches by separating the storage of each table and then building indexes and links to all of the data. So, if the developer knew in advance that someone might want to search for customers who ordered a specific product, an index could be built on the ItemID, and then the back-links could be traced to identify the specific customers.

The newer key-value systems adopt some elements from both of these models (as well as a couple from the relational model). Data stored in separate tables is indexed by the primary key. Using multi-level indexes and the power of distributed data, the storage and retrieval of the associated row data is fast. For even faster access, data is often duplicated. For example, designers might include the Customer name and shipping address with the Order data. Likewise, the Order Item data might be stored within the Orders row, similar to the way it would be handled with a hierarchical model. The critical design concept to remember is that the DBMS can only retrieve data using the primary key. In fact, queries probably cannot use other data in WHERE conditions. This limitation is demonstrated in the query section. However, a few systems, particularly Cassandra, support the creation of additional indexes that can be used for searching. In the example, if the developer knows that someone will want to search for customers who ordered a specific product, a separate index can be created using ItemID on the Orders table.

The most important point of this section: Unlike the relational data normalization rules, there is no fixed method for defining data storage using key-value pairs. Instead, the designer must know how the data will be generated and queried and then design the data storage to optimize the overall performance. Figure 13.7

Figure 13.7

Non-relational storage affects how data can be retrieved. Hierarchical stored and located data by starting at the top level and working down. Network allowed more flexibility by separating the tables and linking them through indexes that had to be built to support queries. Key-Value combines elements of both by using indexes on keys to locate individual rows. Any other searches require additional indexes.

1. Identify the basic data to be stored.
2. Do a base data normalization to identify potential tables.
3. Identify all the ways an application will need to query the data.
4. Identify the primary key-value pairs (base tables).
5. If needed, duplicate data to improve performance.
6. Create additional indexes to support queries not covered by primary keys.
7. Test performance, combine data and reduce indexes if needed.

defines the basic steps that can be used to design data storage for a key-value DBMS. But, each design will be unique and require experimentation to find the best storage approach. The basic rule is that any data that can be accessed via a key will be relatively fast. Storing all related data in one row is faster—even if it means duplicating some information. Searching on non-key items requires creating additional indexes, but adding indexes slows down performance on updates and inserts because the indexes have to be rebuilt. If a design for a transaction system starts to require dozens of indexes, it will probably be better to eliminate all but the essential indexes and create a data warehouse to enable managers to perform additional searches on a copy of the database (see Chapter 9).

Ultimately, getting the best performance out of a key-value pair database requires experimentation with the design. Eventually, as the system software grows and stabilizes, perhaps computer scientists will be able to develop rules to improve the designs.

Cassandra

How are databases designed and queried using Cassandra? All of the key-value pair DBMSs are slightly different and each application requires a custom database structure. Although the general elements are similar, it is important to look at a specific DBMS and a specific problem to understand the features and constraints of the tools. Cassandra is one of the leading non-relational DBMSs, with strong developer support including a company (www.DataStax.com) that specializes in advancing the software and providing support. This level of support is useful to help ensure the DBMS will survive for at least a few years. Remember that these tools are relatively new and many companies are experimenting with different approaches. A second useful feature of Cassandra is that it has an interactive query system that makes it possible to experiment with the database without needing to write code for each example. Ultimately, each application still has to be written in some programming language, but it is helpful to be able to test designs and queries before writing code.

Installation Issues

One of the strengths of the non-relational systems, particularly Cassandra, is the ability to run as a distributed database on multiple server nodes. A drawback to running multiple servers is the cost of the servers—both hardware and software licensing costs. Consequently, most of these tools are open-source projects that are also designed to run on open-source systems—reducing the licensing costs. (Some of the hardware costs can be handled by using cloud-based computing as described in the last section of this chapter.) A challenge with open-source operating systems is that they can be harder to install and manage than Windows-based systems. Also, several variations exist, leading to differences in installation and operating procedures. Cassandra is written in the Java programming language, which means that versions have been compiled to run on most systems (including Windows—but it is not recommended). Cassandra also requires the Python programming language (for the query tool).

It is highly recommended that Cassandra be installed on a virtual machine running an open-source operating system. The Debian version is relatively straightforward to install and it uses packages to install most software which simplifies installation of applications and tools. DataStax has packaged versions and instruc-

1. Install Virtual Machine Server—open source: Debian
http://www.debian.org/releases/stable/installmanual
2. Sun/Oracle version of Java: at least JRE and JNA
   a. Java –version (default is open source Java)
   b. Download and install from Oracle, then set as default
http://www.oracle.com/technetwork/java/javase/downloads/index.html
3. Download and install Cassandra from DataStax (Community edition)
4. Several configuration steps for production are not needed for the sample and testing. And only one node is needed.
   a. Download and install the PetStoreWeb files.
   b. Unzip and copy them to a folder
   c. In terminal mode, run the cql command to install:
      cqlsh –f PetStoreWeb.txt

Figure 13.8

Summary installation steps. Follow the detailed installation steps in the Apache Cassandra Documentation from DataStax: http://www.datastax.com/docs. Be certain to install the recommended version of Java before attempting to install Cassandra.

tions for a couple of the more-popular Linux variants including Debian. Figure 13.8 outlines the basic steps, but several details can be required for each step so you might want to obtain assistance from a local Linux user. Installing the proper version of Java and setting it as the default version is one of the more complex steps.

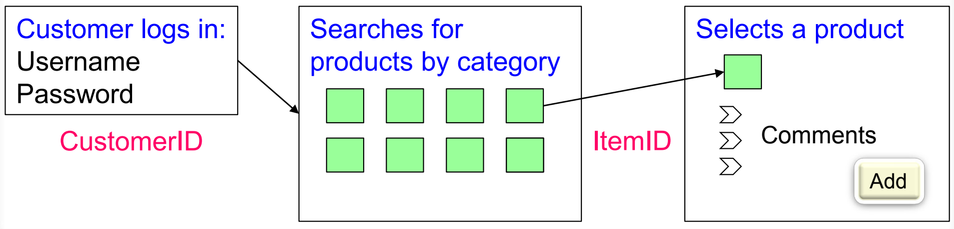
Cassandra has several useful configuration options for production systems. These options are used to initialize and coordinate the multiple nodes in the distributed system. They are not needed for the sample demonstration files—which are tiny. They are critical for optimizing performance in large production databases, but they focus on the distributed networking issues so are not covered in this chapter.

Pet Store Web Example

Many of the non-relational benefits arise when millions of people need to store and access data—particularly on the Web. Figure 13.9 shows a common extension to the Pet Store case. Customers will choose a category and see items in that

Figure 13.9

Pet Store Web Site Usage. Customers see merchandise items based on a selected Category. When an item is selected, the page displays the product, description, price, and a set of comments from other customers. Customers who are logged in can add their own comments.



- Find CustomerID given the Username
- List Merchandise given a Category
- Display Merchandise data given an ItemID
- List all comments and customer screen name for a specified ItemID
- Insert a new comment given ItemID and CustomerID

Figure 13.10

Initial application queries. These queries will affect the database design. Lookups by ID are handled as primary keys. Other lookups will require additional indexes.

category. When they select a specific item, the details of the product are shown along with a set of comments entered by other customers. Once the customer is logged in, he or she can add a comment to the list. A customer can make only one comment on a given product, but some might want to change the comment later. The basic process should be familiar, because many Web sites support comments by users. With potentially millions of customers and their comments, the database could become large. It is also important that the page displaying a product be retrieved and generated quickly—customers will not tolerate delays.

Think about the usage display for a minute in terms of data. The existing (relational) database already has tables for Customer and Merchandise. Those basic tables will probably transfer cleanly—but the ID values will have to be changed to uuids. A new table will probably have to be defined for the Comments, and the details are covered in the next section. But also think about the potential queries needed by the application. Figure 13.10 shows the main queries that will be needed by the application. These queries are important because they will affect the design of the database. Queries using IDs will become key-value pairs or primary keys, and the others will have to be handled by separate indexes.

Database Design

Figure 13.11 shows the three tables needed for the Pet Store Web application. The Customer and Merchandise tables are essentially copied from the relational design, except that the primary keys are uuids. The ItemComments table is new. Notice that the keys are the same as they would be in a relational database: ItemID + CustomerID. Also, notice the duplication of the ScreenName in the ItemComments table (from the Customer table). By placing this small piece of data that is displayed with the comment inside the comment table, it saves a lookup into the Customer table so it can be retrieved significantly faster. But the application will be responsible for maintaining data consistency.

To understand the challenges of design, consider the ItemComments table. The first question to ask: Why is it a separate table? Why not just store the comments within the Merchandise table? It would be straightforward to add a column to the Merchandise table that could hold a repeating set of data as comments. The drawback to storing all comments in one row of the Item table is that retrieving a row of data always retrieves the entire row. If a popular product gets thousands of comments, it could take too long to retrieve that one row. Additionally, it is more difficult to include the other attributes (CustomerID, Date, Rating, and so on). It can be done—but only by creating a fairly large mapped object inside each row, which slows down retrieval and processing. Using both ItemID and CustomerID as keys also makes it straightforward to search for comments by Customer.

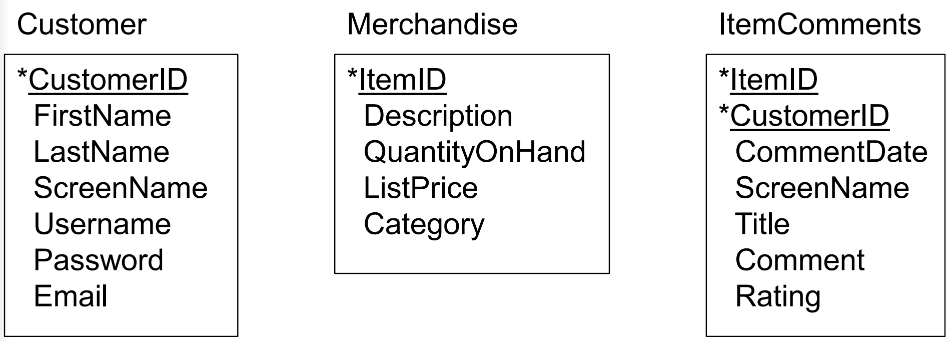


Figure 13.11

Data tables for Pet Store example. Customer and Merchandise are base tables and the ID key columns are uuids. ItemComments are new and each customer can comment once on a given item (but can change the comments later). Notice the duplication of ScreenName in the ItemComments table.

Before creating the tables, Cassandra requires that all tables be defined within a **keyspace**, which is similar to a schema in Oracle or SQL Server. It simply separates one collection of tables from another by assigning a name. Generally, a keyspace is defined for each application. The syntax to create a keyspace is straightforward, as is the command to switch to a new (or different) key space:

```
CREATE Keyspace PetStoreWeb;
USE PetStoreWeb;
```

Tables are created within a keyspace. In earlier versions, and in some existing documentation and error messages, a table was called a **column family**. In current versions, the two terms are synonyms, but it is easier to think of the data as a table than a family. Tables hold **columns**, which are not exactly the same as SQL columns; but in most situations, they are similar. The differences are greater in terms of data storage (and keys) because the columns are actually stored as key-value pairs within each row.

Note that keyspace, table, and column names in Cassandra are normally not case sensitive. The key words (CREATE) are also not case sensitive. But this book uses case to highlight the key words and names to make them easier to read. There are two catches in Cassandra: (1) Double quotes placed around a name make it case sensitive and quotes are then required in all future usage. (2) Cassandra automatically converts all names to lower-case when it stores them. A few procedures (notably COPY) seem to automatically use quotes, so if a command does not work, try entering all names in lower-case.

Figure 13.12 shows the CREATE TABLE commands used to define the tables in Cassandra. The syntax of the command is similar to that in SQL but it has different data storage options, which are not shown here. The default options are fine for the sample database. Also, notice the uuid data type for each of the ID columns. These values will have to be generated by the application. Note the specification of both columns as the PRIMARY KEY for the new ItemComments table. Although this syntax looks similar to SQL, the effects are quite different as will be explained shortly. Finally, notice the use of the set<text> definition for the e-

```
CREATE TABLE Customer(
  CustomerID      uuid,
  FirstName       varchar,
  LastName         varchar,
  ScreenName      varchar,
  Username        varchar,
  Password        varchar,
  Email           set<text>,
  PRIMARY KEY (CustomerID)
);
CREATE TABLE Merchandise (
  ItemID          uuid,
  Description     varchar,
  QOH             int,
  ListPrice       decimal,
  Category        varchar,
  PRIMARY KEY(ItemID)
);

CREATE TABLE ItemComments(
  ItemID          uuid,
  CustomerID      uuid,
  CommentDate     timestamp,
  ScreenName      varchar,
  Title           varchar,
  Comment         varchar,
  Rating int,
  PRIMARY KEY (ItemID, CustomerID)
);
```

Figure 13.12

Data tables for Pet Store example. Customer and Merchandise are base tables and the ID key columns are uuids. ItemComments are new and each customer can comment once on a given item (but can change the comments later). Notice the duplication of ScreenName in the ItemComments table.

mail address in the Customer table. Defining it as a set means that the table can hold multiple e-mail addresses for each customer. It is still up to the application to handle the collection and editing of that data and it slightly complicates the syntax for storing them, but it demonstrates the additional flexibility of collections.

Figure 13.13 shows the primary data types available in Cassandra. For business applications, the most common data types are the standard uuid, int, varchar (or text), decimal (for currency values), timestamp, boolean, and possibly float (for percentages). Avoid the more exotic types of counter and varint, and you should almost always use varchar instead of ascii, which does not support international characters. Collections are defined with the set, list, and map keywords followed by the type of data that will be stored. Most collections will use the text data type.

Primary Keys

The Primary Key definition syntax is similar to that used in SQL. However, primary keys are considerably different in Cassandra than in relational databases;

Data Type	Description	Data Type	Description
ascii	US ASCII text string	inet	IP address as string
bigint	64-bit signed integer	int	32-bit integer
Blob	Binary object/picture	text or varchar	UTF-8 string
boolean	true/false	timestamp	Date+ time, 8 bytes
counter	64-bit integer, but…	uuid	Type 1 or 4 uuid
decimal	variable precision decimal	varint	Arbitrary-precision int
double	64-bit floating point	Java classes	Optional classes in Java
float	32-bit floating point		

Figure 13.13

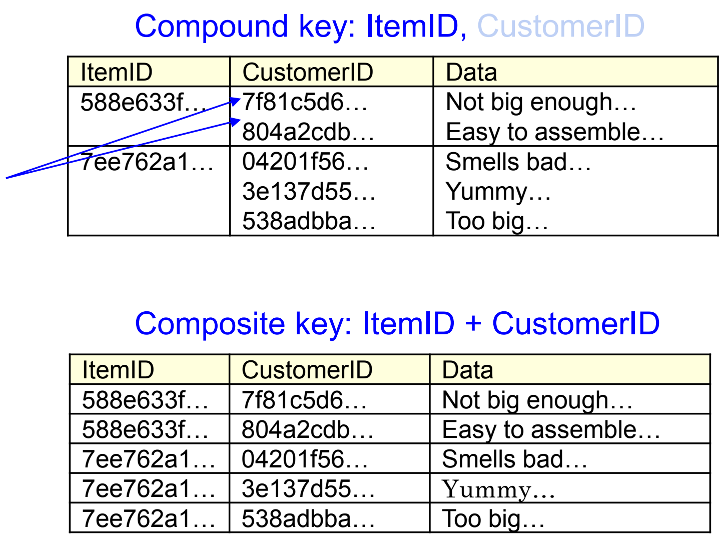
Cassandra data types. The most commonly used types in business applications should be: uuid, int, varchar (or text), decimal (for currency), and timestamp.

particularly when the primary key contains more than one column. Recall that data is stored as a key-value pair. Specifically, each row must have a key that is used in the index to find a specific location. In case you are curious, that key cannot be a counter type, and uuid is by far the most common.

A critical difference with Cassandra is that when the primary key consists of multiple columns or a **compound primary key**, only the first column is used as the partition key—which determines where data is stored. The other columns are clustering columns and data is stored together. In fact, the clustering columns are used to sort the data. In the ItemComments example of PRIMARY KEY (ItemID, CustomerID), the ItemID determines where the data row is stored, and the comments are stored sorted by the CustomerID. Depending on the application goals, it might be useful to change the keys to: PRIMARY KEY (ItemID, CommentDate). This definition would store the comments in order of date, making it easy to retrieve and display them in that order. However, the timestamp data type only splits time down to seconds, so the application would have to be careful to ensure that no two comments are ever written with the exact same date and time.

In some situations, it can be useful to partition the row data on more than one key value. The **composite primary key** is used to define multiple columns as the partitioning key by using a second set of parentheses, such as PRIMARY KEY ((ItemID, CustomerID), optional columns). The difference with a composite key is important. Think of it as defining the data storage by both keys: ItemID + CustomerID. Without the parentheses, only the first column partitions the storage, with the extra parentheses both keys define the storage location—and both are required to retrieve the data. Because Cassandra retrieves all rows based on the partition key, the difference affects the queries.

With a simple compound key (ItemID, CustomerID), a query would retrieve data by specifying ONLY the ItemID value, which would return all of the comments made by each customer. With a composite key ((ItemID, CustomerID)), a query requires BOTH the ItemID and CustomerID values to return exactly one row. The Pet Store application has to use the simple compound key, because when it displays an item, it knows only the value of the ItemID, not all of the Custom-



Compound key: ItemID, CustomerID

ItemID	CustomerID	Data
588e633f…	7f81c5d6…	Not big enough…
	804a2cdb…	Easy to assemble…
7ee762a1…	04201f56…	Smells bad…
	3e137d55…	Yummy…
	538adbba…	Too big…

Composite key: ItemID + CustomerID

ItemID	CustomerID	Data
588e633f…	7f81c5d6…	Not big enough…
588e633f…	804a2cdb…	Easy to assemble…
7ee762a1…	04201f56…	Smells bad…
7ee762a1…	3e137d55…	Yummy…
7ee762a1…	538adbba…	Too big…

Figure 13.14

Compound v. Composite key. The compound key partitions by the first column ONLY. A query specifies just the value for ItemID and returns comments by all customers for that item. A composite key partitions by both columns. A query must list both the ItemID and CustomerID values to retrieve exactly one row. The problem is that there is no easy way to get the list of all CustomerID values in the case of the composite key.

erIDs who have entered comments. And realistically, there is no way to obtain the list of CustomerIDs—without testing every possible value, which would be horribly slow.

Figure 13.14 illustrates the difference using some of the sample data. The compound key uses only the first column (ItemID) to partition (store and retrieve) the data. So a query needs to know only the value of the ItemID and it will return comments from all customers in that "row." The composite key uses an extra set of parentheses to partition by both the ItemID + CustomerID columns. A query needs values for both ID columns to retrieve exactly one row of data. The composite key approach is faster—if the application always has values for both ID columns. In the Pet Store example, the usage description says that the application knows only the ItemID, so the design needs to use a compound key based only the ItemID column.

Some of these points might seem a little confusing at the moment. Do not panic. They are easier to understand once you see the limitations of queries as explained in the next section. So, read the section on queries and then come back and re-read the design guidelines. Remember that data design and storage depend on the queries that need to be answered, so the design (and learning) process is iterative.

Initial Queries

The real differences with a non-relational DBMS arise when looking at queries—which partly explains the misnomer: NoSQL. Interestingly, Cassandra now has an interactive query language named **CQL** (Cassandra Query Language). CQL

```
SELECT Count(*)
FROM Customer;
```

```
count
--------
   99
```

```
SELECT * FROM Customer
WHERE CustomerID=71c1da88-88af-4217-aa41-332ea3d33ae9;
```

```
customerid     email                                                firstname   lastname…
----------------+---------------------------------------------------+------------+---------------+
71c1da88… |  {BCummings@gmail.com, bignotes@gmail.com} |   Brent    |  Cummings |
```

Figure 13.15

Two basic CQL queries. The basic CQL syntax is similar to SQL but much more limited. Count is the only aggregate function supported. The SELECT clause lists columns to retrieve and the WHERE clause can be used to specify primary key entries.

borrows some of the basic structure of SQL, which makes it a little easier to write the syntax, but ultimately, the queries have almost nothing in common with even simple SQL queries. But it is not just a limitation of the query system. The restrictions arise because of the way the DBMS stores the data, which is done to improve performance for specific queries. This section shows some basic queries using the sample Pet Store Web data. A different approach is used here compared to learning SQL: Several of the initial queries will not work—specifically to demonstrate the limits. If at all possible, you should install Cassandra and the sample database and run the queries to follow along. Start the CQL processor by opening a terminal window and typing: cqlsh (for CQL shell). Remember to enter the command to use the keyspace: use PetStoreWeb; Note that the use of uuids makes it a challenge to type some of the queries.

Figure 13.15 shows two basic CQL queries using the SELECT command. The first uses the Count function to return the number of rows in the table. Count is the only aggregation function supported by CQL, but it can be useful to identify large tables. The second query looks similar to a simple SQL query. The SELECT clause can use * for all columns or the names of individual columns can be entered. The WHERE clause is even more restrictive. Initially, the only conditions you can enter in the WHERE clause are conditions on the primary key (CustomerID in the example). Remember that rows are stored as key-value pairs and initially data can be retrieved only through the primary key.

Figure 13.16 shows some basic queries to experiment with variations of the WHERE condition in the SELECT command. The bottom line is that the WHERE clause can contain conditions that only use the primary key and an equals sign. It does not even support conjunctions (And, Or). However, it does support the IN ( ) condition which takes multiple key values and finds matching rows based on equality—which is equivalent to several OR conditions. A token ( ) function exists which does support inequality conditions. However, the token function converts the values to their hashed-storage values and then makes the comparison. The default hashing function essentially randomizes the values, so the results are usually

```
SELECT * FROM Customer WHERE
CustomerID= 71c1da88-88af-4217-aa41-332ea3d33ae9 OR
CustomerID= 378feb73-34cd-451f-90a9-a739a94c30f4;

>>> Error: Expected EOF at OR…
```

```
SELECT * FROM Customer WHERE CustomerID IN
(71c1da88-88af-4217-aa41-332ea3d33ae9,
 378feb73-34cd-451f-90a9-a739a94c30f4);

>>> Retrieves two rows.
```

```
SELECT * FROM Customer
WHERE CustomerID > 71c1da88-88af-4217-aa41-332ea3d33ae9;

>>> Error: Must use EQ or IN
```

```
SELECT CustomerID, LastName FROM Customer
WHERE token(customerid) > token(00000000-0000-0000-0000-000000000000);

>>> Retrieves random rows where the hash value is greater than the hash of 0…
```

Figure 13.16

Experiments with CQL SELECT. Initially, a table can be searched only by individual values of the primary key. Conjunctions (Or, And) and inequalities (<, >) are not allowed. The IN (…) condition is used to find multiple values in one command. The token ( ) function does support inequality values but the comparison is made based on the hashed value of the key which is probably random.

meaningless. However, Cassandra does support a ByteOrdered partitioner, which arranges tokens in the same order as the keys. If this partitioner is specified as the storage mechanism when the table is created, the token function might be useful.

The purpose of the examples is to demonstrate the constraints of the query system. Although the SELECT command might look a little like simple SQL, it is far more limited. Remember that the data storage places strong limits on what can be done to retrieve data. At the moment, the SELECT command can retrieve only data based on specified values of the primary key.

Indexes

Obviously, retrieving data based only on primary keys is too restrictive. Look again at the usage goals for the Web site. At a minimum, it requires finding a Customer based on Username, and retrieving Merchandise based on the Category value. Neither of these columns is in the primary key. In fact, Category could never be a primary key column because it is not unique. So how can Cassandra retrieve data using those conditions? The answer is to create indexes. An index is basically just another set of key-value pairs.

```
SELECT * FROM Merchandise
WHERE Category = ‘Cat’;

>>> Error: No indexed columns present…
```

```
CREATE INDEX idxMerchandiseCategory
ON Merchandise (Category);
```

```
SELECT Category, Description, ListPrice
FROM Merchandise
WHERE Category = ’Cat’;
```

```
category     description           listprice
---------+----------------------+----------
     Cat |          Cat Bed-Small |       25
     Cat |    Cat Litter-10 pound |        8
     Cat |  Cat Food-Dry-10 pound |       10
     Cat |   Cat Food-Dry-5-pound |        7
     Cat |                Cat Toy |        3
     Cat |  Cat Food-Dry-25 pound |       18
     Cat |   Cat Food-Can-Regular |      0.5
     Cat |             Brush-Soft |        8
     Cat |   Cat Food-Can-Premium |        1
     Cat |         Cat Bed-Medium |       35
     Cat |        Flea Collar-Cat |        6
     Cat |             Collar-Cat |        8
     Cat |     Litter Box-Covered |       15
     Cat |             Litter Box |        8
```

Figure 13.17

Creating an index to search by non-key columns. The CREATE INDEX command builds an index that can be used to add new conditions to a SELECT statement. The application requires searching by Category.

Figure 13.17 shows how to create an index on Merchandise Category so that the application can retrieve all items that match a specified category. The CREATE INDEX syntax is similar to SQL:

```
CREATE INDEX indexName ON table (column);
```

Technically, the index name is optional, but it should always be used because then the DROP INDEX command can be used to remove it later. Note that primary keys cannot be indexed—but it would not make any sense to do that. After the index has been created, the specified column can be used in the WHERE clause of a SELECT query—but only with an equals sign. In production databases, the CREATE INDEX command should be issued when the tables are CREATED and before data is loaded.

Figure 13.18 shows an interesting effect of using an index. Remember that only the Category column has been indexed. Yet, now the SELECT statement supports additional conditions in the WHERE clause—as long as the condition applies to a non-key column and the **ALLOW FILTERING** clause is added to the query. CQL will provide a warning if the ALLOW FILTERING clause is missing. Inequality searches are potentially expensive and slow, so be certain that they are

```
SELECT Category, Description, ListPrice
FROM Merchandise
WHERE Category = 'Cat'
AND ListPrice > 10
LIMIT 10
ALLOW FILTERING;
```

```
category     description                listprice
---------+-----------------------+----------
      Cat |          Cat Bed-Small |      25
      Cat | Cat Food-Dry-25 pound |      18
      Cat |         Cat Bed-Medium |      35
      Cat |     Litter Box-Covered |      15
```

Figure 13.18

Secondary indexes enable additional conditions. Conditions on other (non-indexed) columns can be added as long as the ALLOW FILTERING phrase is added at the end. The LIMIT n command can be used in any SELECT query and defaults to 10,000 rows if not specified.

necessary before using them. It is often useful to include the **LIMIT** statement to restrict the number of rows returned. In fact, Cassandra has a default value of 10,000 for the number of rows returned for any query. Queries that might return more rows need to use the LIMIT statement to increase that value. But, before blindly inserting a large number, ask yourself why you need a query to return so many rows. No one is going to read that many, and a large value would slow down almost any Web site. The statement would be useful when it is necessary to extract large chunks of data to transfer to other systems, but small values would be used in production applications.

Note that the additional clause (AND ListPrice > 10) can be used only if the Cat condition is used. Try running the SELECT query with just the ListPrice condition and it will generate an error (no index). If a new index is created for ListPrice the inequality condition by itself (ListPrice > 10) still will not work—because indexed columns can be searched only using equality conditions. The basic SELECT search rule is that a WHERE clause can search for only primary keys

Index Issues

Technically, indexes are supposed to be built on existing data as soon as the CREATE INDEX command is issued. However, some queries in testing returned no matching values after the index was created (Cassandra 1.2). In production situations, it is best to create all indexes before loading data. For the examples in the book, it might be necessary to create the index, remove the data, and reload the data:

```
CREATE INDEX ON Merchandise(…);
TRUNCATE Merchandise;
COPY petstoreweb.merchandise(itemid, description,
qoh, listprice, category) FROM 'Merchandise.csv';
```

Production databases also require periodic use of the nodetool command to repair the database or force updates with the UNIX command line:

```
nodetool repair
```

and indexed columns using equality conditions. It is possible to add additional filtering conditions but only on the other non-key columns which are stored in the same row.

Once more look at the usage plan for the Pet Store Web application. What searches are required in the application that will require indexes? The second issue is the search by Username. When a person logs in, only the Username and Password are provided. The Username has to be unique, so the application needs to retrieve the Customer row with that Username and then verify that the password values match. Because Username is not a primary key, it needs to be indexed:

```
CREATE INDEX idxCustomerUsername ON Customer (Username);
```

The index can be tested by searching for a known Username (BCummings):

```
SELECT CustomerID, Username, Password
FROM Customer
WHERE Username='BCummings';
```

Querying Tables with Compound Keys

The Pet Store Web application needs one more SELECT statement—to retrieve the comments for a given Item. This data is in the ItemComments table which has a compound primary key (ItemID, CustomerID). What command is needed to retrieve this data? Does it need a secondary index? The answer to the second question is "no," which makes the SELECT query straightforward.

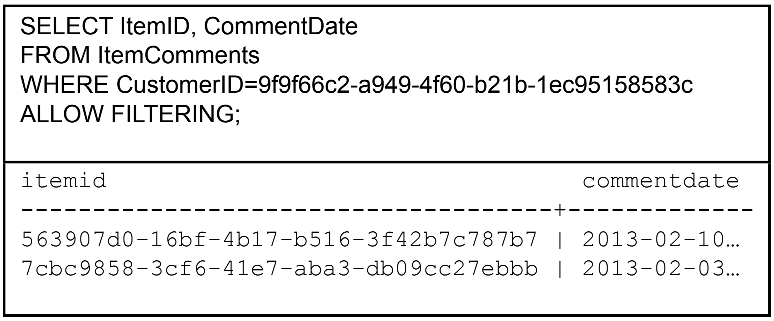
Figure 13.19 shows the Pet Store Web query for retrieving the first 10 comments for a specific ItemID. From the usage diagram, when a user clicks on an item, a page is generated that displays the basic item information (a different query), and then displays some of the comments for that item. The ItemID value is available to the application from the Web click. The designers decided to limit the comments to no more than 10 per page to improve the page performance. The query is relatively simple, which is good. This result is exactly what is needed

Figure 13.19

Queries on compound primary keys. Only the first column in a compound key controls storage so only the ItemID is needed for a search condition. No indexes are necessary and the query will return all rows with the specified key value. A lower LIMIT value is useful for Web pages.

```
SELECT CommentDate, ScreenName, Title, Comment, Rating
FROM ItemComments
WHERE ItemID=7ee762a1-3a27-42a0-a51e-e7988250ecd5
LIMIT 10;
```

```
commentdate   screenname   title      comment                   rating
------------+------------+----------+-------------------------+------
2014-11-14… | Gazer33    | Smells…  | The smell is horrible…  |      4
2014-11-01… | Caged19    | Yummy…   | My human/slave feeds…   |      5
2014-15-21… | Cathouse   | Too big… | OK I only have one cat… |      3
2014-03-07… | RedStar    | Not…     | Not sure it matters…    |      3
```



```
SELECT ItemID, CommentDate
FROM ItemComments
WHERE CustomerID=9f9f66c2-a949-4f60-b21b-1ec95158583c
ALLOW FILTERING;
```

```
itemid                                 commentdate
--------------------------------------+-------------
563907d0-16bf-4b17-b516-3f42b7c787b7 | 2013-02-10…
7cbc9858-3cf6-41e7-aba3-db09cc27ebbb | 2013-02-03…
```

Figure 13.20

Query a compound primary key on the second column. The second (and later) columns in a compound key effectively already have an index and can be retrieved directly with a WHERE statement as long as the ALLOW FILTERING command is used.

for the application, which is why the compound key was chosen in the database design. The key (ItemID, CustomerID) supports a many-to-many relationship that returns all of the customer comments for a given item.

From an application perspective, it might be nice to retrieve the Customer comments sorted by date, but CQL does not support any sorting by clustered columns (as of version 1.2). Instead, the application could read the rows into an array and then sort the data in the code.

What about querying for comments made by a specific customer? As shown in Figure 13.20, because CustomerID is part of the primary key, yes the query will work—but it requires using the ALLOW FILTERING command. What about finding the Item information? CQL does not support any type of JOIN command so the Item data cannot be retrieved with a single query. Instead, the application would have to examine the initial results row-by-row, and then create a new query to retrieve the matching data from the Item table using a single ItemID value at a time.

The point of the example is that the database design was specifically chosen to make the first query easy—not just easy to write but easy and fast to execute. In fact, go back and look at the data storage again in conjunction with the queries used to retrieve the data. The application needed only two secondary indexes to use simple queries to retrieve all of the data. The data was stored in three tables in a distributed system that supports fast write and retrieval. All without using JOIN statements and extra lookups. But, the data tables had to be designed specifically to match the query needs for the application.

INSERT and UPDATE

Cassandra CQL also supports **INSERT** and **UPDATE** commands to add new rows or change the data in an existing row. The syntax for both resembles SQL, as can be seen from two simple examples:

```
INSERT INTO Customer(CustomerID, FirstName, LastName, ScreenName,
Username, Password, Email)
VALUES (469aac21-5600-47c3-882f-f7a1ca269ede, 'Jones', 'Jackie', 'JJJ',
'JJones329', 'password', {'JJones329@gmail.com'} );
```

```
UPDATE Customer
SET Password='password2', ScreenName='JJ3'
WHERE CustomerID=469aac21-5600-47c3-882f-f7a1ca269ede;
```

Note the importance of listing the column names in the INSERT statement. The columns in the primary key are the only required columns, all of the others are optional, so the column names need to be listed to ensure the values are matched correctly to the columns. Observe the braces used in the syntax for the e-mail column because it is defined as a set. Lists use square brackets (['a', 'b']) instead. And mappings require braces and colons such as { 'cost' : '3200', 'name' : 'test15' }.

The UPDATE command changes the column values to the new items. Multiple columns can be set at one time, but the WHERE clause must specify exactly one row. So the UPDATE (and INSERT) command lack the power of the SQL versions. Still, it is convenient to use similar syntax.

A far more interesting twist is what happens if an INSERT command is issued with an ID value that already exists. For instance, assume the two commands have been issued as shown in the short example. Then enter a new command:

```
INSERT INTO Customer(CustomerID, Username, Password)
VALUES (469aac21-5600-47c3-882f-f7a1ca269ede, 'JJones329', 'pass-
word');
```

Note that the CustomerID value exactly matches the one used above. What will be the result of this command? An error message—because of duplication of the IDs? A duplicate row? Try entering the three commands in the order shown, and then issue a SELECT command to examine the values for the specified CustomerID. The answer is that the query processor knows that the CustomerID value already exists, so it effectively converts the INSERT statement into an UPDATE statement. The result is that the Username and Password columns are reset to the values in the last command for the specified CustomerID. Technically, this result means the UPDATE command is not really needed; but what it really means is that you must be careful with any INSERT commands to ensure that the ID values are new (and unique). It is the reason that uuid and timeuuid are important data types—because they ensure that INSERT commands will never overlap an existing ID value.

Cloud Databases

How does cloud computing benefit key-value pair databases? Cloud providers offer a variety of database tools including traditional relational and newer non-relational systems. Some of these DBMSs are available directly from cloud-computing companies, such as DynamoDB from Amazon and App Engine Datastore (bigtable) from Google among others. On the other hand, Cassandra also has an installation script for creating your own cloud using Amazon's EC2 computers. EC2 systems are virtual machines that can be configured quickly and essentially rented by the hour.

A major goal of cloud computing is provide a way to quickly scale an application to handle greater loads—without the need for high upfront fixed costs. Most clouds accomplish this task through distributed virtual servers. With public clouds, a company runs large data centers and installs thousands of servers connected to large-capacity networks. Other companies (you) then configure a virtual machine server and pay hourly (or monthly) rates for using the virtual machine.

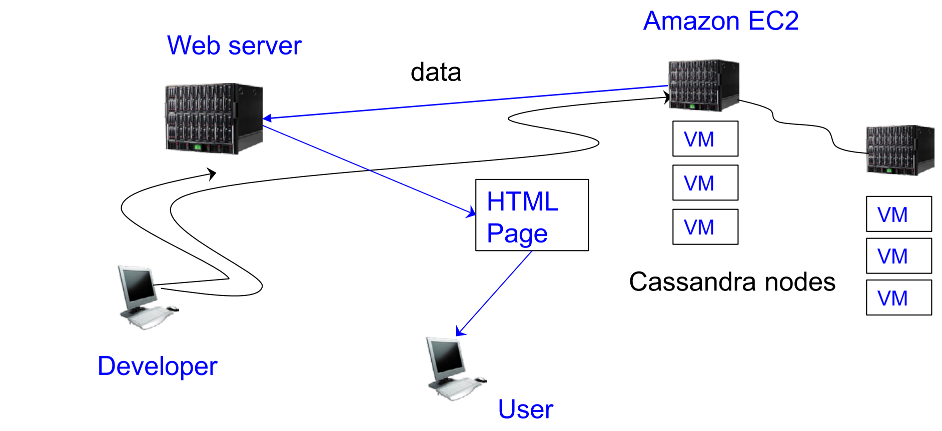


Figure 13.21

Cassandra on Amazon EC2. EC2 has multiple servers in multiple data centers around the world where virtual machines can be rented by the hour. Each VM becomes a node on a Cassandra network. The Web application just writes data to the Cassandra keyspace application and it is distributed across the Amazon network. New nodes can be added in minutes to expand capacity with almost no fixed costs.

If security is critical and cannot be handled through encryption, a company might choose to build its own data centers, but the distributed concepts and virtual machine configurations are largely the same. In both cases, you will be responsible for configuring and running the server and its applications.

DataStax provides a special copy of Cassandra and instructions for installing nodes on an Amazon EC2 cluster. As Figure 13.21 shows, with these tools, it is straightforward to build as many nodes as necessary on Amazon's system. Because Cassandra is designed to be distributed, it runs well on the distributed servers. Detailed configuration options provide control over replication within and across data centers to meet a variety of different Web needs. And more nodes can quickly be added as the number of users increases. Once the base system is configured, it is directly accessible to a Web application running on any server.

In the case of huge applications with millions of global users, the Web application can be written to connect to the closest geographical data center. Cassandra eventually replicates the data so it is available everywhere, even if a few nodes are inaccessible. Yet, most of the data is provided locally to the user, reducing the need to transmit data immediately around the world; which improves performance of the applications.

If you do not want to install and run your own copies of the DBMS, many tools (such as DynamoDB and App Engine Datastore) are available for hourly or monthly lease charges. In these cases, you simply define the tables and columns needed and tell the application to use the cloud databases instead of your own copy. Data distribution and backups are handled by the cloud provider.

The difference between the options largely comes down to cost. Running your own data centers involves a substantial upfront fixed cost, along with expertise and people to manage the centers on a day-to-day basis. Using virtual servers and configuring your own databases through companies such as Amazon and Rackspace eliminates the fixed cost of installing the physical servers and networks. But, the monthly operating costs are higher than if you ran the same capacity on your own machines. The third option of using a prebuilt database cloud (DynamoDB or Google bigtable) has even higher monthly costs, but requires less expertise to configure and manage.

Many times it is difficult to predict the exact level of capacity needed for each month. Cloud-based systems have slightly higher marginal costs, but remove the need to guess ahead of time on the necessary capacity, because they are easy to expand or contract when needed. They also provide professional-level management and bandwidth to even tiny firms. Small firms can get the same high-level of distributed systems without having to pay huge upfront costs, just by purchasing capacity on the public cloud systems. If customer demand increases, presumably the revenues will also increase to cover the higher costs.

Summary

Web-based applications are important to many organizations. Some Web applications are like any other business application, but a few are radically different. Applications built to be used by millions of customers to store data online, such as the social-based sites, have different data needs than a standard business application. Performance and continuous accessibility are critical to these applications. Transactions and perfect data consistency are less important. Cost is another critical factor, particularly when the social features are offered at low or minimal cost.

Highly-distributed databases are an important tool to address these new applications. These systems can use hundreds or thousands of servers in data centers around the world to provide the data backbone for the application. To hold down the licensing costs, several new non-relational database systems were created to provide these features, by using open-source servers and software. Cassandra is a leading tool in many organizations, including Facebook. These new systems focus on storing data in key-value pairs; and replicate data automatically across multiple server nodes.

Database design is important to non-relational systems, but the rules are more flexible. It is useful to start with normalized tables, and then adjust the design to optimize performance for the specific application. The key point is that the design must match the individual needs of the application, so it is important to lay out the usage (use cases in OO terms) and identify all data retrieval needs.

Tables contain primary keys but the keys typically use uuid values which are safer to generate in a distributed system. Initially, rows can be retrieved from a table only through the primary key values; and those values must only use equality conditions. Queries on additional columns can be supported by adding a secondary index on that column, but those queries can also only use equality conditions. Adding too many indexes will slow down processing of new data, so the design has to be conservative. Table JOINs are not supported, so data is often duplicated by including base information in transaction rows. For instance, a Sale table would likely include Customer name and shipping address so that information can be retrieved automatically with each Sale instead of requiring an additional lookup.

Many-to-many relationships are indicated by specifying both columns in the primary key; just as in the relational model. However, the difference between compound and composite keys is critical. Compound keys are written as (ItemID, CustomerID); while composite keys include an extra set of parentheses: ((ItemID, CustomerID)). Compound keys store data using only the first column, while composite keys use both columns to partition the data. Consequently, data stored with compound keys require only a value for the first column (ItemID) to retrieve a row; while composite keys require both values. A compound key actually returns multiple "rows" but a composite key returns exactly one row that matches all of the ID values. Typical applications will likely use compound keys instead of composite keys.

Queries in CQL use a simplified SELECT command. But the command is even more limited than it appears. JOINs are not supported and WHERE conditions are almost always based only on equality constraints. Additional constraints often require the ALLOW FILTERING clause to indicate they might be slower queries. Data results are always limited to a specified number of rows. The default is 10,000 rows, but the value can be changed with the LIMIT clause.

The DMBSs, including Cassandra, are constantly being revised and updated, so limitations are likely to change; but the main limitations of the query system are due to the data storage model and the emphasis on performance. If an application needs more complex queries, it would probably be easier to move it to a relational DBMS or perhaps a data warehouse. The purpose of key-value pair non-relational DBMSs is to provide a fast, reliable way to store and retrieve individual pieces of data to millions of users.

A Developer's View

As more applications move to the Web, performance and continuous availability can become critical. Distributed databases can be significantly more responsive in this environment, but installing thousands of servers and copies of the DBMS software can be expensive. Open-source non-relational systems, such as Cassandra, are designed to handle these issues. Data is stored and retrieved as key-value pairs, which is fast for retrieving specific pieces of data. The tools do not support JOINs, referential integrity, or complex queries; so they are less useful for complex, interrelated business data. But there are times when you need a different tool to handle performance issues, and a DBMS like Cassandra can solve problems that are difficult and expensive to handle with a relational DBMS. However, the performance gains also arise through careful database design adjusted specifically for each application.

Key Terms

ad hoc queries
ALLOW FILTERING
Cassandar Query Language (CQL)
column family
columns
composite primary key
compound primary key
gossip
INSERT
keyspace
key-value pair
LIMIT
list
map
non-relational database
NoSQL
peer-to-peer
primary key
set
solid state drives (SSDs)
sparse table
tunable consistency
universally unique identifier (uuid)
UPDATE

Review Questions

1. How is a table in a key-value pair database different from one in a relational database?
2. How do highly-distributed databases create problems with data consistency?
3. What are the benefits and drawbacks to changing table columns over time?
4. What consistency problems can arise in a key-value pair database like Cassandra?
5. A database contains tables for Employee, Factory, and Assembly, where the Assembly table records which parts were installed by each employee at a specified time in each factory. Why would some of the employee data be stored in the Assembly rows?
6. What programming language is Cassandra written in and why does it matter?
7. What are the benefits and drawbacks to using uuids as primary key values?
8. Briefly explain what queries are supported by a table with a compound key of two columns, without adding indexes.
9. Why does Cassandra require indexes for some queries?
10. How is a composite key different from a compound key?

Exercises

1. Explain why you should avoid storing totals (such as inventory quantity on hand) in a key-value pair database and briefly describe an alternative to avoid totals.
2. An online site wants to hold medical health records for patients in a specific program. The records include basic patient data (name, birthdate, gender), and medical test results at various points in time that include levels for standard items such as Glucose, Potassium, and blood pressure. Design the tables you would use to store the data in Cassandra. Highlight the main queries.
3. Define the key-value-pair tables that would be needed for a Web site that sells custom shirts. Customers choose colors and sizes, and can enter text to be printed on the back, front, or sleeves.
4. Using Exercise 2 in Chapter 3 as a guide, define the key-value-pair tables needed for a Web site that lets individuals track their weight-lifting progress. Each session has multiple exercises (equipment), with different sets of weight levels, and a need to record the number of repetitions. For instance, a bench press might involve set1: 135 pounds, 10 reps, 185 pounds, 10 reps; and so on.
5. Looking at the previous exercise on weight lifting, explain how to change the design to support an application that shows the maximum weight lifted in an exercise over time (session). For example, the highest weight lifted in the bench press over the last year.
6. A Web site is built to access a database of music/songs (not classical music which has unique data elements). Users have the ability to rate each individual song. Define the key-value-pair tables needed for this site. Identify common queries that will be used and any indexes needed to support those queries.
7. Define the tables needed by Cassandra to build a Web site for a basketball league that records points scored by each team, the name of the referee, and the names of the players on each team. Similar to Chapter 3, Exercise 3.
8. Research Cassandra and briefly explain the difference between the timestamp and timeuuid data types.
9. Find a programming tool (not an online Web service) and generate 5 uuid values.
10. Find a different key-value-pair (NoSQL) DBMS and briefly compare it to Cassandra.

Sally's Pet Store

Using the Pet Store Web sample database for Cassandra, write the queries to answer the following questions.

11. Get the Description, ListPrice, and QOH for Item 5e9c2e10-3db1-4189-8c0d-0c700d421f17.

12. List all items in the Fish category? What indexes are needed?
13. ✓ List all items in the Dog category with a list price above $20 and a QOH greater than 50. What indexes are needed?
14. Write a query that enables the system to e-mail the username/password to users who cannot remember what they entered, but do know their e-mail address and name. What indexes are needed?
15. List all of the comments with a rating of less than 3. What indexes are needed?
16. If the application needs to display only the first 10 (earliest) comments fir a specific item (on the Web page), how should the design be modified and what is the new query?
17. How many comments have been made by the user with the screen name of Caged19? What indexes are needed?
18. ✓ Run the three UPDATE and INSERT queries in the text and issue a SELECT statement to see the ending values for the new CustomerID;
19. Change the list price on item ed7bb389-152c-4bdb-8546-4cd070fb4ae9 to $10.
20. Add a new comment to the ItemComments table.

Rolling Thunder Bicycles

21. Rolling Thunder managers want to create a Web site to let owners upload photos of their custom bikes and let other users submit comments or questions; which can then be answered by other users (basically a discussion list). Define the tables needed for this application.
22. ✓ Assuming standard Web conventions, such as login by Username, what initial indexes will be needed for the discussion Web site in the previous question?
23. If Rolling Thunder managers decide to build the entire main ordering system as a Web site, would it be better to use a traditional relational DBMS or a key-value pair system like Cassandra? Explain your answer.

Corner Med

24. The managers of Corner Med want a Web site that handles communication between physicians and patients. They do not want to put all the patient visit data online, but want a secure site that enables patients to ask questions that are answered by physicians. What Cassandra tables would be needed for this site?
25. Assuming the solution to the previous question includes at least a Person and Discussion table, write the query to list all of the questions posted from a specific patient. What indexes would be needed?

Web Site References

http://cassandra.apache.org	Open-soure location of code.
http://www.datastax.com/docs	DataStax documentation site.
http://nosql-database.org/	List of NoSQL database projects.
http://planetcassandra.org/	Cassandra background (DataStax)
http://aws.amazon.com/nosql/	Amazon NoSQL options.
https://developers.google.com/appengine/	Google App Engine.

Additional Reading

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem. Interesting, readable comments on distributed transactions, scalability, and consistency as a reason for non-relational databases. (Brewer's CAP Theorem and references to other articles.)

Glossary

24 -7 Operation of an application or database 24 hours a day, 7 days a week. Because the database can never be shut down, performing maintenance is a challenge.

Abstract data types In SQL 1999 the ability to define more complex data domains that support inheritance for storing objects.

Accessibility A design goal to make the application usable by as many users as possible, including those with physical challenges. One solution is to support multiple input and output methods.

ACID transactions The acronym for transactions that specifies the four required elements of a safe transaction: atomicity, consistency, isolation, and durability.

Active data objects (ADO) Microsoft's component (COM) approach to connect program code and Web server scripts to a database. Provides SQL statement and row-level access to virtually any database.

Active server pages (ASP) Microsoft's Web pages that enable you to run script programs on the server. Useful for providing access to a server database for Internet users.

Address, physical The location of data stored in memory or on a file system. Used to establish a pointer to a specific piece of data.

ad hoc query An unplanned question asked by users or administrators. Because it is unplanned, it is hard to support with non-relational databases.

Administrative tasks Jobs that need to be performed to keep the application running, such as updating data in lookup tables, backing up the database, and assigning users to groups

Advanced Encryption Standard (AES) A single-key encryption system to replace DES, based on a Belgian encryption system: Rijndael. It supports key lengths of 128, 192 and 256 bits, making it considerably more secure than DES.

Aesthetics An application design goal, where layout, colors, and artwork are used to improve the appearance of the application—not detract from it. By its nature, the value of any design is subjective.

Aggregation The generic name for several SQL functions that operate across the selected rows. Common examples include SUM, COUNT, and AVERAGE.

Aggregation association A relationship where individual items become elements in a new class. For example, an Order contains Items. In UML, the association is indicated with a small open diamond on the association end. *See composition.*

Alias A temporary name for a table or a column. Often used when you need to refer to the same table more than once, as in a self-join.

ALL A SQL SELECT clause often used with subqueries. Used in a WHERE clause to match all of the items in a list. For example, Price > ALL (…) means that the row matches only if Price is greater than the largest value in the list.

ALLOW FILTERING Cassandra query option required for supporting most WHERE clause options including AND, OR, and some range questions. Might reduce query performance so it is a warning that complex queries should be avoided.

ALTER TABLE A SQL data definition command that changes the structure of a table. To improve performance, some systems limit the changes to adding new columns. In these situations to make major changes, you have to create a new table and copy the old data.

Anchor tag The HTML tag that signifies a link. Denoted with <A>.

ANY A SQL SELECT clause often used with subqueries. Used in a WHERE clause to match at least one of the items in a list. For example, Price > ANY (…) means that the row matches as long as Price is greater than at least one item in the list.

Application A complete system that performs a specific collection of tasks. It typically consists of integrated forms and reports and generally contains menus and a Help system.

Application Design Guide A standard set of design principles that should be followed when building applications. The standard makes it easier for users to operate new applications, since techniques they learn in one system will work in another.

Application generator A DBMS tool that assists the developer in creating a complete application package. Common tools include menu and toolbar generators and an integrated context-sensitive Help system

Association Connections between classes or entities. Generally, they represent business rules. For example, an order can be placed by one customer. It is important to identify whether the association is one-to-one, one-to-many, or many-to-many.

Association role In UML the point where an association attaches to a class. It can be named, and generally shows multiplicity, aggregation, or composition.

Association rules A data mining technique that examines a set of transactions to see which items are commonly purchased together.

Atomic The smallest single-valued form of a data element. A table cannot be in first normal form if the cells contain non-atomic data. The definition is subjective depending on the application. For instance, an address such as 123 Main Street is usually considered to be atomic even though it refers to both a house number and a street.

Atomicity The transaction element that specifies that all changes in a transaction must succeed or fail together.

Attribute A feature or characteristic of an entity. An attribute will become a column in a data table. Employee attributes might include name, address, date hired, and phone.

Authentication Providing a verification system to determine who actually wrote a message. Common systems use a dual-key encryption system.

Autonumber A type of data domain where the DBMS automatically assigns a unique identification number for each new row of data. Useful for generating primary keys.

B+tree An indexed data storage method that is efficient for a wide range of data access tasks. Tree searches provide a consistent level of performance that is not affected by the size of the database.

Back end In a client-server system, the back end usually consists of a central database. In general, hardware and data placed at the back end is designed to be centralized and shared. *See front end.*

Base table A table that contains data about a single basic entity. It generally contains no foreign keys, so data can be entered into this table without reference to other tables. For example, Customer would be a base table; Order would not.

Behavioral security Emphasizes the role of people or employees in security. It is related to logical security but involves interesting problems because it deals with mistakes that people make, such as tricking people to reveal passwords.

BETWEEN A SQL comparison operator that determines whether an item falls between two values. Often useful for dates.

Binary large object (BLOB) A data domain for undefined, large chunks of data. A BLOB (or simple object) type can hold any type of data, but the programmer is often responsible for displaying, manipulating, and searching the data.

Binary search A search technique for sorted data. Start at the middle of the data. If the search value is greater than the middle value, split the following data in half. Keep reducing by half until the value is found.

Bitmap index A compact, high-speed indexing method where the key values and conditions are compressed to a small size that can be stored and searched rapidly.

Boolean algebra Creating and manipulating logic queries connected with AND, OR, and NOT conditions.

Bound control A control on a form that is tied to a column in the database. When data is entered or changed, the changes are automatically saved to the data table.

Boyce-Codd normal form (BCNF) All dependencies must be explicitly shown through keys. There cannot be a hidden dependency between nonkey and key columns.

Browser, Web A software package on a client personal computer used to access and display Web pages from the Internet.

Brute force attack An attempt to break a security system by trying every possible combination of passwords or encryption keys.

Business intelligence (BI) The general process of analyzing data to find patterns. Tools include automated statistical systems and user-driven exploratory options. Also called data mining.

Business rules The conditions and assumptions that describe how an organization operates. One-to-one and one-to-many rules are particularly important. For instance, a common business rule is that a sale is placed by only one customer.

Call-level interface (CLI) A set of libraries that enable programmers to work in a language outside the DBMS (e.g., C++) and utilize the features of the DBMS. The DBMS provides the communication libraries and handles much of the data exchange itself.

Cascading delete When tables are linked by data, if you delete a row in a higher level table, matching rows in other tables are deleted automatically. For example, if you delete Customer 1173, all orders placed by that customer are also deleted.

Cascading style sheets (CSS) A style sheet that defines how elements are to be displayed on a Web page. Cascading means that a style can be overridden by declaring it inside the HTML file, but to maintain consistency, styles should be defined only in the style sheet.

Cascading triggers Multiple events that arise when a change that fires a trigger on one table causes a change in a second table, that triggers a change in a third table and so on.

CASE A SQL operator supported by some systems. It examines multiple conditions (cases) and takes the appropriate action when it finds a match.

Casandra Query Language (CQL) An interactive query system for retrieving data from a Cassandra database. It includes commands similar in syntax to SQL but with limited options. In particular the SELECT command does not support JOINs.

Certificate authority A company that ensures the validity of public keys and the applicant's identify for dual-key encryption systems.

Check box A square button that signifies a choice. By the design guide, users can select multiple options with check boxes, as opposed to option buttons that signify mutually exclusive choices.

Clarity The goal of making an application easier to use through elegant design and organization that matches user tasks so that the purpose and use of the application is clear to the user.

Class A descriptor for a set of objects with similar structure, behavior, and relationships. That is, a class is the model description of the business entity. A business model might have an Employee class, where one specific employee is an object in that class.

Class diagram A graph of classes connected through relationships. It is designed to show the static structure of the model. Similar to the entity-relationship diagram.

Class hierarchy A graph that highlights the inheritance relationships between classes.

Classification analysis A data mining technique that classifies groups of objects, such as customers. It determines which factors are important classifier variables.

Client/server A technique for organizing systems where a few computers hold most of the data, which is retrieved by individuals using personal computer clients.

Cloud computing Using servers that are connected to the Internet. Typically the servers are leased as virtual machines that can be expanded as needed. Users rely on Web-based clients to run applications and retrieve data.

Cluster, data A physical data storage technique to improve performance by storing related data in the same data blocks so that the operating system retrieves the related data in one pass.

Cluster, server A collection of computer servers that share the workload. If one machine fails, the others pick up the load. New servers can be added at any time to improve performance.

Cluster analysis A data mining technique that groups elements of a dataset, often based on how close the items are to each other.

Cold site A facility that can be leased from a disaster backup specialist. A cold site contains power and telecommunication lines, but no computer. In the event of a disaster, a company calls the computer vendor and begs for the first available machine to be sent to the cold site.

Collaboration diagram A UML diagram to display interactions among objects. It does not show time as a separate dimension. It is used to model processes.

Column A column in a table represents an attribute or measure on the object defined by the table. Columns have a defined data type. A simple example would be a Phone Number in a Customer table. In a relational DBMS, columns hold a single value for each row. In a non-relational DBMS, columns may contain repeating or complex data.

Column family The early term for a table in the Cassandra non-relational DBMS. The term still arises in some documentation and error messages.

Combo box A combination of a list box and a text box that is used to enter new data or to select from a list of items. A combo box saves space compared to a list box since the list is displayed only when selected by the user. Known as a select box on Web forms.

Comma separated values (CSV) A method of storing data for transfer to different computers or applications. Tabular data is stored in rows with the columns separated by commas. The data is stored in a simple text file.

Command button A button on a form that is designed to be clicked. The designer writes the code that is activated when the button is clicked.

Common gateway interface (CGI) With Web servers, CGI is a predefined system for transferring data across the Internet. Current scripting languages hide the details, so you can simply retrieve data as it is needed.

Common language runtime (CLR) Microsoft's base programming language. It is embedded into versions of SQL Server from 2005, so database procedures can be written in CLR langugages such as Visual Basic and C# including access to all of their functions. Needed to use RegEx within SQL Server. Note that CLR support is turned off by default.

Composite key A primary key that consists of more than one column. Indicates a many-to-many relationship between the columns.

Composite primary key In non-relational DBMSs, particularly Cassandra, a composite key uses two or more columns to define the partitioning key so both values are required to retrieve the associated data. It is defined with an extra set of parentheses: PRIMARY KEY ((a, b), others).

Composition association A relationship in which an object is composed of a collection of other objects. For example, a bicycle is built from components. In UML, it is indicated with a small filled diamond on the association end.

Compound primary key In non-relational DBMSs, particularly Cassandra, a compound key defines a many-to-many relationship across multiple columns. But unlike the composite key, it uses only the first column to partition the data so all elements of the many side are retrieved by specifying only the value for the first key column. The definition does not include extra parentheses: PRIMARY KEY (a, b, c, …).

Computer-aided software engineering (CASE) Computer programs that are designed to support the analysis and development of computer systems. They make it easier to create, store, and share diagrams and data definitions. Some versions can analyze existing code and generate new code.

Concatenate A programming operation that appends one string on the end of a second string. For example, LastName & ", " & FirstName could yield "Smith, John".

Concatenated key *See composite key.*

Concurrent access Performing two (or more) operations on the same data at the same time. The DBMS must sequence the operations so that some of the changes are not lost.

Confidence In data mining with association rules, a measure of the strength of a rule measured by the percentage of transactions with item A that also contain item B. The probability that B is in the basket given that A is already there.

Consistency, application The goal of making an application easier to use by using the same features, colors, and commands throughout. Modern applications also strive for consistency with a common design guide.

Consistency, transaction The transaction requirement that specifies all data must remain internally consistent when changes are committed and can be validated by application checks.

Constraint In SQL, a constraint is a rule that is enforced on the data. For example, there can be primary-key and foreign-key constraints that limit the data that can be entered into the declared columns. Other business rules can form constraints, such as Price > 0.

Context-sensitive help Help messages that are tailored to the specific task the user is performing.

Context sensitive menu A menu that changes depending on the object selected by the user.

Control break A report consisting of grouped data uses control breaks to separate the groups. The break is defined on the key variable that identifies each member of the group.

Controls The generic term for an item placed on a form. Typical controls consist of text boxes, combo boxes, and labels.

Correlated subquery A subquery that must be reevaluated for each row of the main query. Can be slow on some systems. Can often be avoided by creating a temporary table and using that in the subquery instead.

CREATE DOMAIN A SQL data definition command to create a new data domain that is composed of existing domain types.

CREATE SCHEMA A SQL data definition command to create a new logical grouping of tables. With some systems it is equivalent to creating a new database. This command is not available in Oracle or SQL Server.

CREATE TABLE A SQL data definition command to create a new table. The command is often generated with a program.

CREATE VIEW A SQL command to create a new view or saved query.

Cross join Arises when you do not specify a join condition for two tables. It matches every row in the first table with every row in the second table. Also known as the Cartesian product. It should be avoided.

Crosstab A special SQL query (not offered by all systems) that creates a tabular output based on two groups of data. Access uses a TRANSFORM command to create a cross tabulation.

Cursor, graphics The current location pointer in a graphical environment.

Cursor, database A row pointer that tracks through a table, making one row of data active at a time.

Cylinder Disk drives are partitioned into cylinders (or sectors) that represent a portion of a track.

Data administration Planning and coordination required to define data consistently throughout the company.

Data administrator (DA) The person in charge of the data resources of a company. The DA is responsible for data integrity, consistency, and integration.

Data bound A control that contains a link to a table and column data source. When a data-bound control is displayed, the data is retrieved from the database and shown in the control. Changes are written to the defined column and table.

Data definition language (DDL) A set of commands that are used to define data, such as CREATE TABLE. Graphical interfaces are often easier to use, but the data definition commands are useful for creating new tables with a program.

Data device Storage space allocated to hold database tables, indexes, and rollback data. *See tablespace.*

Data dictionary Holds the definitions of all of the data tables and describes the type of data that is being stored.

Data hierarchy A structured ordering of data where higher levels contain aggregates of the lower levels. Some common hierarchies include dates (year, quarter, month, day), and geography (country, state, city).

Data independence Separates the data from the programs, which often enables the data definition to be changed without altering the program.

Data integrity Keeping accurate data, which means few errors and means that the data reflects the true state of the business. A DBMS enables you to specify constraints or rules that help maintain integrity, such as prices must always be greater than 0.

Data manipulation language (DML) A set of commands used to alter the data. *See INSERT, DELETE, and UPDATE.*

Data mining Searching databases for unknown patterns and information. Tools include statistical analysis, pattern-matching techniques, and data segmentation analysis, classification analysis, association rules, and cluster analysis.

Data normalization The process of creating a well-behaved set of tables to efficiently store data, minimize redundancy, and ensure data integrity. *See first, second, and third normal form.*

Data replication In a distributed system, placing duplicate copies of data on several servers to reduce overall transmission time and costs.

Data repository A complete listing of all terms used in a database design, including column names and tables. *See data dictionary.*

Data type A type of data that can be held by a column. Each DBMS has predefined system domains (integer, float, string, etc.). Some systems support user-defined domains that are named combinations of other data types.

Data volume The estimated size of the database. Computed for each table by multiplying the estimated number of rows times the average data length of each row.

Data warehouse A specialized database that is optimized for management queries. Data is extracted from online transaction processing systems. The data is cleaned and optimized for searching and analysis. Generally supported by parallel processing and RAID storage.

Database A collection of data stored in a standardized format, designed to be shared by multiple users. A collection of tables for a particular business situation.

Database administration The technical aspects of creating and running the database. The basic tasks are performance monitoring, backup and recovery, and assigning and controlling security.

Database administrator (DBA) A specialist who is trained in the administration of a particular DBMS. DBAs are trained in the details of installing, configuring, and operating the DBMS.

Database cursor A variable created within a programming language that defines a SELECT statement and points to one row of data at a time. Data on that row can be retrieved or edited using the programming language.

Database engine The heart of the DBMS. It is responsible for storing, retrieving, and updating the data.

Database management system (DBMS) Software that defines a database, stores the data, supports a query language, produces reports, and creates data entry screens.

Datasheet A gridlike form that displays rows and columns of data. Generally used as a subform, a datasheet displays data in the least amount of space possible.

Deadlock A situation that exists when two (or more) processes each have a lock on a piece of data that the other one needs.

Default values Values that are displayed and entered automatically. Used to save time at data entry.

Degree, tree The maximum number of children allowed beneath one node in a B-tree. Most systems choose an odd number greater than or equal to three.

DELETE A SQL data manipulation command that deletes rows of data. It is always used with a WHERE clause to specify which rows should be deleted.

Deletion anomaly Problems that arise when you delete data from a table that is not in third normal form. For example, if all customer data is stored with each order, when you delete an order, you could lose all associated customer data.

DeMorgan's law An algebraic law that states: To negate a condition that contains an AND or an OR connector, you negate each of the two clauses and switch the connector. An AND becomes an OR and vice versa.

DENSE_RANK A data mining extension to SQL that rank orders data. If ties exist, they receive the same dense rank, but the next value receives a dense rank that is one unit higher. Compare to RANK.

Dependence An issue in data normalization. An attribute A depends on another attribute B if the values of A change in response to changes in B. For example, a customer's name depends on the CustomerID (each employee has a specific name). On the other hand, a customer's name does not depend on the OrderID. Customers do not change their names each time they place an order.

Depth, tree The number of levels in a B-tree or the number of nodes between the root and the leaves.

Derived class A class that is created as an extension of another class. The programmer need only define the new attributes and methods. All others are inherited from the higher-level classes. *See inheritance.*

DESC The modifier in the SQL SELECT … ORDER BY statement that specifies a descending sort (e.g., Z … A). ASC can be used for ascending, but it is the default, so it is not necessary.

Dimension An attribute in an OLAP cube that is used to group and search the data.

Direct access A data storage method where the physical location is computed from the logical key value. Data can be stored and retrieved with no searches.

Direct manipulation of objects A graphical interface method that is designed to mimic real-world actions. For example, you can copy files by dragging an icon from one location to another.

Disaster plan A contingency plan that is created and followed if a disaster strikes the computer system. Plans include off-site storage of backups, notifying personnel, and establishing operations at a safe site.

DISTINCT An SQL keyword used in the SELECT statement to remove duplicate rows from the output.

Distributed database Multiple independent databases that operate on two or more computers that are connected and share data over a network. The databases are usually in different physical locations. Each database is controlled by an independent DBMS.

Dockable toolbar A toolbar that users can drag to any location on the application window. It is generally customized with options and buttons to perform specific tasks.

Domain-key normal form (DKNF) The ultimate goal in designing a database. Each table represents one topic, and all of the business rules are expressed in terms of domain constraints and key relationships. That is, all of the business rules are explicitly described by the table rules.

Drag-and-drop A graphical interface technique where actions are defined by holding down a mouse key, dragging an icon, and dropping the icon on a new object.

Drill down The act of moving from a display of summary data to more detail. Commonly used in examining data in a data warehouse or OLAP application. *See Roll up.*

Drive head The mechanism that reads and writes data onto a disk. Modern drives have several drive heads.

DROP TABLE A SQL data definition command that completely removes a table from the database—including the definition. Use it sparingly.

Dual-key encryption An encryption technique that uses two different keys: one private and one public. The public key is published so anyone can retrieve it. To send an encrypted message to someone, you use the person's public key. At that point, only the person's private key will decrypt the message. Encrypting a message first with your private key can also be used to verify that you wrote the message.

Durability The transaction element that specifies that when a transaction is committed, all changes are permanently saved even if there is a hardware or system failure.

Edit The Microsoft DAO command to alter data on the current row.

Electronic data interchange (EDI) Exchanging data over networks with external agents such as suppliers, customers, and banks.

Encapsulation In object-oriented programming, the technique of defining attributes and methods within a common class. For example, all features and capabilities of an Employee class would be located together. Other code objects can use the properties and methods but only by referencing the Employee object.

Encryption Encoding data with a key value so the data becomes unreadable. Two general types of encryption are used today: single key (e.g., DES) and dual key (e.g., RSA).

Entity An item in the real world that we wish to identify and track.

Entity-relationship diagram (ERD) A graph that shows the associations (relationships) between business entities. Under UML, the class diagram displays similar relationships.

Equi-join A SQL equality join condition. Rows from two tables are joined if the columns match exactly. Equi-join is the most common join condition. Theta joins support inequality conditions.

Error handling Special programming code used to trap errors. The try/catch or On Error Goto syntaxes are common. The goal is to catch errors and handle them automatically without interfering with the user.

Event Something that arises during database or form operations. Events are named and developers can write code that is executed when a specific event is triggered.

EXCEPT A SQL operator that examines rows from two SELECT statements. It returns all rows from one statement except those that would be returned by the second statement. Sometimes implemented as a SUBTRACT command. *See UNION.*

EXISTS A SQL keyword used to determine if subqueries return any rows of data.

Expert system (ES) A system with a knowledge base consisting of data and rules that enables a novice to make decisions as effectively as an expert.

Extensible markup language (XML) A tag-based notation system that is used to assign names and structure to data. It was mainly designed for transferring data among diverse systems.

Extraction, transformation, and transportation (ETT) The three steps in populating a data warehouse from existing files or databases. Extraction means selecting the data you want. Transformation is generally the most difficult step and requires making the data consistent. Transportation implies that the data has to be physically moved over a network to the data warehouse. Oracle refers to the topic as ETL (loading).

Fact table The table or query holding the facts to be presented in an OLAP cube.

Fault tolerance Various methods of building a system so that if something fails, other components pick up the load without losing the entire transaction or application.

Feasibility study A quick examination of the problems, goals, and expected costs of a proposed system. The objective is to determine whether the problem can reasonably be solved with a computer system.

Feedback A design feature where the application provides information to the user as tasks are accomplished or errors arise. Feedback can be provided in many forms (e.g., messages, visual cues, or audible reminders).

FETCH The command used in SQL cursor programming to retrieve the next row of data into memory.

First normal form (1NF) A table is in 1NF when there are no repeating groups within it. Each cell can contain only one value. For example, how may items can be placed in one Order table? The items repeat, so they must be split into a separate table.

Fixed-width storage Storing each row of data in a fixed number of bytes per column.

Fixed-with-overflow storage Storing a portion of the row data in a limited number of bytes, and moving extra data to an overflow location.

Focus In a window environment, a form or control has focus when it is the one that will receive keystrokes. It is usually highlighted.

For Each … Next In VBA, an iteration command to automatically identify objects in a group and apply some operation to that collection. Particularly useful when dealing with cells in a spreadsheet.

Foreign key A column in one table that is a primary key in a second table. It does not need to be a key in the first table. For example, in an Order table, CustomerID is a foreign key because it is a primary key in the Customer table.

Forms development The process of designing and creating input forms to collect data and store it in the database.

Forms generator A DBMS tool that enables you to set up input forms on the screen.

Fourth normal form (4NF) There cannot be hidden dependencies between key columns. A multi-valued dependency exists when a key determines two separate but independent attributes. Split the table to make the two dependencies explicit.

FROM The SQL SELECT clause that signifies the tables from which the query will retrieve data. Used in conjunction with the JOIN or INNER JOIN statement.

Front end In a client-server or multi-tier system, the forms and applications that are displayed or run on the user's computer. The portion of the application seen and manipulated by the user. *See back end.*

FULL JOIN A join that matches all rows from both tables if they match, plus all rows from the left table that do not match, and all rows from the right table that do not match. Rarely used and rarely available. *See left join and right join.*

Function A procedure designed to perform a specific computation. The difference between a function and a subroutine is that a function returns a specific value (not including the parameters).

Generalization association A relationship among classes that begins with a generic class. More detailed classes are derived from it and inherit the properties and methods of the higher level classes.

Geocode Assigning location coordinates of latitude and longitude to a dataset.

Geographic information system (GIS) Designed to identify and display relationships among business data and locations. A good example of the use of objects in a database environment.

Globally-unique identifier (GUID) A large number that can be created on one computer and be different from all other numbers created. Often used for generated keys in a replicated database.

Gossip A small communication channel in the Cassandra DBMS used to monitor server node status and metadata across database replicas.

GRANT The SQL command to give someone access to specific tables or queries.

Graphics interchange file (GIF) One standard method of storing graphical images. Commonly used for images shared on the Internet.

Group break A report that splits data into groups. The split-point is called a break. Also known as a control break.

GROUP BY A SQL SELECT clause that computes an aggregate value for each item in a group. For example, SELECT Department, SUM(Salary) FROM Employee GROUP BY Department; computes and lists the total employee salaries for each department.

Hashed-key access *See direct access.* Hash refers to the function used to reduce a key value to a numbered location—usually modulo division by a prime number.

HAVING A SQL clause used with the GROUP BY statement. It restricts the output to only those groups that meet the specified condition.

Heads-down data entry Touch typists concentrate on entering data without looking at the screen. Forms for this task should minimize keystrokes and use audio cues.

Help system A method for displaying, sequencing, and searching help documentation. Developers need to write the help files in a specific format and then use a help compiler to generate the final help file.

Hidden dependency A dependency specified by business rules that is not shown in the table structure. It generally indicates that the table needs to be normalized further and is an issue with Boyce-Codd or fourth normal form.

Hierarchical database An older DBMS type that organizes data in hierarchies that can be rapidly searched from top to bottom, e.g., Customer – Order – OrderItem.

Horizontal partition Splitting a table into groups based on the rows of data. Rows that are seldom used can be moved to slower, cheaper storage devices.

Hot site A facility that can be leased from a disaster backup specialist. A hot site contains all the power, telecommunication facilities, and computers necessary to run a company. In the event of a disaster, a company collects its backup data, notifies workers, and moves operations to the hot site.

Human factors design An attempt to design computer systems that best accommodate human users.

Hypertext link Hypertext (e.g., Web) documents consist of text and graphics with links that retrieve new pages. Clicking on a link is the primary means of navigation and obtaining more information.

Hypertext markup language (HTML) A display standard that is used to create documents to be shared on the Internet. Several generators will create HTML documents from standard word processor files.

Icon A small graphical representation of some idea or object. Typically used in a graphical user interface to execute commands and manipulate underlying objects.

IN A SQL WHERE clause operator typically used with subqueries. It returns a match if the selected item matches one of the items in the list. For example, WHERE ItemID IN (115, 235, 536) returns a match for any of the items specified. Typically, another SELECT statement is inserted in the parentheses.

Index A sorted list of key values from the original table along with a pointer to the rest of the data in each row. Used to speed up searches and data retrieval.

Indexed sequential access method (ISAM) A data storage method that relies on an index to search and retrieve data faster than a pure sequential search.

Inequality join A SQL join where the comparison is made with an inequality (greater than or less than) instead of an equality operator. Useful for placing data into categories based on ranges of data. The general form is also known as a theta join.

Inheritance In object-oriented design, the ability to define new classes that are derived from higher-level classes. New classes inherit all prior properties and methods, so the programmer only needs to define new properties and methods.

INNER JOIN A SQL equality join condition. Rows from two tables are joined if the columns match exactly. The most common join condition. Rows that have no match in the other table are not displayed.

Input mask A special string that defines how data can be entered. Used to control the way users enter values into text boxes, such as requiring numeric values for currency items.

InputBox A predefined simplistic Window form that might be used to get one piece of data from the user. But it is better to avoid it and create your own form.

INSERT Two SQL commands that insert data into a table. One version inserts a single row at a time. The other variation copies selected data from one query and appends it as new rows in a different table.

Insertion anomaly Problems that arise when you try to insert data into a table that is not in third normal form. For example, if you find yourself repeatedly entering the same data (e.g., a customer's address), the table probably needs to be redefined.

Internet A collection of computers loosely connected to exchange information worldwide. Owners of the computers make files and information available to other users.

INTERSECT A set operation on rows of data from two SELECT statements. Only rows that are in both statements will be retrieved. *See UNION.*

Intranet A network internal to a company that uses Internet technologies to share data.

Isolation The transaction requirement that says the system must give each transaction the perception that it is running in isolation with no concurrent access issues.

Isolation level Used to assign locking properties in transactions. At a minimum, it is used to specify optimistic or pessimistic locks.

Some systems support intermediate levels.

Iteration Causing a section of code to be executed repeatedly, such as the need for a loop to track through each row of data. Typical commands include Do … Loop, and For … Next.

Java A programming language developed by Sun Microsystems that is supposed to be able to run unchanged on diverse computers. Originally designed as a control language for embedded systems, Java is targeted for Internet applications. The source of many bad puns in naming software products.

Java 2 enterprise edition (J2EE) A back-end server-based system for building complex applications. It is based on Java but consists of an entire environment.

JDBC A set of methods to connect Java code to databases. Similar in purpose to ADO, but works only in Java. Sometimes referred to as Java Database Connectivity.

JOIN When data is retrieved from more than one table, the tables must be joined by some column or columns of data. *See INNER JOIN and LEFT JOIN.*

Keyspace The top level namespace in the Cassandra non-relational DBMS. It functions like a schema in SQL to separate tables by application. Table names must be unique within a keyspace but different keyspaces can have table names that exist in other keyspaces.

Key-value pair A common method of identifying and transferring data in Web applications that is used for storage in non-relational DBMSs such as Cassandra. A key number uniquely identifies a collection of data contained in the value. For example, (CID, Customer Data) would enable a single number (CID) to retrieve data associated with the specified customer.

Label A simple text item displayed on a form or report. Values cannot be altered by the user.

Latency Time delay in a system. In a Web-based system, the delay created by slow links. Long download times create higher latency which leads to more server conflicts.

Leaves The bottom nodes in a B-tree (opposite from the root).

LEFT JOIN An outer join that includes all of the rows from the "left" table, even if there are no matching rows in the "right" table. The missing values are indicated by Nulls. *See right join and inner join.* Left and right are defined by the order the tables are listed; left is first.

Lifetime The length of time that a programming variable stays available. For example, variables created within subroutines are created when the routine is executed and then destroyed when it exits. Global variables stay alive for all routines within the module.

Lift The potential gain attributed to an association rule compared to purchases without the rule.

LIKE The SQL pattern-matching operator used to compare string values. The standard uses percent (%) to match any number of characters, and underscore (_) to match a single character. Some systems (e.g., Access) use an asterisk (*) and a question mark (?) instead.

LIMIT The CQL keyword used in the Cassandra SELECT command to set the maximum number of rows retrieved. A default value is always used even if the LIMIT command is not specified, so beware of queries that do not retrieve the expected number of rows.

List One type of Cassandra data collection (along with map and set). It enables a single column to hold multiple values, such as multiple e-mail addresses for one person. A list stores items as a sequence (1, 2, 3, …)

List box A control on a form that displays a list of choices. The list is always displayed and takes up a fixed amount of space on the screen.

List of values (LOV) An important technique for selecting data on Oracle forms. Used instead of combo/select boxes, the Oracle form maintains a small, buffered list of data that can be selected for a text box. Particularly useful in distributed databases because only portions of the list are sent to the user.

Local area network (LAN) A collection of personal computers within a small geographic area. All components of the network are owned or controlled by one company.

Local variable A variable defined within a subroutine. It can be accessed only within that subroutine and not from other procedures.

Logic Logic statements that define a program's purpose and structure. Can be written in pseudocode independently of the program's syntax. Logic structures include loops, conditions, subroutines, and input/output commands.

Logical security Determining which users should have access to which data. It deals with preventing three data problems: (1) unauthorized disclosure, (2) unauthorized modification, and (3) unauthorized withholding.

Loop Each loop must have a beginning, an end condition, and some way to increment a variable. *See iteration.*

Map One type of Cassandra data collection (along with list and set). It enables a single column to hold multiple values stored as key-value pairs. For instance, a phone map might hold items such as cell=1111, home=2222, work=3333.

Market basket analysis *See association rules.*

Master-detail A common one-to-many relationship often found on business forms, where the main form (e.g., Order) displays data for the master component, and a subform (e.g., Order Items) displays detail data. Sometimes called a parent-child relationship.

Measure The numeric data displayed in an OLAP cube.

Menu A set of application commands grouped together—usually on a toolbar. It provides an easy reference for commonly used commands and highlights the structure of the application.

Metadata Data about data, or the description of the data tables and columns. Usually held in the data dictionary. For example, table definitions and column domains are metadata.

Method A function or operation that a class can perform. For example, a Customer class would generally have an AddNew method that is called whenever a new customer object is added to the database.

Microsoft Assistance Markup Language (MAML) A help system authoring language similar to XML introduced by Microsoft for use with the Windows Vista operating system. Discountinued.

Middle tier In a multi-tier application, a set of programs that lies between the front and back ends. It is generally used to define and process business rules.

Modal form A form that takes priority on the screen and forces the user to deal with it before continuing. It should be avoided because it interrupts the user.

Module (or package) A collection of subroutines, generally related to a common purpose.

MsgBox A predefined method in Windows for displaying a brief message on the screen and presenting a few limited choices to the user. Because it is modal and interrupts the user, it should be used sparingly.

Multidimensional expressions (MDX) A language for querying OLAP data that was initiated by Microsoft, but in the process of becoming a standard. It is designed to handle relatively complex computations and OLAP cubes.

Multiplicity The UML term for signifying the quantities involved in an association. It is displayed on an association line with a minimum value, an ellipses (…), and a maximum value or asterisk (*) for many. For example, a customer can place from zero to many orders, so the multiplicity is (0…*).

N-ary association An association among three or more classes. It is drawn as a diamond on a UML class diagram. The term comes from extending English terms: unary means one, binary means two, ternary means four; so N-ary means many.

Naming conventions Program teams should name their variables and controls according to a consistent format. One common approach is to use a three letter prefix to identify the type of variable, followed by a descriptive name.

Nested conditions Conditional statements that are placed inside other conditional statements. For example, If (x > 0) Then … If (y < 4) Then … Some nested conditions can be replaced with a Case statement.

Nested query *See subquery.*

Network attached storage (NAS) Disk drives or other storage media that are connected to servers via a network connection instead of building them into the computer itself. The separation provides some autonomy, making it easier to provide backup and to replace the storage devices. Typically the drives are RAID configured.

Network database An older DBMS type that expanded the hierarchical database by supporting multiple connections between entities. A network database is characterized by the requirement that all connections had to be supported by an index.

Node An entry in a B-tree that contains a key value and links to other nodes. The top-most node is the root, the bottom nodes are leaves.

Non-relational database Several newer database designs were developed to optimize performance for specific tasks--primarily for Web-based applications. The data storage does not follow data normalization rules and imposes limitations on queries to store and retrieve huge amounts of data.

Normalization *See data normalization.*

NoSQL Read as either Not SQL or Not Only SQL. Often used to refer to non-relational databases where data storage does not follow normalization rules. More specifically, the tools rarely support JOIN queries. By restricting operations allowed, the systems are optimized for specific data storage and retrieval tasks for giant databases.

NOT The SQL negation operator. Used in the WHERE clause to reverse the truth value of a statement. *See DeMorgan's law.*

NotInList An event corresponding to a combo box in Access. It is triggered when a user enters a value that does not yet exist in the selected list. Often used to add new data to a table, such as new customers.

Null A missing (or currently unassigned) value.

Object An instance or particular example of a class. For example, in an Employee class, one individual employee would be an object. In a relational environment, a class is stored as a table, while an individual row in the table contains data for one object.

Object browser A tool provided within Microsoft software that displays properties and methods for available objects.

Object-oriented database management system (OODBMS) A database system that holds objects, including properties and methods. It supports links between objects, including inheritance.

Object-oriented programming A programming methodology where code is encapsulated within the definition of various objects (or classes). Systems are built from (hopefully) reusable objects. You control the system by manipulating object properties and calling object methods.

Offset, file A method of identifying the location of data within an operating system file. The offset is the number of bytes from the start of the file. It is often used as a pointer to the physical data items.

Online analytical processing (OLAP) The use of a database for data analysis. The focus is on retrieval of the data. The primary goals are to provide acceptable response times, maintain security, and make it easy for users to find the data they need.

Online transaction processing (OLTP) The use of a database for transaction processing. It consists of many insert and update operations and supports hundreds of concurrent accesses. High-speed storage of data, reliability, and data integrity are primary goals. Examples include airline reservations, online banking, and retail sales.

Open database connectivity (ODBC) A standard created by Microsoft to enable software to access a variety of databases. Each DBMS vendor provides an ODBC driver. Application code can generally be written once. To change the DBMS, you simply install and set up the proper ODBC driver.

Optimistic lock A transaction lock that does not block other processes. If the data is changed between read and write steps, the system generates an error that must be handled by code.

Option button A round button that is used to indicate a choice. By the design guide, option buttons signify mutually exclusive choices, as opposed to check boxes.

ORDER BY The clause in the SQL SELECT statement that lists the columns to sort the output. The modifiers ASC and DESC are used to specify ascending and descending sort orders.

Outer join A generic term that represents a left join or a right join. It returns rows from a table, even if there is no matching row in the other table.

Pack A maintenance operation that must be periodically performed on a database to remove fragments of deleted data.

Package A UML mechanism to group logical elements together. It is useful for isolating sections of a design. Packages can provide an overview of the entire system without having to see all the details.

Page footer A report element that appears at the bottom of every page. Often used for page numbers.

Page header A report element that appears at the top of every page. Often used for column headings and subtitles.

Parameter A variable that is passed to a subroutine or function and used in its computations.

Partition, SQL A SQL 2003 standard used to implement OLAP computations. Similar to a GROUP BY clause because you specify columns whose values are used to define the partitions. But partitions offer additional options and enable you to display detail and aggregate data at the same time. Also known as a data window.

Pass-by-reference A subroutine parameter that can be altered within the subroutine. If it is altered, the new value is returned to the calling program. That is, the subroutine can alter variables in other parts of the code. Usually a dangerous approach. *See pass-by-value.*

Pass-by-value A subroutine parameter that cannot be altered within the subroutine. Only its value is passed. If the subroutine changes the value, the original value in the calling program is not changed.

Pass-through query SQL queries that are ignored by Access on the front end. They enable you to write complex SQL queries that are specific to the server database.

Peer-to-peer Refers to distributed databases and servers. No single server or DBMS is in charge of coordinating processing. All devices are considered equal and queries can be addressed to any server (usually the closest one).

Persistent objects In object-oriented programming, the ability to store objects (in a file or database) so that they can be retrieved at a later date.

Persistent stored modules (PSM) In SQL-99 a proposed method for storing methods associated with objects. The module code would be stored and retrieved automatically by the DBMS.

Pessimistic lock A complete isolation level that blocks other processes from reading a locked piece of data until the transaction is complete. Program code will receive an error message if the data element is locked.

Physical security The branch of security that involves physically protecting the equipment and people. It includes disaster planning, physical access to equipment, and risk analysis and prevention.

PivotTable Microsoft's tool for enabling managers to examine OLAP data dynamically—typically inside of an Excel spreadsheet. The data is extracted from the database or OLAP cube, and managers can click buttons to examine summaries or details.

Pointer A logical or physical address of a piece of data.

Polymorphism In a class hierarchy each new class inherits methods from the prior classes. Through polymorphism you can override those definitions and assign a new method (with the same name) to the new class.

Primary key A column or set of columns that identify a particular row in a table. In non-relational DBMSs, the primary key defines how the data will be stored and retrieved.

Private key In a dual-key encryption system, the key that is never revealed to anyone else. A message encrypted with a public key can be decrypted only with the matching private key.

Procedural language A traditional programming language that is based on following procedures and is typically executed one statement at a time. Compared to SQL, which operates on sets of data with one command.

Procedure A subroutine or function that is designed to perform one specific task. It is generally wise to keep procedures small.

Property An attribute or feature of an entity that we wish to track. The term is often applied in an object-oriented context. *See attribute.*

Prototype An initial outline of an application that is built quickly to demonstrate and test various features of the application. Often used to help users visualize and improve forms and reports.

Pseudo column A computed column in a table that can only display data, not accept updates. Used to predefine row-wise calculations that are commonly used, such as Value=price*quantity.

Public key In a dual-key encryption system, the key that is given to the public. A message encrypted with a public key can only be decrypted with the matching private key.

Query by example (QBE) A fill-in-the-form approach to designing queries. You select tables and columns from a list and fill in blanks for conditions and sorting. It is relatively easy to use, requires minimal typing skills, generally comes with a Help system, and is useful for beginners.

RANK A data mining extension to SQL that rank orders data. If ties exist, they receive the same rank, and the next entry receives rank that is equal to its position in the overall list. Compare to DENSE RANK.

Rapid application development (RAD) A systems design methodology that attempts to reduce development time through efficiency and overlapping stages.

Redundant array of independent drives (RAID) A collection of several inexpensive disk drives that are treated as a single drive by the operating system. Data is written in stripes across the drives, supporting parallel operations to substantially improve performance and backup at the same time.

Referential integrity A data integrity constraint where data can be entered into a foreign key column only if the data value already exists in the base table. For example, clerks should not be able to enter an Order for CustomerID 1173 if CustomerID 1173 is not in the Customer table.

Reflexive association A relationship from one class back to itself. Most commonly seen in business in an Employee class, where some employees are managers over other employees.

Reflexive join A situation that exists when a table is joined to itself through a second column. For example, the table Employee(EmployeeID, …, ManagerID) could have a join from ManagerID to EmployeeID.

Regular expression (RegEx) A pattern matching tool for searching complex strings. Search online references for details. It has many options to match individual characters, sets of characters, repetitions, and sequences.

Relational database The most popular type of DBMS. All data is stored in tables (sometimes called relations). Tables are logically connected by the data they hold (e.g., through the key values). Relational databases should be designed through data normalization.

Relationship An association between two or more entities. *See association.*

Repeating groups Groups of data that repeat, such as items being ordered by a customer, multiple phone numbers for a client, and tasks assigned to a worker.

Replicate Make a deliberate copy of a database so it can be distributed to a new location and the contents later synchronized with the master copy.

Replication manager In a distributed database that relies on replication, the manager is an automated system that transfers changes to the various copies of the database. It has to handle conflicts if two people changed the same data before it was replicated.

Report footer A report element that appears at the end of the report. Often used for summary statistics or graphs.

Report header A report element that appears only at the start of the report. Often used for title pages and overviews.

Report services A server-based tool that processes reports and enables users to browse for reports and generate them on demand.

Report writer A DBMS tool that enables you to set up reports on the screen to specify how items will be displayed or calculated. Most of these tasks are performed by dragging data onto the screen.

Resource file A special text file associated with a form or report. It is commonly used to hold language translations. The form references a tag in the resource file, and the system pulls the matching value from the appropriately translated resource file for the desired language.

Resume A VBA error-handling operator that tells the processor to return to a new location and continue evaluating the code. Resume by itself returns to the line that caused the error. Resume Next returns to the line immediately following the error. Resume <label> sends the processor to a new location.

REVOKE The SQL command used to remove permissions that were granted to certain users.

RIGHT JOIN An outer join that includes all the rows from the "right" table, even if there are no matching rows in the "left" table. The missing values are indicated by Nulls. *See right join and inner join.* Left and right are defined by the order the tables are listed; left is first.

Rivest-Shamir-Adelman (RSA) encryption A dual-key encryption system that was patented in the United States by three mathematicians. *See dual-key encryption.*

Role Security group where permissions are assigned by the task or role that is performed instead of to individual people. Roles make it easier to change employee permissions.

Roll back A database system transaction feature. If an error occurs in a sequence of changes, the preceding changes can be rolled back to restore the database to a safe state with correct data.

Roll forward If an error occurs in processing transactions, the database can be restarted and loaded from a known checkpoint. Then partially completed transactions can be rolled forward to record the interrupted changes.

Roll up The act of aggregating detail data into a display of category summaries. Commonly used in examining data in a data warehouse or OLAP application. *See drill down.*

Root The top-most or entry node in a B-tree.

Row-by-row calculations The way that SQL performs in-line calculations. For example, the statement SELECT Price*Quantity AS Extended goes through each row and multiplies the row's value for Price times the matching value for Quantity.

Scalability The ability to handle increased demands without needing to make major changes to an application. Generally applied to server hardware, it is often supported through server clusters (or farms).

Schema A collection of tables that are grouped together for a common purpose. Generally used as a naming method.

Scope Refers to where a variable is accessible. Variables defined within a subroutine can typically be accessed only by code within that subroutine. Variables defined within a module are globally accessible by any code within that module.

Scroll bars A common graphical interface feature used to move material horizontally or vertically.

Second normal form (2NF) A table is in 2NF if every nonkey column depends on the entire key (not just part of it). This issue arises only if there is a concatenated key (with multiple columns).

Secure sockets layer (SSL) An Internet standard that uses RSA encryption with public keys to establish a secure, encrypted session between a browser and a Web server.

SELECT The primary data retrieval command for SQL. The main components are SELECT … FROM … INNER JOIN … WHERE.

Self-join A table joined to itself. *See reflexive join.*

Serialization A transaction requirement that specifies that each transaction is treated completely separately and run as if there were no other transactions.

Set One type of Cassandra data collection (along with list and map). It enables a single column to hold multiple values stored as key-value pairs. Order is unimportant but usually alphabetical. Columns are defined with angle brackets: email set<text>

Shell site *See cold site.*

Single-row form An input form that displays data from one row of a table at a time. The most common input form, since the designer has full control over the layout of the form.

Snapshot In a constantly changing database, you can take a snapshot that provides a copy of the data at one point in time.

Snowflake design A database design used for OLAP. A fact table is connected to dimension tables. The dimension tables can be joined to other dimension tables. It is less restrictive than the star design.

Solid-state drive (SSD) A data storage device that uses electronic flash (non-volatile) memory instead of moving disks. Modern versions are usually smaller than rotating disks, draw less power, and are tens or hundreds of times faster at data storage and retrieval.

Sparse table A table that contains missing data for many columns. Often an indicator that non-relational systems might be more efficient at storing and retrieving data because they use space only for actual data.

SQL A standardized database language, used for data retrieval (queries), data definition, and data manipulation.

SQL 1999 (SQL3) Approved in 1999, SQL 1999 was largely designed to add object-oriented features to the SQL language.

SQL 2003 An almost completed new version of SQL. Its primary contribution is a formal definition of integrating XML and multimedia into relational databases.

SQL 2008 The latest release of the SQL standard which is primarily an updated version of SQL 2006. Added support for XQuery and procedural code (persistent stored modules).

SQL injection attack An attack on the application and database usually performed by an outsider. The person enters special text into a text box to try and alter the SQL query. The special text often includes the comment character (--) and quotation marks. Your application code should test all user input and remove those characters.

Star design A database design used for OLAP. A fact table is connected to dimension tables that provide categories for analysis. More restrictive than the snowflake design, all dimension tables are directly connected to the fact table.

Startup form A form that is used to direct users to different parts of the application. Often used as the first form to appear. Options on the form should match the tasks of the users. Also called a switchboard form.

Storage area network (SAN) A high-speed network that moves disk drives out of the computer and into a separate location. Separating the disk drives from the processor makes it easier to share the data, to back it up, and to scale up by adding new processors.

Style sheet A special file that describes the desired layout, fonts, and styles for a set of Web pages. It is a powerful method to establish and change styles on many Web pages through making minor changes to one file.

Subform form A form that is displayed inside another (main) form. The data in the subform is generally linked to the row currently being displayed on the main form.

Subquery Using a second query to retrieve additional data within the main query. For example, to retrieve all sales where price was greater than the average, the WHERE clause could use a subquery to compute the average price.

Subroutine A separate section of code designed to perform one specific purpose. A subroutine can use parameters to exchange data with the calling routine.

Subtable In SQL 1999 a subtable inherits all of the columns from a base table. It provides inheritance similar to that of the abstract data types; however, all data is stored in separate columns.

Super-aggregate In a query, totals of totals. An important concept in OLAP queries provide by the ROLLUP option.

Support A data mining measure in association rules, measured by the percent of transactions that contain both items.

Surrogate keys Internally generated keys used to identify objects. They are often better keys than keys created by external sources because they are easier to guarantee uniqueness. See AutoNumber.

Switchboard form *See startup form.*

Synonym A short name for the full database path. Advantages of the short name are that it is easy to remember and the user never needs to know where the data is located. In addition, if everyone uses the synonym, database administrators can easily move the server databases to different locations just by altering the synonym's properties.

Syntax The specific format of commands that can be created in a program. Programs consist of logical steps, but each command must be given in the proper syntax for the compiler to understand it. Compilers generally check for syntax and prompt you with messages.

Tab order The sequence of controls followed on a form when the user presses the tab or return keys.

Table A collection of data for one class or entity. It consists of columns for each attribute and a row of data for each specific entity or object.

Tablespace In Oracle a tablespace is disk space that is allocated to hold tables, indexes, and other system data. You must first know the approximate size of the database.

Tabular form An input form that displays data in columns and rows. It is used when there are few columns of data or when the user needs to see multiple rows at the same time.

Template A special form (or report) that defines the standard elements to be applied to all forms. It also specifies styles and common code. It is used to improve consistency of forms and reports within an application.

Text box A common form control that is used to display and enter data.

Third normal form (3NF) A table is in third normal form (3NF) if each nonkey column depends on the whole key and nothing but the key.

Three-tier client/server A client/server system with a middle layer to hold code that defines business rules and consolidates access to various transaction servers.

Toggle button A three-dimensional variation of the check box. It is used to signify a choice of options.

Toolbar A small object in applications that can hold buttons and text menus. Users can execute commands with one or two mouse clicks. Used to hold frequently used commands, and commands that are used across the entire application, such as printing.

Tooltip A short message that is displayed when the user moves the mouse cursor over an item on the screen. Extremely useful for identifying the purpose of icons.

TOP A SQL SELECT clause provided by Access that restricts the displayed output to a specified number of rows. You can set the number of rows directly or use a percentage of the total number.

Transaction In a database application a transaction is a set of changes that must all be made together. Transactions must be identified to the DBMS and then committed or rolled back (if there is an error). For example, a transfer of money from one bank account to another requires two changes to the database—both must succeed or fail together.

Transaction log A sequential file that records transactions as they are being created. If something happens during the transaction update, the DBMS uses the transaction log to complete or roll back the incomplete transactions.

Transaction processing Collecting data for the purpose of recording transactions. Common examples include sales, human resource management, and financial accounting.

Trigger An event that causes a procedure to be executed. For example, clicking a button can be a trigger, as can a change in a data value.

Tunable consistency A concept emphasized in the Cassandra distributed DBMS. A developer can choose the level of consistency required for read and write operations, such as a single node, agreement across all nodes, or consistency across a quorum. Higher consistency (ALL) is closer to SQL but can result in delays.

Tuning Setting indexes, rewriting queries, and setting storage and other parameters to improve the database application performance.

Two-phase commit A mechanism for

handling concurrency and deadlock problems in a distributed database. In the first phase the coordinating DBMS sends updates to the other databases and asks them to prepare the transaction. Once they have agreed, the coordinator sends a message to commit the changes.

Unicode A standard method of storing and displaying a variety of character sets. Almost all current world character sets have been defined, as well as several ancient languages. It uses 2 bytes to represent each character, enabling it to handle over 65,000 characters or ideograms.

Unified Modeling Language (UML) A standardized modeling language for designing and documenting computer and business systems.

UNION An SQL clause to combine rows from two SELECT statements. Both queries must have the same number of columns with the same domains. Most systems also support INTERSECT and EXCEPT (or SUBTRACT) operators.

Universally unique identifier (uuid) A type of large number defined by an ISO standard to enable a computer to generate values that will be unique across any computer. Several types are defined (at least 4). Some use the unique MAC network card address and a timestamp. Others use pure random numbers, which might overlap with a tiny probability. Often used as generated key values in distributed (non-relational) databases.

UPDATE A SQL data manipulation command that changes the values in specified columns. A WHERE clause specifies which rows will be affected.

Use case In UML, a diagram that shows how a specific group of people will use the system.

User interface The look and feel of the application as it is seen by the user. Graphical interfaces are commonly employed in which users can manipulate icons and data on the screen to perform their tasks.

Validation tables Simple tables of one or two columns that contain standardized data for entry into other tables. For example, a list of departments would be stored in a validation table. To enter a department name into an Employee table, the user would be given a choice of the rows in the validation table.

VARCHAR A common method for storing character data. It stands for variable characters. Each column of data uses the exact amount of bytes needed to store the specific data.

Variable A location in memory used to hold temporary values. Variables have a scope and a lifetime depending on where they are created and how they are defined. They also have a specific data type, although the Variant data type in VBA can hold any common type of data.

Variable length storage A system of storing data columns where only a pointer is stored in the actual row. The actual data is stored in a pooled area.

Vertical partition Splitting a table into groups based on the columns of data. Large columns or columns that are seldom used (e.g., pictures) can be moved to slower, cheaper storage devices.

View A saved query. You can build new queries that retrieve data from the view. A view is saved as an SQL statement—not as the actual data.

Virtual machine (VM) A method of running multiple computer operating systems on one physical computer. Each VM has its own disk space and operating system and appears to be a separate computer. The underlying computer needs to support multiprocessing and have sufficient memory. VMs are relatively easy to backup and transfer to other physical computers.

Visual Basic (VB) A stand-alone programming language sold by Microsoft and used to develop applications for the Windows environment. The professional version supports database connections. The program can be compiled into a stand-alone executable file.

Visual Basic for Applications (VBA) The programming language that underlies almost all of Microsoft's tools, including Access.

Volume table of contents (VTOC) A design tool that can be used to outline the overall structure of an application. It generally shows a sequence of interrelated menus.

WHERE The SQL clause that restricts the rows that will be used in the query. It can also refer to data in subqueries.

Wide area network (WAN) A network that is spread across a larger geographic area. Parts of the network are outside the control of a single firm. Long-distance connections often use public carriers.

With … End With In VBA, a shortcut for examining or altering several properties for a single object. Once the object is specified in the With statement, you simply refer to the properties inside the "loop."

WITH GRANT OPTION A security permission option that transfers the ability to assign permissions to the specified role or user.

World Wide Web (WWW) A first attempt to set up an international database of information. Web browsers display graphical pages of information. Hypertext connections enable you to get related information by clicking highlighted words or icons. A standard method for displaying text and images on client computers.

XML *See extensible markup language.*

XML schema A definitional file that describes the tags, data types, and attributes allowed in an associated XML data file.

XQuery A standardized language to retrieve individual elements or groups of data from an XML file or text group.

Index