nine

Designing Databases

Monkey Business Images/Shutterstock

Chapter Objectives

After studying this chapter, you should be able to:

- Concisely define each of the following key database design terms: *relation, primary key, functional dependency, foreign key, referential integrity, field, data type, null value, denormalization, file organization, index,* and *secondary key.*
- Explain the role of designing databases in the analysis and design of an information system.
- Transform an entity-relationship (E-R) diagram into an equivalent set of well-structured (normalized) relations.
- Merge normalized relations from separate user views into a consolidated set of well-structured relations.
- Choose storage formats for fields in database tables.
- Translate well-structured relations into efficient database tables.
- Explain when to use different types of file organizations to store computer files.
- Describe the purpose of indexes and the important considerations in selecting attributes to be indexed.

Chapter Preview . . .

In Chapter 7 you learned how to represent an organization's data graphically using an entity-relationship (E-R) diagram and Microsoft Visio. In this chapter, you learn guidelines for clear and efficient data files and about logical and physical database design. It is likely that the human interface and database design steps will happen in parallel, as illustrated in the SDLC in Figure 9-1.

Logical and physical database design has five purposes:

1. Structure the data in stable structures that are not likely to change over time and that have minimal redundancy.
2. Develop a logical database design that reflects the actual data requirements that exist in the forms (hard copy and computer displays) and reports of an information system. For this reason, database design is often done in parallel with the design of the human interface of an information system.
3. Develop a logical database design from which we can do physical database design. Because most information systems today use relational database management systems, logical database design usually uses a relational database model, which represents data in simple tables with common columns to link related tables.
4. Translate a relational database model into a technical file and database design.
5. Choose data-storage technologies (such as hard disk, CD-ROM, or flash disk) that will efficiently, accurately, and securely process database activities.

The implementation of a database (i.e., creating and loading data into files and databases) is done during the next phase of the systems development life cycle. Because implementation is technology specific, we address implementation issues only at a general level in Chapter 10.

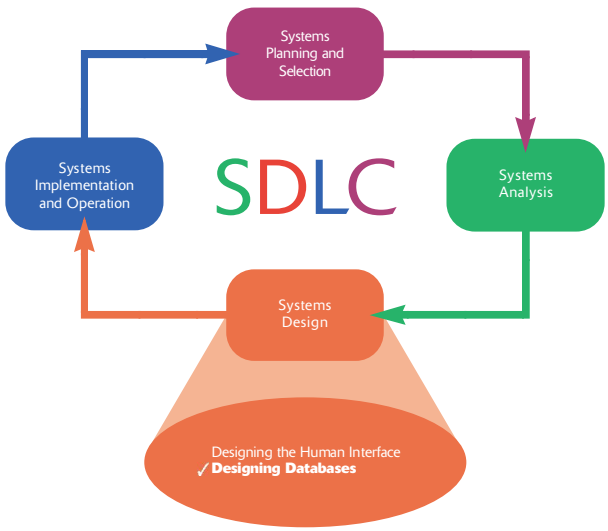


FIGURE 9-1
Systems development life cycle. Systems analysts design databases during the systems design phase. Database design typically occurs in parallel with other design steps.

Database Design

File and database design occurs in two steps. You begin by developing a logical database model, which describes data using a notation that corresponds to a data organization used by a database management system. This system software is responsible for storing, retrieving, and protecting data (such as Microsoft Access, Oracle, or SQL Server). The most common style for a logical database model is the relational database model. Once you develop a clear and precise logical database model, you are ready to prescribe the technical specifications for computer files and databases in which to store the data ultimately. A physical database design provides these specifications.

You typically do logical and physical database design in parallel with other systems design steps. Thus, you collect the detailed specifications of data necessary for logical database design as you design system inputs and outputs. Logical database design is driven not only from the previously developed E-R data model for the application but also from form and report layouts. You study data elements on these system inputs and outputs and identify interrelationships among the data. As with conceptual data modeling, the work of all systems development team members is coordinated and shared through the project dictionary or repository. The designs for logical databases and system inputs and outputs are then used in physical design activities to specify to computer programmers, database administrators, network managers, and others how to implement the new information system. We assume for this text that the design of computer programs and distributed information processing and data networks are topics of other courses, so we concentrate on the aspect of physical design most often undertaken by a systems analyst—physical file and database design.

The Process of Database Design

Figure 9-2 shows that database modeling and design activities occur in all phases of the systems development process. In this chapter we discuss methods that help you finalize logical and physical database designs during the design phase. In logical database design you use a process called *normalization*, which is a way to build a data model that has the properties of simplicity, nonredundancy, and minimal maintenance.

In most situations, many physical database design decisions are implicit or eliminated when you choose the data-management technologies to use with the application. We concentrate on those decisions you will make most frequently and use Microsoft Access to illustrate the range of physical database design parameters you must manage. The interested reader is referred to Hoffer, Ramesh, and Topi (2011) for a more thorough treatment of techniques for logical and physical database design.

Four steps are key to logical database modeling and design:

1. Develop a logical data model for each known user interface (form and report) for the application, using normalization principles.
2. Combine normalized data requirements from all user interfaces into one consolidated logical database model; this step is called *view integration*.
3. Translate the conceptual E-R data model for the application, developed without explicit consideration of specific user interfaces, into normalized data requirements.
4. Compare the consolidated logical database design with the translated E-R model and produce, through view integration, one final logical database model for the application.

During physical database design, you use the results of these four key logical database design steps. You also consider definitions of each attribute; descriptions

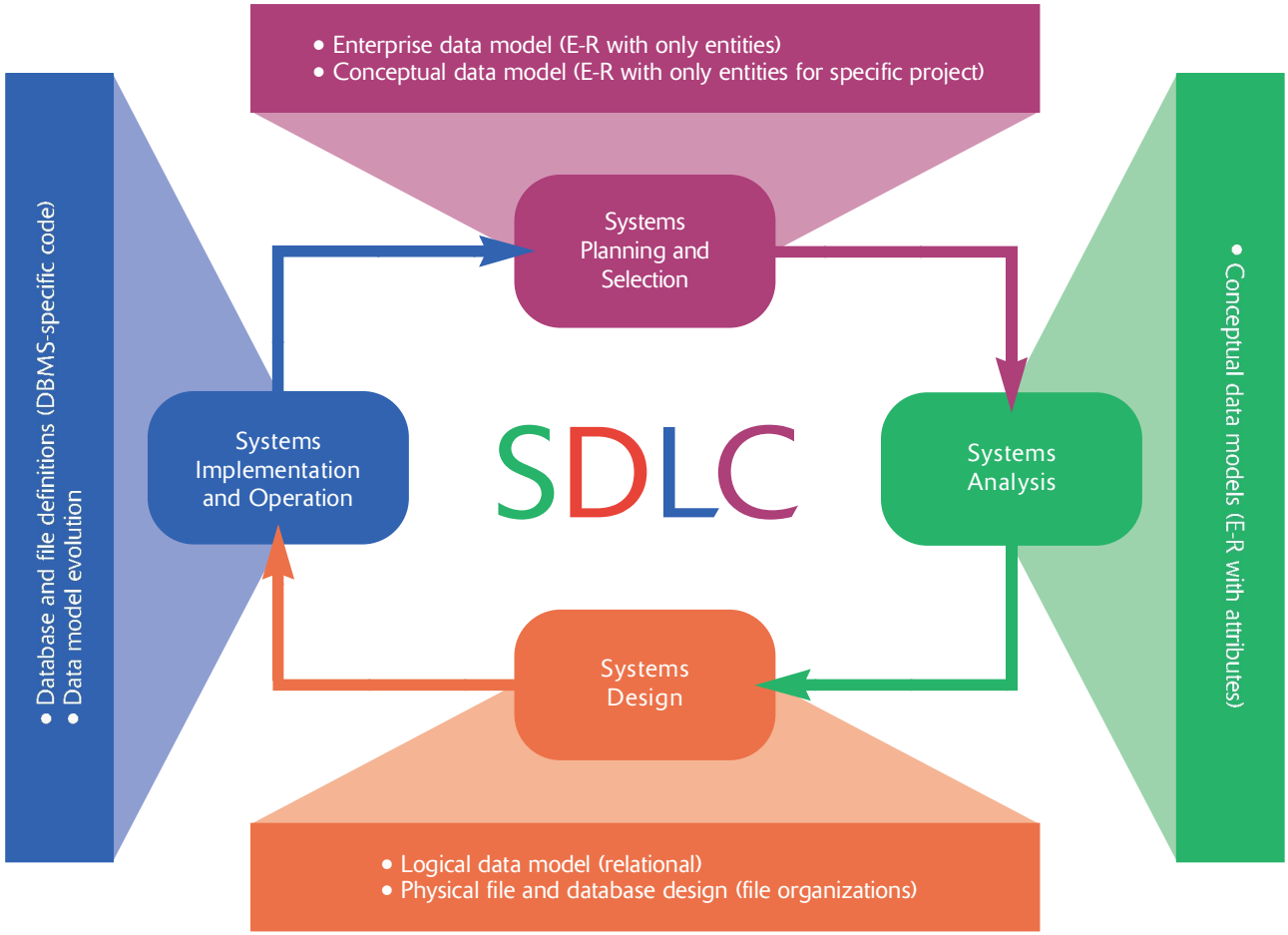


FIGURE 9-2
Relationship between data modeling and the systems development life cycle.

of where and when data are entered, retrieved, deleted, and updated; expectations for response time and data integrity; and descriptions of the file and database technologies to be used. These inputs allow you to make key physical database design decisions, including the following:

1. Choosing the storage format (called *data type*) for each attribute from the logical database model; the format is chosen to minimize storage space and to maximize data quality. Data type involves choosing length, coding scheme, number of decimal places, minimum and maximum values, and potentially many other parameters for each attribute.
2. Grouping attributes from the logical database model into physical records (in general, this is called *selecting a stored record*, or *data structure*).
3. Arranging related records in secondary memory (hard disks and magnetic tapes) so that individual and groups of records can be stored, retrieved, and updated rapidly (called *file organizations*). You should also consider protecting data and recovering data after errors are found.
4. Selecting media and structures for storing data to make access more efficient. The choice of media affects the utility of different file organizations. The primary structure used today to make access to data more rapid is key indexes, on unique and nonunique keys.

In this chapter we show how to do each of the logical database design steps and discuss factors to consider in making each physical file and database design decision.

Deliverables and Outcomes

During logical database design, you must account for every data element on a system input or output—form or report—and on the E-R model. Each data element (like customer name, product description, or purchase price) must be a piece of raw data kept in the system's database, or in the case of a data element on a system output, the element can be derived from data in the database. Figure 9-3 illustrates the outcomes from the four-step logical database design process. Figures 9-3A and 9-3B (step 1) contain two sample system outputs for a customer order processing system at Pine Valley Furniture. A description of the associated database requirements, in the form of what we call *normalized relations*, is listed below each output diagram. Each relation (think of a relation as a table with rows and columns) is named, and its attributes (columns) are listed within parentheses. The **primary key** attribute—that attribute whose value is unique across all occurrences of the relation—is indicated by an underline, and an attribute of a relation that is the primary key of another relation is indicated by a dashed underline.

Primary key
An attribute whose value is unique across all occurrences of a relation.

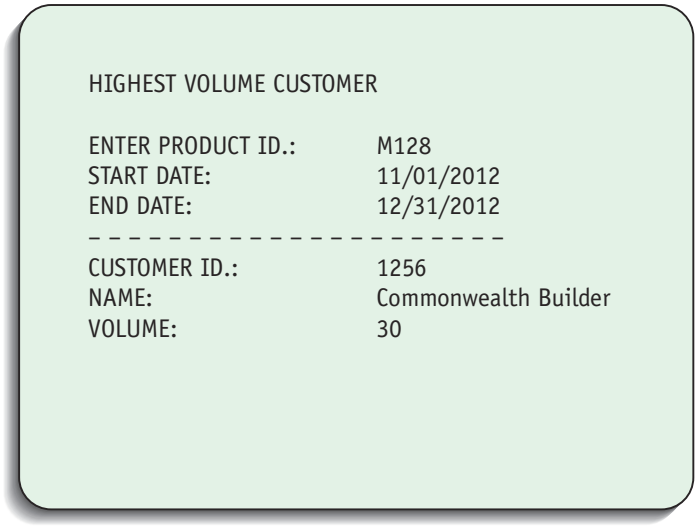
In Figure 9-3A data are shown about customers, products, and the customer orders and associated line items for products. Each of the attributes of each relation either appears in the display or is needed to link related relations. For example, because an order is for some customer, an attribute of ORDER is the associated Customer_ID. The data for the display in Figure 9-3B are more complex. A backlogged product on an order occurs when the amount ordered (Order_Quantity) is less than the amount shipped (Ship_Quantity) for invoices associated with an order. The query refers to only a specified time period, so the Order_Date is needed. The INVOICE Order_Number links invoices with the associated order.

Figure 9-3C (step 2) shows the result of integrating these two separate sets of normalized relations. Figure 9-3D (step 3) shows an E-R diagram for a customer order processing application that might be developed during conceptual data modeling along with equivalent normalized relations. Figure 9-3E (step 4) shows a set of normalized relations that would result from reconciling the logical database designs of Figures 9-3C and 9-3D. Normalized relations like those in Figure 9-3E are the primary deliverable from logical database design.

Finally, Figure 9-3F shows the E-R diagram drawn in Microsoft Visio. Visio actually shows the tables and relationships between the tables from the normalized relations. Thus, the associative entities, LINE ITEM and SHIPMENT, are shown as entities on the Visio diagram; we do not place relationship names on either side of these entities on the Visio diagram because these represent associative entities. Visio also shows for these entities the primary keys of the associated ORDER, INVOICE, and PRODUCT entities. Also, note that the lines for the Places and Bills relationships are dashed. This Visio notation indicates that ORDER and INVOICE have their own primary keys that do not include the primary keys of CUSTOMER and ORDER, respectively (what Visio calls non-identifying relationships). Because LINE ITEM and SHIPMENT both include in their primary keys the primary keys of other entities (which is common for associative entities), the relationships around LINE ITEM and SHIPMENT are identifying, and hence the relationship lines are solid.

It is important to remember that relations do not correspond to computer files. In physical database design, you translate the relations from logical database design into specifications for computer files. For most information

A

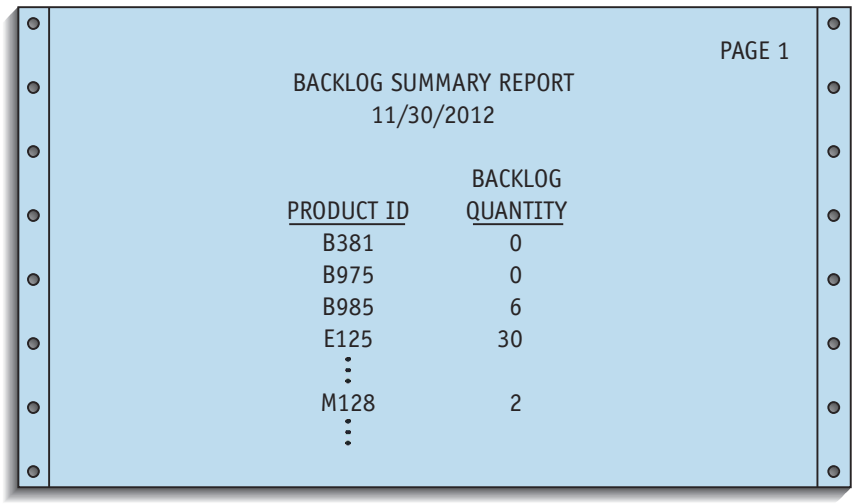


This inquiry screen shows the customer with the largest volume total sales of a specified product during an indicated time period.

Relations:

CUSTOMER(Customer_ID,Name)
ORDER(Order_Number,Customer_ID,Order_Date)
PRODUCT(Product_ID)
LINE ITEM(Order_Number,Product_ID,Order_Quantity)

B



This report shows the unit volume of each product that has been ordered less than amount shipped through the specified date.

Relations:

PRODUCT(Product_ID)
LINE ITEM(Product_ID,Order_Number,Order_Quantity)
ORDER(Order_Number,Order_Date)
SHIPMENT(Product_ID,Invoice_Number,Ship_Quantity)
INVOICE(Invoice_Number,Invoice_Date,Order_Number)

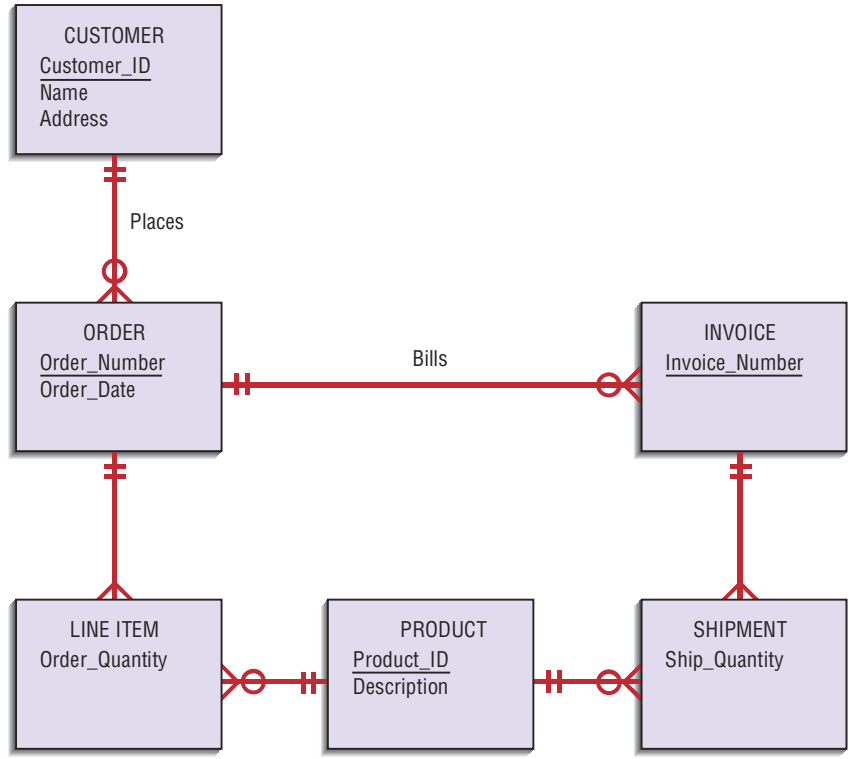
C

CUSTOMER(Customer_ID,Name)
PRODUCT(Product_ID)
INVOICE(Invoice_Number,Invoice_Date,Order_Number)
ORDER(Order_Number,Customer_ID,Order_Date)
LINE ITEM(Order_Number,Product_ID,Order_Quantity)
SHIPMENT(Product_ID,Invoice_Number,Ship_Quantity)

FIGURE 9-3
Simple example of logical data modeling: (A) Highest-volume customer query screen, (B) Backlog summary report, (C) Integrated set of relations, (D) Conceptual data model and transformed relations, (E) Final set of normalized relations, (F) Microsoft Visio E-R diagram.

FIGURE 9-3
(*continued*)

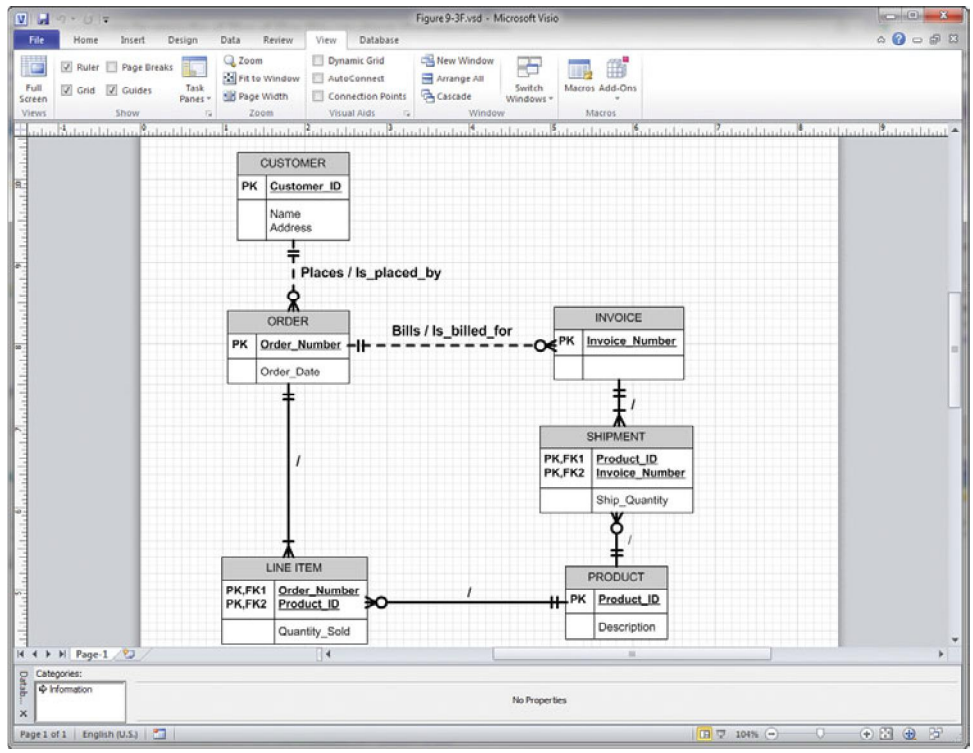
D



E

CUSTOMER(Customer_ID,Name,Address)
PRODUCT(Product_ID,Description)
ORDER(Order_Number,Customer_ID,Order_Date)
LINE ITEM(Order_Number,Product_ID,Order_Quantity)
INVOICE(Invoice_Number,Order_Number,Invoice_Date)
SHIPMENT(Invoice_Number,Product_ID,Ship_Quantity)

F



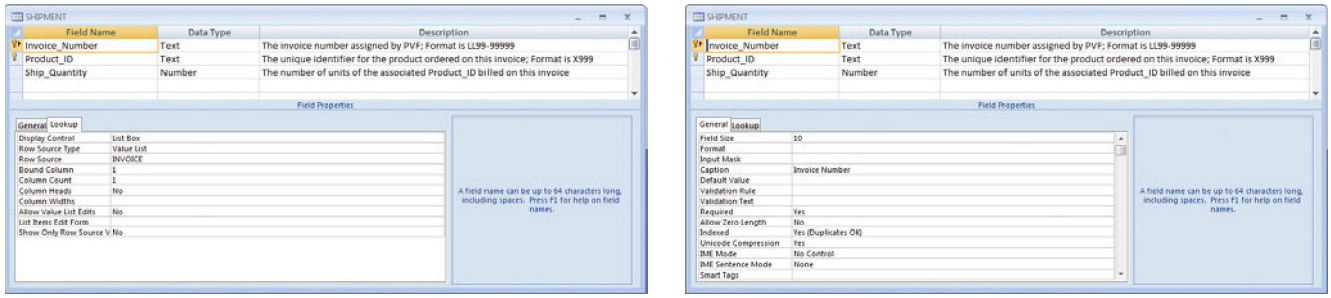


FIGURE 9-4
Definition of SHIPMENT table in Microsoft Access: (A) Table with invoice_number properties, (B) Invoice_number lookup properties.

systems, these files will be tables in a relational database. These specifications are sufficient for programmers and database analysts to code the definitions of the database. The coding, done during systems implementation, is written in special database definition and processing languages, such as Structured Query Language (SQL), or by filling in table definition forms, such as with Microsoft Access. Figure 9-4 shows a possible definition for the SHIPMENT relation from Figure 9-3E using Microsoft Access. This display of the SHIPMENT table definition illustrates choices made for several physical database design decisions.

- All three attributes from the SHIPMENT relation, and no attributes from other relations, have been grouped together to form the fields of the SHIPMENT table.
- The Invoice_Number field has been given a data type of Text, with a maximum length of 10 characters.
- The Invoice_Number field is required because it is part of the primary key for the SHIPMENT table (the value that makes every row of the SHIPMENT table unique is a combination of Invoice_Number and Product_ID).
- An index is defined for the Invoice_Number field, but because there may be several rows in the SHIPMENT table for the same invoice (different products on the same invoice), duplicate index values are allowed (so Invoice_Number is what we will call a *secondary key*).
- The Invoice_Number, because it references the Invoice_Number from the INVOICE table, is defined as a Lookup to the first column (Invoice_Number) of the INVOICE table; in this way, all values that are placed in the Invoice_Number field of the SHIPMENT table must correspond to a previously entered invoice.

Many other physical database design decisions were made for the SHIPMENT table, but they are not apparent on the display in Figure 9-4. Further, this table is only one table in the PVF Order Entry database, and other tables and structures for this database are not illustrated in this figure.

Relational Database Model

Many different database models are in use and are the basis for database technologies. Although hierarchical and network models have been popular in the past, they are not often used today for new information systems. Object-oriented database models are emerging but are still not common. The vast majority of information systems today use the relational database model.

FIGURE 9-5
EMPLOYEE1 relation with sample data.

EMPLOYEE1

Emp_ID	Name	Dept	Salary
100	Margaret Simpson	Marketing	42,000
140	Allen Beeton	Accounting	39,000
110	Chris Lucero	Info Systems	41,500
190	Lorenzo Davis	Finance	38,000
150	Susan Martin	Marketing	38,500

Relational database model
Data represented as a set of related tables or relations.

Relation
A named, two-dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of unnamed rows.

The **relational database model** represents data in the form of related tables or relations. A **relation** is a named, two-dimensional table of data. Each relation (or table) consists of a set of named columns and an arbitrary number of unnamed rows. Each column in a relation corresponds to an attribute of that relation. Each row of a relation corresponds to a record that contains data values for an entity.

Figure 9-5 shows an example of a relation named EMPLOYEE1. This relation contains the following attributes describing employees: Emp_ID, Name, Dept, and Salary. The table contains five sample rows, corresponding to five employees.

You can express the structure of a relation by a shorthand notation in which the name of the relation is followed (in parentheses) by the names of the attributes in the relation. The identifier attribute (called the *primary key* of the relation) is underlined. For example, you would express EMPLOYEE1 as follows:

Employee (Emp_ID, Name, Dept, Salary)

Not all tables are relations. Relations have several properties that distinguish them from nonrelational tables:

1. Entries in cells are simple. An entry at the intersection of each row and column has a single value.
2. Entries in columns are from the same set of values.
3. Each row is unique. Uniqueness is guaranteed because the relation has a nonempty primary key value.
4. The sequence of columns can be interchanged without changing the meaning or use of the relation.
5. The rows may be interchanged or stored in any sequence.

Well-Structured Relations

Well-structured relation (or table)
A relation that contains a minimum amount of redundancy and allows users to insert, modify, and delete the rows without errors or inconsistencies.

What constitutes a **well-structured relation** (or **table**)? Intuitively, a well-structured relation contains a minimum amount of redundancy and allows users to insert, modify, and delete the rows in a table without errors or inconsistencies. EMPLOYEE1 (Figure 9-5) is such a relation. Each row of the table contains data describing one employee, and any modification to an employee's data (such as a change in salary) is confined to one row of the table.

In contrast, EMPLOYEE2 (Figure 9-6) contains data about employees and the courses they have completed. Each row in this table is unique for the combination of Emp_ID and Course, which becomes the primary key for the table. It is not a well-structured relation, however. If you examine the sample data in the table, you notice a considerable amount of redundancy. For example, the Emp_ID, Name, Dept, and Salary values appear in two separate rows for employees 100, 110, and 150. Consequently, if the salary for employee 100 changes, we must record this fact in two rows (or more, for some employees).

The problem with this relation is that it contains data about two entities: EMPLOYEE and COURSE. You will learn to use principles of normalization to divide EMPLOYEE2 into two relations. One of the resulting relations is

EMPLOYEE2

Emp_ID	Name	Dept	Salary	Course	Date_Completed
100	Margaret Simpson	Marketing	42,000	SPSS	6/19/2012
100	Margaret Simpson	Marketing	42,000	Surveys	10/7/2012
140	Alan Beeton	Accounting	39,000	Tax Acc	12/8/2012
110	Chris Lucero	Info Systems	41,500	SPSS	1/12/2012
110	Chris Lucero	Info Systems	41,500	C++	4/22/2012
190	Lorenzo Davis	Finance	38,000	Investments	5/7/2012
150	Susan Martin	Marketing	38,500	SPSS	6/19/2012
150	Susan Martin	Marketing	38,500	TQM	8/12/2012

FIGURE 9-6
Relation with redundancy.

EMPLOYEE1 (Figure 9-5). The other we will call EMP COURSE, which appears with sample data in Figure 9-7. The primary key of this relation is the combination of Emp_ID and Course (we emphasize this by underlining the column names for these attributes).

Normalization

We have presented an intuitive discussion of well-structured relations, however, we need rules and a process for designing them. **Normalization** is a process for converting complex data structures into simple, stable data structures. For example, we used the principles of normalization to convert the EMPLOYEE2 table with its redundancy to EMPLOYEE1 (Figure 9-5) and EMP COURSE (Figure 9-7).

Normalization
The process of converting complex data structures into simple, stable data structures.

Rules of Normalization

Normalization is based on well-accepted principles and rules. The many normalization rules, are too numerous to cover in this text (see Hoffer, Ramesh, and Topi [2011] for more complete coverage). Besides the five properties of relations outlined previously, two other rules are frequently used.

1. *Second normal form (2NF).* Each nonprimary key attribute is identified by the whole key (what we call *full functional dependency*).
2. *Third normal form (3NF).* Nonprimary key attributes do not depend on each other (what we call *no transitive dependencies*).

The result of normalization is that every nonprimary key attribute depends upon the whole primary key and nothing but the primary key. We discuss second and third normal form in more detail next.

EMP COURSE

Emp_ID	Course	Date_ Completed
100	SPSS	6/19/2012
100	Surveys	10/7/2012
140	Tax Acc	12/8/2012
110	SPSS	1/22/2012
110	C++	4/22/2012
190	Investments	5/7/2012
150	SPSS	6/19/2012
150	TQM	8/12/2012

FIGURE 9-7
EMP COURSE relation.

Functional Dependence and Primary Keys

Functional dependency
A particular relationship between two attributes. For a given relation, attribute B is functionally dependent on attribute A if, for every valid value of A, that value of A uniquely determines the value of B. The functional dependence of B on A is represented by A→B.

Normalization is based on the analysis of functional dependence. A **functional dependency** is a particular relationship between two attributes. In a given relation, attribute B is functionally dependent on attribute A if, for every valid value of A, that value of A uniquely determines the value of B. The functional dependence of B on A is represented by an arrow, as follows: A→B (e.g., Emp_ID→Name in the relation of Figure 9-5). Functional dependence does not imply mathematical dependence—that the value of one attribute may be computed from the value of another attribute; rather, functional dependence of B on A means that there can be only one value of B for each value of A. Thus, for a given Emp_ID value, only one Name value can be associated with it; the value of Name, however, cannot be derived from the value of Emp_ID. Other examples of functional dependencies from Figure 9-3B are in ORDER, Order_Number→Order_Date, and in INVOICE, Invoice_Number→Invoice_Date and Order_Number.

An attribute may be functionally dependent on two (or more) attributes, rather than on a single attribute. For example, consider the relation EMP COURSE (Emp_ID, Course, Date_Completed) shown in Figure 9-7. We represent the functional dependency in this relation as follows: Emp_ID, Course Date_Completed. In this case, Date_Completed cannot be determined by either Emp_ID or Course alone, because Date_Completed is a characteristic of an employee taking a course.

You should be aware that the instances (or sample data) in a relation do not prove that a functional dependency exists. Only knowledge of the problem domain, obtained from a thorough requirements analysis, is a reliable method for identifying a functional dependency. However, you can use sample data to demonstrate that a functional dependency does not exist between two or more attributes. For example, consider the sample data in the relation EXAMPLE (A, B, C, D) shown in Figure 9-8. The sample data in this relation prove that attribute B is not functionally dependent on attribute A, because A does not uniquely determine B (two rows with the same value of A have different values of B).

Second Normal Form

Second normal form (2NF)
A relation for which every nonprimary key attribute is functionally dependent on the whole primary key.

A relation is in **second normal form (2NF)** if every nonprimary key attribute is functionally dependent on the whole primary key. Thus, no nonprimary key attribute is functionally dependent on a part, but not all, of the primary key. Second normal form is satisfied if any one of the following conditions apply:

1. The primary key consists of only one attribute (such as the attribute Emp_ID in relation EMPLOYEE1).
2. No nonprimary key attributes exist in the relation.
3. Every nonprimary key attribute is functionally dependent on the full set of primary key attributes.

FIGURE 9-8
EXAMPLE relation.

EXAMPLE

A	B	C	D
X	U	X	Y
Y	X	Z	X
Z	Y	Y	Y
Y	Z	W	Z

EMPLOYEE2 (Figure 9-6) is an example of a relation that is not in second normal form. The shorthand notation for this relation is:

EMPLOYEE2(Emp_ID, Name, Dept, Salary, Course, Date_Completed)

The functional dependencies in this relation are the following:

Emp_ID→Name, Dept, Salary
Emp_ID, Course→Date_Completed

The primary key for this relation is the composite key Emp_ID, Course. Therefore, the nonprimary key attributes Name, Dept, and Salary are functionally dependent on only Emp_ID but not on Course. EMPLOYEE2 has redundancy, which results in problems when the table is updated.

To convert a relation to second normal form, you decompose the relation into new relations using the attributes, called *determinants*, that determine other attributes; the determinants are the primary keys of these relations. EMPLOYEE2 is decomposed into the following two relations:

1. EMPLOYEE1(Emp_ID, Name, Dept, Salary): This relation satisfies the first second normal form condition (sample data shown in Figure 9-5).
2. EMP COURSE(Emp_ID, Course, Date_Completed): This relation satisfies second normal form condition three (sample data appear in Figure 9-7).

Third Normal Form

A relation is in **third normal form (3NF)** if it is in second normal form with no functional dependencies between two (or more) nonprimary key attributes (a functional dependency between nonprimary key attributes is also called a *transitive dependency*). For example, consider the relation SALES(Customer_ID, Customer_Name, Salesperson, Region) (sample data shown in Figure 9-9A).

Third normal form (3NF)
A relation that is in second normal form and has no functional (transitive) dependencies between two (or more) nonprimary key attributes.

The following functional dependencies exist in the SALES relation:

1. Customer_ID→Customer_Name, Salesperson, Region (Customer_ID is the primary key.)
2. Salesperson→Region (Each salesperson is assigned to a unique region.)

Notice that SALES is in second normal form because the primary key consists of a single attribute (Customer_ID). However, Region is functionally dependent on Salesperson, and Salesperson is functionally dependent on Customer_ID. As a result, data maintenance problems arise in SALES.

1. A new salesperson (Robinson) assigned to the North region cannot be entered until a customer has been assigned to that salesperson (because a value for Customer_ID must be provided to insert a row in the table).
2. If customer number 6837 is deleted from the table, we lose the information that salesperson Hernandez is assigned to the East region.
3. If salesperson Smith is reassigned to the East region, several rows must be changed to reflect that fact (two rows are shown in Figure 9-9A).

These problems can be avoided by decomposing SALES into the two relations, based on the two determinants, shown in Figure 9-9(B). These relations are the following:

SALES1(Customer_ID, Customer_Name, Salesperson)
SPERSON(Salesperson, Region)

Foreign key
An attribute that appears as a nonprimary key attribute in one relation and as a primary key attribute (or part of a primary key) in another relation.

Note that Salesperson is the primary key in SPERSON. Salesperson is also a foreign key in SALES1. A **foreign key** is an attribute that appears as a nonprimary key attribute in one relation (such as SALES1) and as a primary key

FIGURE 9-9
Removing transitive dependencies: (A) Relation with transitive dependency, (B) Relations in 3NF.

A

SALES

Customer_ID	Customer_Name	Salesperson	Region
8023	Anderson	Smith	South
9167	Bancroft	Hicks	West
7924	Hobbs	Smith	South
6837	Tucker	Hernandez	East
8596	Eckersley	Hicks	West
7018	Arnold	Faulb	North

B

SALES1

Customer_ID	Customer_Name	Salesperson
8023	Anderson	Smith
9167	Bancroft	Hicks
7924	Hobbs	Smith
6837	Tucker	Hernandez
8596	Eckersley	Hicks
7018	Arnold	Faulb

SPERSON

Salesperson	Region
Smith	South
Hicks	West
Hernandez	East
Faulb	North

attribute (or part of a primary key) in another relation. You designate a foreign key by using a dashed underline.

Referential integrity
An integrity constraint specifying that the value (or existence) of an attribute in one relation depends on the value (or existence) of the same attribute in another relation.

A foreign key must satisfy **referential integrity,** which specifies that the value of an attribute in one relation depends on the value of the same attribute in another relation. Thus, in Figure 9-9B, the value of Salesperson in each row of table SALES1 is limited to only the current values of Salesperson in the SPERSON table. Referential integrity is one of the most important principles of the relational model.

Transforming E-R Diagrams into Relations

Normalization produces a set of well-structured relations that contains all of the data mentioned in system inputs and outputs developed in human interface design. Because these specific information requirements may not represent all future information needs, the E-R diagram you developed in conceptual data modeling is another source of insight into possible data requirements for a new application system. To compare the conceptual data model and the normalized relations developed so far, your E-R diagram must be transformed into relational notation, normalized, and then merged with the existing normalized relations.

Transforming an E-R diagram into normalized relations and then merging all the relations into one final, consolidated set of relations can be accomplished in four steps. These steps are summarized briefly here, and then steps 1, 2, and 4 are discussed in detail in subsequent sections of this chapter.

1. *Represent entities.* Each entity type in the E-R diagram becomes a relation. The identifier of the entity type becomes the primary key of the relation, and other attributes of the entity type become nonprimary key attributes of the relation.
2. *Represent relationships.* Each relationship in an E-R diagram must be represented in the relational database design. How we represent a relationship depends on its nature. For example, in some cases we represent a relationship by making the primary key of one relation a foreign key of another relation. In other cases, we create a separate relation to represent a relationship.

3. *Normalize the relations.* The relations created in steps 1 and 2 may have unnecessary redundancy. So, we need to normalize these relations to make them well structured.
4. *Merge the relations.* So far in database design we have created various relations from both a bottom-up normalization of user views and from transforming one or more E-R diagrams into sets of relations. Across these different sets of relations, redundant relations (two or more relations that describe the same entity type) may need to be merged and renormalized to remove the redundancy.

Represent Entities

Each regular entity type in an E-R diagram is transformed into a relation. The identifier of the entity type becomes the primary key of the corresponding relation. Each nonkey attribute of the entity type becomes a nonkey attribute of the relation. You should check to make sure that the primary key satisfies the following two properties:

1. The value of the key must uniquely identify every row in the relation.
2. The key should be nonredundant; that is, no attribute in the key can be deleted without destroying its unique identification.

Some entities may have keys that include the primary keys of other entities. For example, an EMPLOYEE DEPENDENT may have a Name for each dependent, but, to form the primary key for this entity, you must include the Employee_ID attribute from the associated EMPLOYEE entity. Such an entity whose primary key depends upon the primary key of another entity is called a *weak entity.*

Representation of an entity as a relation is straightforward. Figure 9-10A shows the CUSTOMER entity type for Pine Valley Furniture. The corresponding CUSTOMER relation is represented as follows:

CUSTOMER(Customer_ID, Name, Address, City_State_ZIP, Discount)

In this notation, the entity type label is translated into a relation name. The identifier of the entity type is listed first and underlined. All nonkey attributes are listed after the primary key. This relation is shown as a table with sample data in Figure 9-10B.

A

FIGURE 9-10
Transforming an entity type to a relation: (A) E-R diagram, (B) Relation.

B

CUSTOMER

Customer_ID	Name	Address	City_State_Zip	Discount
1273	Contemporary Designs	123 Oak St.	Austin, TX 28384	5%
6390	Casual Corner	18 Hoosier Dr.	Bloomington, IN 45821	3%

Represent Relationships

The procedure for representing relationships depends on both the degree of the relationship—unary, binary, ternary—and the cardinalities of the relationship.

Binary 1:N and 1:1 Relationships A binary one-to-many (1:*N*) relationship in an E-R diagram is represented by adding the primary key attribute (or attributes) of the entity on the one side of the relationship as a foreign key in the relation that is on the many side of the relationship.

Figure 9-11A, an example of this rule, shows the Places relationship (1:*N*) linking CUSTOMER and ORDER at Pine Valley Furniture. Two relations, CUSTOMER and ORDER, were formed from the respective entity types (see Figure 9-11B). Customer_ID, which is the primary key of CUSTOMER (on the one side of the relationship) is added as a foreign key to ORDER (on the many side of the relationship).

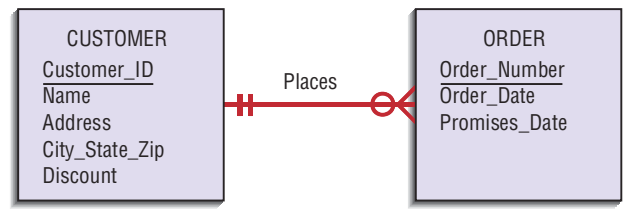
One special case under this rule was mentioned in the previous section. If the entity on the many side needs the key of the entity on the one side as part of its primary key (this is a so-called weak entity), then this attribute is added not as a nonkey but as part of the primary key.

For a binary or unary one-to-one (1:1) relationship between the two entities A and B (for a unary relationship, A and B would be the same entity type), the relationship can be represented by any of the following choices:

1. Adding the primary key of A as a foreign key of B
2. Adding the primary key of B as a foreign key of A
3. Both of the above

Binary and Higher-Degree M:N Relationships Suppose that a binary many-to-many (*M:N*) relationship (or associative entity) exists between two entity types A and B. For such a relationship, we create a separate relation C. The primary key of this relation is a composite key consisting of the primary key for each of the two entities in the relationship.

FIGURE 9-11
Representing a 1:*N* relationship: (A) E-R diagram, (B) Relations.



A

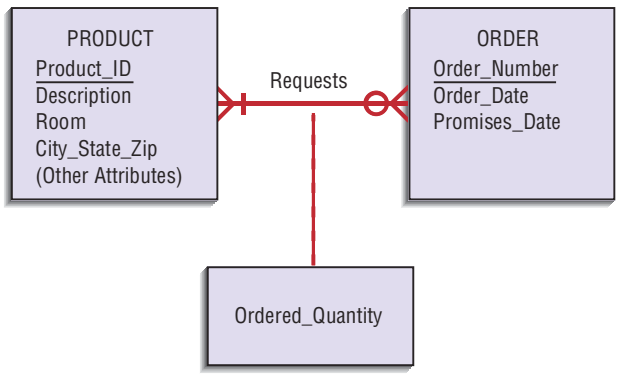
CUSTOMER

Customer_ID	Name	Address	City_State_ZIP	Discount
1273	Contemporary Designs	123 Oak St.	Austin, TX 28384	5%
6390	Casual Corner	18 Hoosier Dr.	Bloomington, IN 45821	3%

ORDER

Order_Number	Order_Date	Promised_Date	Customer_ID
57194	3/15/12	3/28/12	6390
63725	3/17/12	4/01/12	1273
80149	3/14/12	3/24/12	6390

B



A

ORDER

Order_Number	Order_Date	Promised_Date
61384	2/17/2012	3/01/2012
62009	2/13/2012	2/27/2012
62807	2/15/2012	3/01/2012

ORDER LINE

Order_Number	Product_ID	Quantity_ Ordered
61384	M128	2
61384	A261	1

PRODUCT

Product_ID	Description	(Other Attributes)
M128	Bookcase	—
A261	Wall unit	—
R149	Cabinet	—

B

FIGURE 9-12
Representing an *M:N* relationship: (A) E-R diagram, (B) Relations.

Any nonkey attributes that are associated with the *M:N* relationship are included with the relation C.

Figure 9-12A, an example of this rule, shows the Requests relationship (*M:N*) between the entity types ORDER and PRODUCT for Pine Valley Furniture. Figure 9-12B shows the three relations (ORDER, ORDER LINE, and PRODUCT) that are formed from the entity types and the Requests relationship. A relation (called ORDER LINE in Figure 9-12B) is created for the Requests relationship. The primary key of ORDER LINE is the combination (Order_Number, Product_ID), which consists of the respective primary keys of ORDER and PRODUCT. The nonkey attribute Quantity_Ordered also appears in ORDER LINE.

Occasionally, the relation created from an *M:N* relationship requires a primary key that includes more than just the primary keys from the two related relations. Consider, for example, the following situation:

In this case, Date must be part of the key for the SHIPMENT relation to uniquely distinguish each row of the SHIPMENT table, as follows:

SHIPMENT(Customer_ID, Vendor_ID, Date, Amount)

If each shipment has a separate nonintelligent key (a system-assigned unique value that has no business meaning; e.g., order number, customer number), say a shipment number, then Date becomes a nonkey and Customer_ID and Vendor_ID become foreign keys, as follows:

SHIPMENT(Shipment_Number, Customer_ID, Vendor_ID, Date, Amount)

In some cases, a relationship may be found among three or more entities. In such cases, we create a separate relation that has as a primary key the composite of the primary keys of each of the participating entities (plus any necessary additional key elements). This rule is a simple generalization of the rule for a binary *M:N* relationship.

Unary Relationships To review, a unary relationship is a relationship between the instances of a single entity type, which are also called *recursive relationships*. Figure 9-13 shows two common examples. Figure 9-13A shows a one-to-many relationship named Manages that associates employees with another employee who is their manager. Figure 9-13B shows a many-to-many relationship that associates certain items with their component items. This relationship is called a *bill-of-materials structure*.

For a unary 1:*N* relationship, the entity type (such as EMPLOYEE) is modeled as a relation. The primary key of that relation is the same as for the entity type. Then a foreign key is added to the relation that references the primary key values. A **recursive foreign key** is a foreign key in a relation that references the primary key values of that same relation. We can represent the relationship in Figure 9-13A as follows:

Recursive foreign key
A foreign key in a relation that references the primary key values of that same relation.

EMPLOYEE(Emp_ID, Name, Birthdate, Manager_ID)

In this relation, Manager_ID is a recursive foreign key that takes its values from the same set of worker identification numbers as Emp_ID.

For a unary *M:N* relationship, we model the entity type as one relation. Then we create a separate relation to represent the *M:N* relationship. The primary key of this new relation is a composite key that consists of two attributes (which need not have the same name) that both take their values from the same primary key. Any attribute associated with the relationship (such as Quantity in Figure 9-13B) is included as a nonkey attribute in this new relation. We can express the result for Figure 9-13B as follows:

ITEM(Item_Number, Name, Cost)
ITEM-BILL(Item_Number, Component_Number, Quantity)

Summary of Transforming E-R Diagrams to Relations

We have now described how to transform E-R diagrams to relations. Table 9-1 lists the rules discussed in this section for transforming entity-relationship diagrams into equivalent relations. After this transformation, you should check the resulting relations to determine whether they are in third normal form and, if necessary, perform normalization as described earlier in the chapter.

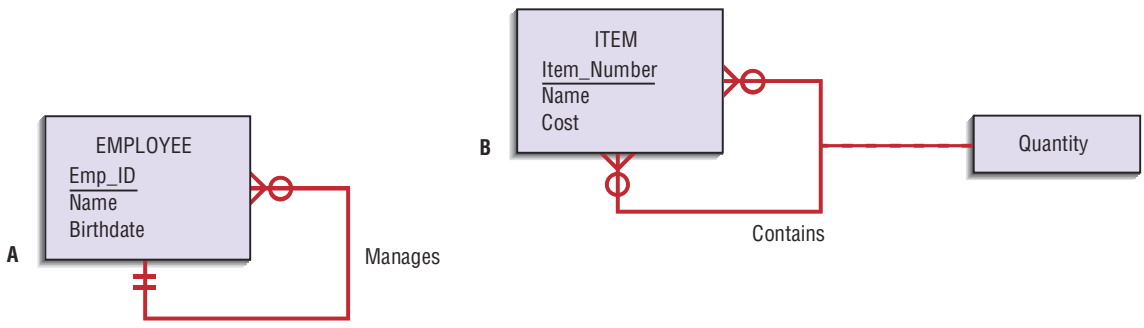


FIGURE 9-13
Two unary relations: (A) EMPLOYEE with manages relationship (1:*N*), (B) Bill-of-materials structure (*M:N*).

TABLE 9-1: E-R to Relational Transformation

E-R Structure	Relational Representation
Regular entity	Create a relation with primary key and nonkey attributes.
Weak entity	Create a relation with a composite primary key (which includes the primary key of the entity on which this weak entity depends) and nonkey attributes.
Binary or unary 1:1 relationship	Place the primary key of either entity in the relation for the other entity or do it for both entities.
Binary 1:*N* relationship	Place the primary key of the entity on the one side of the relationship as a foreign key in the relation for the entity on the many side.
Binary or unary *M*:*N* relationship or associative entity	Create a relation with a composite primary key using the primary keys of the related entities, plus any nonkey attributes of the relationship or associative entity.
Binary or unary *M*:*N* relationship or associative entity with additional key(s)	Create a relation with a composite primary key using the primary keys of the related entities and additional primary key attributes associated with the relationship or associative entity, plus any nonkey attributes of the relationship or associative entity.
Binary or unary *M*:*N* relationship or associative entity with its own key	Create a relation with the primary key associated with the relationship or associative entity, plus any nonkey attributes of the relationship or associative entity and the primary keys of the related entities (as nonkey attributes).

Merging Relations

As part of the logical database design, normalized relations likely have been created from a number of separate E-R diagrams and various user interfaces. Some of the relations may be redundant—they may refer to the same entities. If so, you should merge those relations to remove the redundancy. This section describes merging relations, or *view integration*, which is the last step in logical database design and prior to physical file and database design.

An Example of Merging Relations

Suppose that modeling a user interface or transforming an E-R diagram results in the following 3NF relation:

EMPLOYEE1(Emp_ID, Name, Address, Phone)

Modeling a second user interface might result in the following relation:

EMPLOYEE2(Emp_ID, Name, Address, Jobcode, Number_of_Years)

Because these two relations have the same primary key (Emp_ID) and describe the same entity, they should be merged into one relation. The result of merging the relations is the following relation:

EMPLOYEE(Emp_ID, Name, Address, Phone, Jobcode, Number_of_Years)

Notice that an attribute that appears in both relations (such as Name in this example) appears only once in the merged relation.

View Integration Problems

When integrating relations, you must understand the meaning of the data and must be prepared to resolve any problems that may arise in that process. In this section, we describe and illustrate three problems that arise in view integration: synonyms, homonyms, and dependencies between nonkeys.

Synonyms
Two different names that are used for the same attribute.

Synonyms In some situations, two or more attributes may have different names but the same meaning, as when they describe the same characteristic of an entity. Such attributes are called **synonyms.** For example, Emp_ID and Employee_Number may be synonyms.

When merging the relations that contain synonyms, you should obtain, if possible, agreement from users on a single standardized name for the attribute and eliminate the other synonym. Another alternative is to choose a third name to replace the synonyms. For example, consider the following relations:

STUDENT1(Student_ID, Name)
STUDENT2(Matriculation_Number, Name, Address)

In this case, the analyst recognizes that both the Student_ID and the Matriculation_Number are synonyms for a person's social security number and are identical attributes. One possible resolution would be to standardize on one of the two attribute names, such as Student_ID. Another option is to use a new attribute name, such as SSN, to replace both synonyms. Assuming the latter approach, merging the two relations would produce the following result:

STUDENT(SSN, Name, Address)

Homonym
A single attribute name that is used for two or more different attributes.

Homonyms In other situations, a single attribute name, called a **homonym,** may have more than one meaning or describe more than one characteristic. For example, the term *account* might refer to a bank's checking account, savings account, loan account, or other type of account; therefore, *account* refers to different data, depending on how it is used.

You should be on the lookout for homonyms when merging relations. Consider the following example:

STUDENT1(Student_ID, Name, Address)
STUDENT2(Student_ID, Name, Phone_Number, Address)

In discussions with users, the systems analyst may discover that the attribute Address in STUDENT1 refers to a student's campus address, whereas in STUDENT2 the same attribute refers to a student's home address. To resolve this conflict, we would probably need to create new attribute names and the merged relation would become:

STUDENT(Student_ID, Name, Phone_Number, Campus_Address,
Permanent_Address)

Dependencies between Nonkeys When two 3NF relations are merged to form a single relation, dependencies between nonkeys may result. For example, consider the following two relations:

STUDENT1(Student_ID, Major)
STUDENT2(Student_ID, Adviser)

Because STUDENT1 and STUDENT2 have the same primary key, the two relations may be merged:

STUDENT(Student_ID, Major, Adviser)

However, suppose that each major has exactly one adviser. In this case, Adviser is functionally dependent on Major:

Major→Adviser

If the previous dependency exists, then STUDENT is in 2NF but not 3NF, because it contains a functional dependency between nonkeys. The analyst can create 3NF relations by creating two relations with Major as a foreign key in STUDENT:

STUDENT(Student_ID, Major)
MAJOR ADVISER(Major, Adviser)

Logical Database Design for Hoosier Burger

In Chapter 7 we developed an E-R diagram for a new inventory control system at Hoosier Burger (Figure 9-14 repeats the diagram from Chapter 7). In this section we show how this E-R model is translated into normalized relations and how to normalize and then merge the relations for a new report with the relations from the E-R model.

In this E-R model, four entities exist independently of other entities: SALE, PRODUCT, INVOICE, and INVENTORY ITEM. Given the attributes shown in Figure 9-14, we can represent these entities in the following four relations:

SALE(Receipt_Number, Sale_Date)
PRODUCT(Product_ID, Product_Description)
INVOICE(Vendor_ID, Invoice_Number, Invoice_Date, Paid?)
INVENTORY ITEM(Item_Number, Item_Description, Quantity_in_Stock, Minimum_Order_Quantity, Type_of_Item)

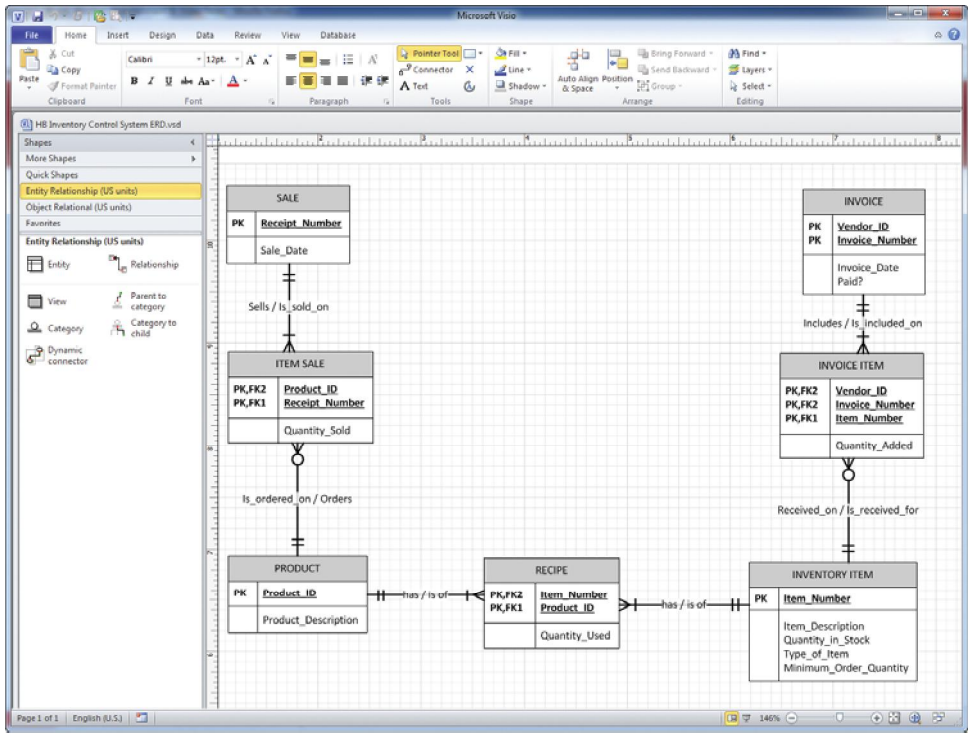


FIGURE 9-14
Final E-R diagram for Hoosier Burger's inventory control system.

The entities ITEM SALE and INVOICE ITEM as well as the associative entity RECIPE each have composite primary keys taken from the entities to which they relate, so we can represent these three entities in the following three relations:

ITEM SALE(Receipt_Number, Product_ID, Quantity_Sold)
INVOICE ITEM(Vendor_ID, Invoice_Number, Item_Number, Quantity_ Added)
RECIPE(Product_ID, Item_Number, Quantity_Used)

Because no many-to-many, one-to-one, or unary relationships are involved, we have now represented all the entities and relationships from the E-R model. Also, each of the previous relations is in 3NF because all attributes are simple, all nonkeys are fully dependent on the whole key, and there are no dependencies between nonkeys in the INVOICE and INVENTORY ITEM relations.

Now suppose that Bob Mellankamp wanted an additional report that was not previously known by the analyst who designed the inventory control system for Hoosier Burger. A rough sketch of this new report, listing volume of purchases from each vendor by type of item in a given month, appears in Figure 9-15. In this report, the same type of item may appear many times if multiple vendors supply the same type of item.

This report contains data about several relations already known to the analyst, including:

INVOICE(Vendor_ID, Invoice_Number, Invoice_Date): primary keys and the date are needed to select invoices in the specified month of the report
INVENTORY ITEM(Item_Number, Type_of_Item): primary key and a nonkey in the report
INVOICE ITEM(Vendor_ID, Invoice_Number, Item_Number, Quantity_Added): primary keys and the raw quantity of items invoiced that are subtotaled by vendor and type of item in the report

In addition, the report includes a new attribute: Vendor_Name. After some investigation, an analyst determines that Vendor_ID→Vendor_Name. Because the whole primary key of the INVOICE relation is Vendor_ID and Invoice_Number, if Vendor_Name were part of the INVOICE relation, this relation would violate the 3NF rule. So, a new VENDOR relation must be created as follows:

VENDOR(Vendor_ID, Vendor_Name)

Now, Vendor_ID not only is part of the primary key of INVOICE but also is a foreign key referencing the VENDOR relation. Hence, a one-to-many relationship from VENDOR to INVOICE is necessary. The systems analyst determines that an

FIGURE 9-15
Hoosier Burger monthly vendor load report.

Monthly Vendor Load Report
for Month: xxxxx

Page *x* of *n*

Vendor			
ID	Name	Type of Item	Total Quantity Added
V1	V1name	aaa	nnn1
		bbb	nnn2
		ccc	nnn3
V2	V2name	bbb	nnn4
		mmm	nnn5

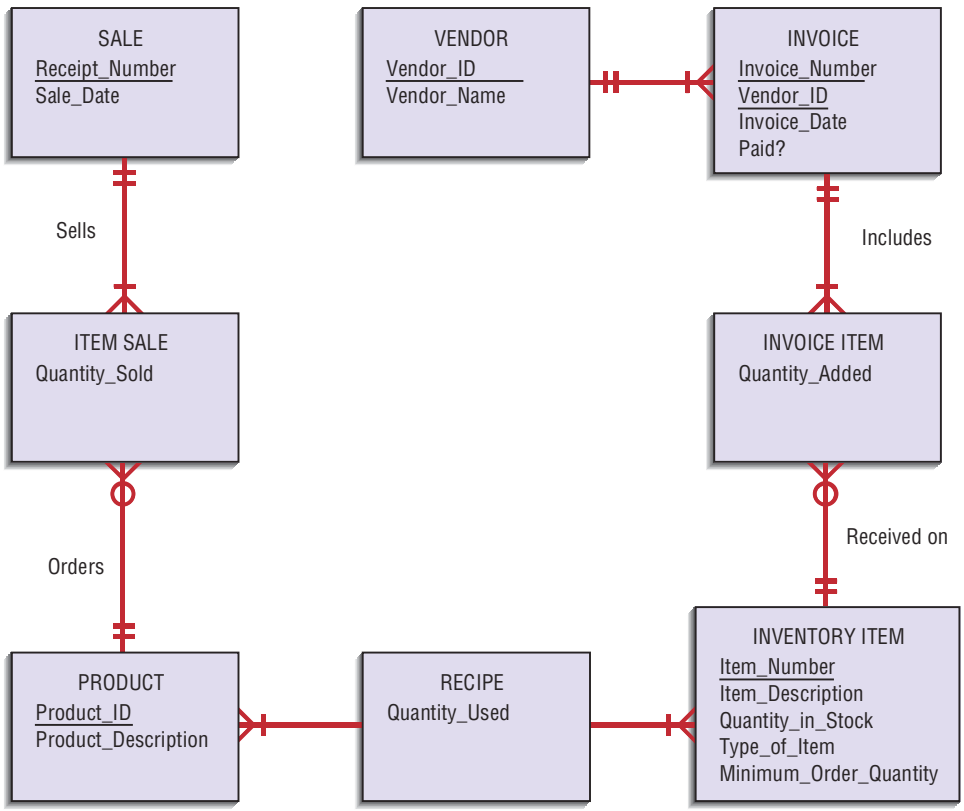


FIGURE 9-16
E-R diagram corresponding to normalized relations of Hoosier Burger's inventory control system.

invoice must come from a vendor, and keeping data about a vendor is not necessary unless the vendor invoices Hoosier Burger. An updated E-R diagram, reflecting these enhancements for new data needed in the monthly vendor load report, appears in Figure 9-16. The normalized relations for this database are:

SALE(Receipt_Number, Sale_Date)
PRODUCT(Product_ID, Product_Description)
INVOICE(Vendor_ID, Invoice_Number, Invoice_Date, Paid?)
INVENTORY ITEM(Item_Number, Item_Description, Quantity_in_Stock, Type_of_Item, Minimum_Order_Quantity)
ITEM SALE(Receipt_Number, Product_ID, Quantity_Sold)
INVOICE ITEM(Vendor_ID, Invoice_Number, Item_Number, Quantity_Added)
RECIPE(Product_ID, Item_Number, Quantity_Used)
VENDOR(Vendor_ID, Vendor_Name)

Physical File and Database Design

Designing physical files and databases requires certain information that should have been collected and produced during prior SDLC phases. This information includes:

- Normalized relations, including volume estimates
- Definitions of each attribute
- Descriptions of where and when data are used: entered, retrieved, deleted, and updated (including frequencies)
- Expectations or requirements for response time and data integrity
- Descriptions of the technologies used for implementing the files and database so that the range of required decisions and choices for each is known

Normalized relations are, of course, the result of logical database design. Statistics on the number of rows in each table, as well as the other information listed may have been collected during requirements determination in systems analysis. If not, these items need to be discovered to proceed with database design.

We take a bottom-up approach to reviewing physical file and database design. Thus, we begin the physical design phase by addressing the design of physical fields for each attribute in a logical data model.

Designing Fields

Field
The smallest unit of named application data recognized by system software.

A **field** is the smallest unit of application data recognized by system software, such as a programming language or database management system. An attribute from a logical database model may be represented by several fields. For example, a student name attribute in a normalized student relation might be represented as three fields: last name, first name, and middle initial. Each field requires a separate definition when the application system is implemented.

In general, you will represent each attribute from each normalized relation as one or more fields. The basic decisions you must make in specifying each field concern the type of data (or storage type) used to represent the field and data integrity controls for the field.

Choosing Data Types

Data type
A coding scheme recognized by system software for representing organizational data.

A **data type** is a coding scheme recognized by system software for representing organizational data. The bit pattern of the coding scheme is usually immaterial to you, but the space to store data and the speed required to access data are of consequence in the physical file and database design. The specific file or database management software you use with your system will dictate which choices are available to you. For example, Table 9-2 lists the data types available in Microsoft Access.

Selecting a data type balances four objectives that will vary in degree of importance for different applications:

1. Minimize storage space
2. Represent all possible values of the field
3. Improve data integrity for the field
4. Support all data manipulations desired on the field

You want to choose a data type for a field that minimizes space, represents every possible legitimate value for the associated attribute, and allows the data to be manipulated as needed. For example, suppose a "quantity sold" field can be represented by a Number data type. You would select a length for this field that would handle the maximum value, plus some room for growth of the business. Further, the Number data type will restrict users from entering inappropriate values (text), but it does allow negative numbers (if this is a problem, application code or form design may be required to restrict the values to positive).

Be careful—the data type must be suitable for the life of the application; otherwise, maintenance will be required. Choose data types for future needs by anticipating growth. Also, be careful that date arithmetic can be done so that dates can be subtracted or time periods can be added to or subtracted from a date.

Several other capabilities of data types may be available with some database technologies. We discuss a few of the most common of these features next: calculated fields and coding and compression techniques.

TABLE 9-2: Microsoft Access Data Types

Data Type	Description
Text	Text or combinations of text and numbers, as well as numbers that don't require calculations, such as phone numbers. A specific length is indicated, with a maximum number of characters of 255. One byte of storage is required for each character used.
Memo	Lengthy (up to 65,535 characters) text or combinations of text and numbers. One byte of storage is required for each character used.
Number	Numeric data used in mathematical calculations. Either 1, 2, 4, or 8 bytes of storage space is required, depending on the specified length of the number.
Date/Time	Date and time values for the years 100 through 9999. Eight bytes of storage space is required.
Currency	Currency values and numeric data used in mathematical calculations involving data with one to four decimal places. Accurate to 15 digits on the left side of the decimal separator and to 4 digits on the right side. Eight bytes of storage space is required.
Autonumber	A unique sequential (incremented by 1) number or random number assigned by Microsoft Access whenever a new record is added to a table. Typically, 4 bytes of storage is required.
Yes/No	Yes and No values and fields that contain only one of two values (Yes/No, True/False, or On/Off). One bit of storage is required.
OLE Object	An object (such as a Microsoft Excel spreadsheet, a Microsoft Word document, graphics, sounds, or other binary data) linked to or embedded in a Microsoft Access table. Up to 1 gigabyte of storage possible.
Hyperlink	Text or combinations of text and numbers stored as text and used as a hyperlink address (typical URL form).
Lookup Wizard	Creates a field that allows you to choose a value from another table (the table's primary key) or from a list of values by using a list box or combo box. Clicking this option starts the Lookup Wizard, which creates a Lookup field. After you complete the wizard, Microsoft Access sets the data type based on the values selected in the wizard. Used for foreign keys to enforce referential integrity. Space requirement depends on length of foreign key or lookup value.

Calculated Fields It is common that an attribute is mathematically related to other data. For example, an invoice may include a "total due" field, which represents the sum of the amount due on each item on the invoice. A field that can be derived from other database fields is called a **calculated** (or **computed** or **derived**) **field** (recall that a functional dependency between attributes does not imply a calculated field). Some database technologies allow you to explicitly define calculated fields along with other raw data fields. If you specify a field as calculated, you would then usually be prompted to enter the formula for the calculation; the formula can involve other fields from the same record and possibly fields from records in related files. The database technology will either store the calculated value or compute it when requested.

Calculated (or computed or derived) field
A field that can be derived from other database fields.

Coding and Compression Techniques Some attributes have few values from a large range of possible values. For example, although a six-digit field (five numbers plus a value sign) can represent numbers −99999 to 99999, maybe only 100 positive values within this range will ever exist. Thus, a Number data type does not adequately restrict the permissible values for data integrity, and storage space for five digits plus a value sign is wasteful. To use space more efficiently (and less space may mean faster access because the data you need are closer together), you can define a field for an attribute so that the possible attribute values are not represented literally but rather are abbreviated. For example, suppose in Pine Valley Furniture each product has a finish attribute, with possible values Birch, Walnut, Oak, and so forth. To store this attribute as Text might require 12, 15, or even 20 bytes to represent the longest finish value.

Suppose that even a liberal estimate is that Pine Valley Furniture will never have more than twenty-five finishes. Thus, a single alphabetic or alphanumeric character would be more than sufficient. We not only reduce storage space but also increase integrity (by restricting input to only a few values), which helps to achieve two of the physical file and database design goals. Codes also have disadvantages. If used in system inputs and outputs, they can be more difficult for users to remember, and programs must be written to decode fields if codes will not be displayed.

Controlling Data Integrity

We have already explained that data typing helps control data integrity by limiting the possible range of values for a field. You can use additional physical file and database design options to ensure higher-quality data. Although these controls can be imposed within application programs, it is better to include these as part of the file and database definitions so that the controls are guaranteed to be applied all the time, as well as uniformly for all programs. The five popular data integrity control methods are default value, input mask, range control, referential integrity, and null value control.

Default value
The value a field will assume unless an explicit value is entered for that field.

- *Default value:* A **default value** is the value a field will assume unless an explicit value is entered for the field. For example, the city and state of most customers for a particular retail store will likely be the same as the store's city and state. Assigning a default value to a field can reduce data-entry time (the field can simply be skipped during data entry) and data-entry errors, such as typing *IM* instead of *IN* for *Indiana*.

Input mask
A pattern of codes that restricts the width and possible values for each position of a field.

- *Input mask:* Some data must follow a specified pattern. An **input mask** (or field template) is a pattern of codes that restricts the width and possible values for each position within a field. For example, a product number at Pine Valley Furniture is four alphanumeric characters—the first is alphabetic and the next three are numeric—defined by an input mask of L999, where L means that only alphabetic characters are accepted, and 9 means that only numeric digits are accepted. M128 is an acceptable value, but 3128 or M12H would be unacceptable. Other types of input masks can be used to convert all characters to uppercase, indicate how to show negative numbers, suppress showing leading zeros, or indicate whether entry of a letter or digit is optional.
- *Range control:* Both numeric and alphabetic data may have a limited set of permissible values. For example, a field for the number of product units sold may have a lower bound of 0, and a field that represents the month of a product sale may be limited to the values JAN, FEB, and so forth.
- *Referential integrity:* As noted earlier in this chapter, the most common example of referential integrity is cross-referencing between relations. For example, consider the pair of relations in Figure 9-17A. In this case, the values for the foreign key Customer_ID field within a customer order must be limited to the set of Customer_ID values from the customer relation; we would not want to accept an order for a nonexisting or unknown customer. Referential integrity may be useful in other instances. Consider the employee relation example in Figure 9-17B. In this example, the employee relation has a field of Supervisor_ID. This field refers to the Employee_ID of the employee's supervisor and should have referential integrity on the Employee_ID field within the same relation. Note in this case that because some employees do not

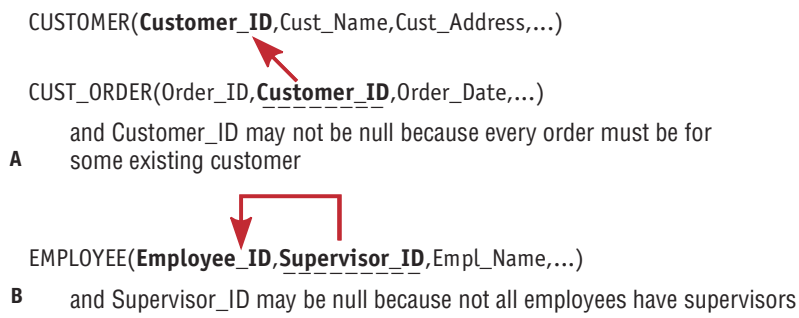


FIGURE 9-17
Examples of referential integrity field controls: (A) Referential integrity between relations, (B) Referential integrity within a relation.

have supervisors, this referential integrity constraint is weak because the value of a Supervisor_ID field may be empty.

- *Null value control:* A **null value** is a special field value, distinct from 0, blank, or any other value, that indicates that the value for the field is missing or otherwise unknown. It is not uncommon that when it is time to enter data—for example, a new customer—you might not know the customer's phone number. The question is whether a customer, to be valid, must have a value for this field. The answer for this field is probably initially no, because most data processing can continue without knowing the customer's phone number. Later, a null value may not be allowed when you are ready to ship product to the customer. On the other hand, you must always know a value for the Customer_ID field. Because of referential integrity, you cannot enter any customer orders for this new customer without knowing an existing Customer_ID value, and the customer's name is essential for visual verification of correct data entry. Besides using a special null value when a field is missing its value, you can also estimate the value, produce a report indicating rows of tables with critical missing values, or determine whether the missing value matters in computing needed information.

Null value
A special field value, distinct from 0, blank, or any other value, that indicates that the value for the field is missing or otherwise unknown.

Designing Physical Tables

A relational database is a set of related tables (tables are related by foreign keys referencing primary keys). In logical database design, you grouped into a relation those attributes that concern some unifying, normalized business concept, such as a customer, product, or employee. In contrast, a **physical table** is a named set of rows and columns that specifies the fields in each row of the table. A physical table may or may not correspond to one relation. Whereas normalized relations possess properties of well-structured relations, the design of a physical table has two goals different from those of normalization: efficient use of secondary storage and data-processing speed.

Physical table
A named set of rows and columns that specifies the fields in each row of the table.

The efficient use of secondary storage (disk space) relates to how data are loaded on disks. Disks are physically divided into units (called *pages*) that can be read or written in one machine operation. Space is used efficiently when the physical length of a table row divides close to evenly into the length of the storage unit. For many information systems, this even division is difficult to achieve because it depends on factors, such as operating system parameters, outside the control of each database. Consequently, we do not discuss this factor of physical table design in this text.

A second and often more important consideration when selecting a physical table design is efficient data processing. Data are most efficiently processed

Denormalization
The process of splitting or combining normalized relations into physical tables based on affinity of use of rows and fields.

when they are stored close to one another in secondary memory, thus minimizing the number of input/output (I/O) operations that must be performed. Typically, the data in one physical table (all the rows and fields in those rows) are stored close together on disk. **Denormalization** is the process of splitting or combining normalized relations into physical tables based on affinity of use of rows and fields. Consider Figure 9-18. In Figure 9-18A, a normalized product relation is split into separate physical tables with each containing only engineering, accounting, or marketing product data; the primary key must be included in each table. Note, the Description and Color attributes are repeated in both the engineering and marketing tables because these attributes relate to both kinds of data. In Figure 9-18B, a customer relation is denormalized by putting rows from different geographic regions into separate tables. In both cases, the goal is to create tables that contain only the data used together in programs. By placing data used together close to one another on disk, the number of disk I/O operations needed to retrieve all the data needed in a program is minimized.

Denormalization can increase the chance of errors and inconsistencies that normalization avoided. Further, denormalization optimizes certain data processing at the expense of others, so if the frequencies of different processing activities change, the benefits of denormalization may no longer exist.

FIGURE 9-18
Examples of denormalization: (A) Denormalization by columns, (B) Denormalization by rows.

A

Normalized Product Relation

PRODUCT(Product_ID,Description,Drawing_Number,Weight,Color,Unit_Cost,Burden_Rate,Price,Product_Manager)

Denormalized Functional Area Product Relations for Tables

Engineering: E_PRODUCT(Product_ID,Description,Drawing_Number,Weight,Color)
Accounting: A_PRODUCT(Product_ID,Unit_Cost,Burden_Rate)
Marketing: M_PRODUCT(Product_ID,Description,Color,Price,Product_Manager)

B

Normalized Customer Table

CUSTOMER

Customer_ID	Name	Region	Annual_Sales
1256	Rogers	Atlantic	10,000
1323	Temple	Pacific	20,000
1455	Gates	South	15,000
1626	Hope	Pacific	22,000
2433	Bates	South	14,000
2566	Bailey	Atlantic	12,000

Denormalized Regional Customer Tables

A_CUSTOMER

Customer_ID	Name	Region	Annual_Sales
1256	Rogers	Atlantic	10,000
2566	Bailey	Atlantic	12,000

P_CUSTOMER

Customer_ID	Name	Region	Annual_Sales
1323	Temple	Pacific	20,000
1626	Hope	Pacific	22,000

S_CUSTOMER

Customer_ID	Name	Region	Annual_Sales
1455	Gates	South	15,000
2433	Bates	South	14,000

Various forms of denormalization can be done, but no hard-and-fast rules will help you decide when to denormalize data. Here are three common situations in which denormalization often makes sense (see Figure 9-19 for illustrations):

1. *Two entities with a one-to-one relationship.* Figure 9-19A shows student data with optional data from a standard scholarship application a student may complete. In this case, one record could be formed with four fields from the STUDENT and SCHOLARSHIP APPLICATION FORM normalized relations. (*Note:* In this case, fields from the optional entity must have null values allowed.)
2. *A many-to-many relationship (associative entity) with nonkey attributes.* Figure 9-19B shows price quotes for different items from different vendors. In this case, fields from ITEM and PRICE QUOTE relations might be combined into one physical table to avoid having to combine all three tables together. (*Note:* It may create considerable duplication of data—in the example, the ITEM fields, such as Description, would repeat for each price quote—and excessive updating if duplicated data changes.)
3. *Reference data.* Figure 9-19C shows that several ITEMs have the same STORAGE INSTRUCTIONS, and STORAGE INSTRUCTIONS relate only to ITEMs. In this case, the storage instruction data could be stored in the ITEM table, thus reducing the number of tables to access but also creating redundancy and the potential for extra data maintenance.

Arranging Table Rows

The result of denormalization is the definition of one or more physical files. A computer operating system stores data in a **physical file,** which is a named set of table rows stored in a contiguous section of secondary memory. A file contains rows and columns from one or more tables, as produced from denormalization. To the operating system—like Windows, Linux, or Mac OS—each table may be one file or the whole database may be in one file, depending on how the database technology and database designer organize data. The way the operating system arranges table rows in a file is called a **file organization.** With some database technologies, the systems designer can choose among several organizations for a file.

Physical file
A named set of table rows stored in a contiguous section of secondary memory.

File organization
A technique for physically arranging the records of a file.

If the database designer has a choice, he or she chooses a file organization for a specific file to provide:

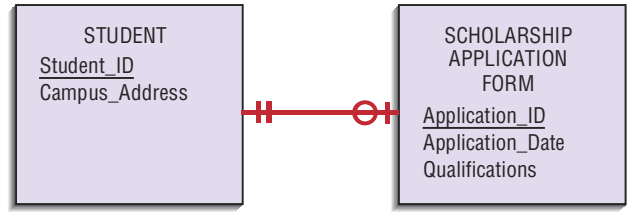
1. Fast data retrieval
2. High throughput for processing transactions
3. Efficient use of storage space
4. Protection from failures or data loss
5. Minimal need for reorganization
6. Accommodation of growth
7. Security from unauthorized use

Often these objectives conflict, and you must select an organization for each file that provides a reasonable balance among the criteria within the resources available.

To achieve these objectives, many file organizations utilize the concept of a pointer. A **pointer** is a field of data that can be used to locate a related field or row of data. In most cases, a pointer contains the address of the associated data, which has no business meaning. Pointers are used in file organizations when it is not possible to store related data next to each other. Because such situations are often the case, pointers are common. In most cases, fortunately, pointers are hidden from a programmer. Yet, because a database designer may need to decide whether and how to use pointers, we introduce the concept here.

Pointer
A field of data that can be used to locate a related field or row of data.

FIGURE 9-19
Possible denormalization situations: (A) Two entities with a one-to-one relationship, (B) A many-to-many relationship with nonkey attributes, (C) Reference data.



Normalized relations:

STUDENT(Student_ID, Campus_Address, Application_ID)

APPLICATION(Application_ID, Application_Date, Qualifications, Student_ID)

Denormalized relation:

STUDENT(Student_ID, Campus_Address, Application_Date, Qualifications) and Application_Date and Qualifications may be null

(*Note:* We assume Application_ID is not necessary when all fields are stored in one record, but this field can be included if it is required application data.)

A

Normalized relations:

VENDOR(Vendor_ID, Address, Contact_Name)

ITEM(Item_ID, Description)

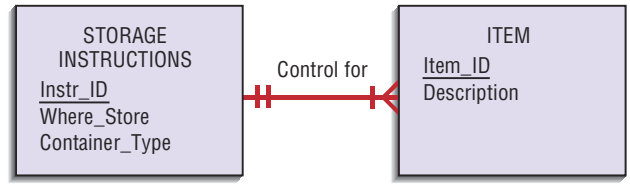
PRICE QUOTE(Vendor_ID, Item_ID, Price)

Denormalized relations:

VENDOR(Vendor_ID, Address, Contact_Name)

ITEM-QUOTE(Vendor_ID, Item_ID, Description, Price)

B



Normalized relations:

STORAGE(Instr_ID, Where_Store, Container_Type)

ITEM(Item_ID, Description, Instr_ID)

Denormalized relation:

ITEM(Item_ID, Description, Where_Store, Container_Type)

C

Literally hundreds of different file organizations and variations have been created, but we outline the basics of three families of file organizations used in most file management environments: sequential, indexed, and hashed, as illustrated in Figure 9-20. You need to understand the particular variations of each method available in the environment for which you are designing files.

Sequential File Organizations In a **sequential file organization,** the rows in the file are stored in sequence according to a primary key value (see Figure 9-20A). To locate a particular row, a program must normally scan the file from the beginning until the desired row is located. A common example of a sequential file is the alphabetical list of persons in the white pages of a phone directory (ignoring any index that may be included with the directory). Sequential files are fast if you want to process rows sequentially, but they are essentially impractical for random row retrievals. Deleting rows can cause wasted space or the need to compress the file. Adding rows requires rewriting the file, at least from the point of insertion. Updating a row may also require rewriting the file, unless the file organization supports rewriting over the updated row only. Moreover, only one sequence can be maintained without duplicating the rows.

Sequential file organization
The rows in the file are stored in sequence according to a primary key value.

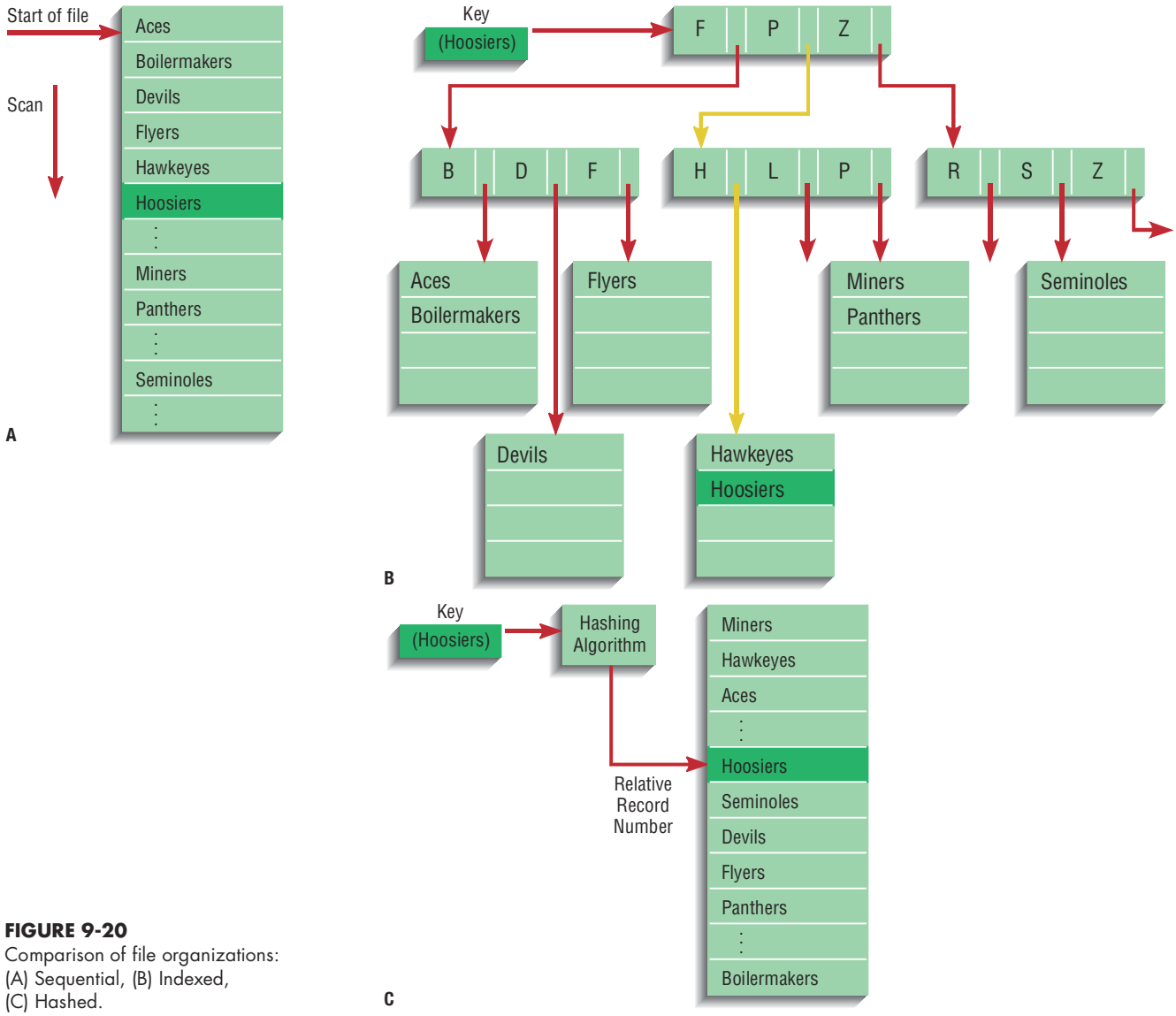


FIGURE 9-20
Comparison of file organizations: (A) Sequential, (B) Indexed, (C) Hashed.

Indexed file organization
The rows are stored either sequentially or nonsequentially, and an index is created that allows software to locate individual rows.

Index
A table used to determine the location of rows in a file that satisfy some condition.

Secondary key
One or a combination of fields for which more than one row may have the same combination of values.

Indexed File Organizations In an **indexed file organization,** the rows are stored either sequentially or nonsequentially, and an index is created that allows the application software to locate individual rows (see Figure 9-20B). Like a card catalog in a library, an **index** is a structure that is used to determine the rows in a file that satisfy some condition. Each entry matches a key value with one or more rows. An index can point to unique rows (a primary key index, such as on the Product_ID field of a product table) or to potentially more than one row. An index that allows each entry to point to more than one record is called a **secondary key** index. Secondary key indexes are important for supporting many reporting requirements and for providing rapid ad hoc data retrieval. An example would be an index on the Finish field of a product table.

The example in Figure 9-20B, typical of many index structures, illustrates that indexes can be built on top of indexes, creating a hierarchical set of indexes, and the data are stored sequentially in many contiguous segments. For example, to find the record with key "Hoosiers," the file organization would start at the top index and take the pointer after the entry P, which points to another index for all keys that begin with the letters *G* through *P* in the alphabet. Then the software would follow the pointer after the H in this index, which represents all those records with keys that begin with the letters *G* through *H*. Eventually, the search through the indexes either locates the desired record or indicates that no such record exists. The reason for storing the data in many contiguous segments is to allow room for some new data to be inserted in sequence without rearranging all the data.

The main disadvantages of indexed file organizations are the extra space required to store the indexes and the extra time necessary to access and maintain indexes. Usually these disadvantages are more than offset by the advantages. Because the index is kept in sequential order, both random and sequential processing are practical. Also, because the index is separate from the data, you can build multiple index structures on the same data file (just as in the library where there are multiple indexes on author, title, subject, and so forth). With multiple indexes, software may rapidly find records that have compound conditions, such as finding books by Tom Clancy on espionage.

The decision of which indexes to create is probably the most important physical database design task for relational database technology, such as Microsoft Access, SQL Server, Oracle, DB2, and similar systems. Indexes can be created for both primary and secondary keys. When using indexes, there is a trade-off between improved performance for retrievals and degrading performance for inserting, deleting, and updating the rows in a file. Thus, indexes should be used generously for databases intended primarily to support data retrievals, such as for decision support applications. Because they impose additional overhead, indexes should be used judiciously for databases that support transaction processing and other applications with heavy updating requirements.

Here are some rules for choosing indexes for relational databases:

1. Specify a unique index for the primary key of each table (file). This selection ensures the uniqueness of primary key values and speeds retrieval based on those values. Random retrieval based on primary key value is common for answering multitable queries and for simple data-maintenance tasks.
2. Specify an index for foreign keys. As in the first guideline, this speeds processing multitable queries.
3. Specify an index for nonkey fields that are referenced in qualification and sorting commands for the purpose of retrieving data.

To illustrate the use of these rules, consider the following relations for Pine Valley Furniture:

PRODUCT(Product_Number, Description, Finish, Room, Price)
ORDER(Order_Number, Product_Number, Quantity)

You would normally specify a unique index for each primary key: Product_Number in PRODUCT and Order_Number in ORDER. Other indexes would be assigned based on how the data are used. For example, suppose that a system module requires PRODUCT and PRODUCT_ORDER data for products with a price below $500, ordered by Product_Number. To speed up this retrieval, you could consider specifying indexes on the following nonkey attributes:

1. Price in PRODUCT because it satisfies rule 3
2. Product_Number in ORDER because it satisfies rule 2

Because users may direct a potentially large number of different queries against the database, especially for a system with a lot of ad hoc queries, you will probably have to be selective in specifying indexes to support the most common or frequently used queries.

Hashed File Organizations In **hashed file organization,** the address of each row is determined using an algorithm (see Figure 9-20C) that converts a primary key value into a row address. Although there are several variations of hashed files, in most cases the rows are located nonsequentially as dictated by the hashing algorithm. Thus, sequential data processing is impractical. On the other hand, retrieval of random rows is fast. Some of the issues in the design of hashing file organizations, such as how to handle two primary keys that translate into the same address, are beyond our scope (see Hoffer, Ramesh, and Topi [2011] for a thorough discussion).

Hashed file organization
The address for each row is determined using an algorithm.

Summary of File Organizations The three families of file organizations—sequential, indexed, and hashed—cover most of the file organizations you will have at your disposal as you design physical files and databases. Table 9-3 summarizes the comparative features of these file organizations. You can use this table to help choose a file organization by matching the file characteristics and file processing requirements with the features of the file organization.

Designing Controls for Files

Two of the goals of physical table design mentioned earlier are protection from failures or data loss and security from unauthorized use. These goals are achieved primarily by implementing controls on each file. Data integrity controls, a primary type of control, was mentioned earlier in the chapter. Two other important types of controls address file backup and security.

It is almost inevitable that a file will be damaged or lost, because of either software or human errors. When a file is damaged, it must be restored to an accurate and reasonably current condition. A file and database designer has several techniques for file restoration, including:

- Periodically making a backup copy of a file
- Storing a copy of each change to a file in a transaction log or audit trail
- Storing a copy of each row before or after it is changed

For example, a backup copy of a file and a log of rows after they were changed can be used to reconstruct a file from a previous state (the backup copy) to its current values. This process would be necessary if the current file were so damaged that it could not be used. If the current file is operational but inaccurate,

TABLE 9-3: Comparative Features of Sequential, Indexed, and Hashed File Organizations

	File Organization		
Factor	**Sequential**	**Indexed**	**Hashed**
Storage space	No wasted space	No wasted space for data, but extra space for index	Extra space may be needed to allow for addition and deletion of records
Sequential retrieval on primary key	Very fast	Moderately fast	Impractical
Random retrieval on primary key	Impractical	Moderately fast	Very fast
Multiple key retrieval	Possible, but requires scanning whole file	Very fast with multiple indexes	Not possible
Deleting rows	Can create wasted space or require reorganizing	If space can be dynamically allocated, this is easy, but requires maintenance of indexes	Very easy
Adding rows	Requires rewriting file	If space can be dynamically allocated, this is easy, but requires maintenance of indexes	Very easy, except multiple keys with same address require extra work
Updating rows	Usually requires rewriting file	Easy, but requires maintenance of indexes	Very easy

then a log of earlier images of rows can be used in reverse order to restore a file to an accurate but previous condition. Then a log of the transactions can be re-applied to the restored file to bring it up to current values. It is important that the information system designer make provisions for backup, audit trail, and row image files so that data files can be rebuilt when errors and damage occur.

An information system designer can build data security into a file by several means, including:

- Coding, or encrypting, the data in the file so that they cannot be read unless the reader knows how to decrypt the stored values
- Requiring data file users to identify themselves by entering user names and passwords, and then possibly allowing only certain file activities (read, add, delete, change) for selected users for selected data in the file
- Prohibiting users from directly manipulating any data in the file, and rather requiring programs and users to work with a copy (real or virtual) of the data they need; the copy contains only the data that users or programs are allowed to manipulate, and the original version of the data will change only after changes to the copy are thoroughly checked for validity

Security procedures such as these all add overhead to an information system, so only necessary controls should be included.

Physical Database Design for Hoosier Burger

A set of normalized relations and an associated E-R diagram for Hoosier Burger (Figure 9-16) were presented in the section "Logical Database Design for Hoosier Burger" earlier in this chapter. The display of a complete design of this database

would require more documentation than space permits in this text, so we illustrate in this section only a few key decisions from the complete physical database.

As outlined in this chapter, to translate a logical database design into a physical database design, you need to make the following decisions:

- Create one or more fields for each attribute and determine a data type for each field.
- For each field, decide whether it is calculated, needs to be coded or compressed, must have a default value or input mask, or must have range, referential integrity, or null value controls.
- For each relation, decide whether it should be denormalized to achieve desired processing efficiencies.
- Choose a file organization for each physical file.
- Select suitable controls for each file and the database.

Remember, the specifications for these decisions are made in physical database design, and then the specifications are coded in the implementation phase using the capabilities of the chosen database technology. These database technology capabilities determine what physical database design decisions you need to make. For example, for Microsoft Access, which we assume is the implementation environment for this illustration, the only choice for file organization is indexed, so the file organization decision becomes on which primary and secondary key attributes to build indexes.

We illustrate these physical database design decisions only for the INVOICE table. The first decision most likely would be whether to denormalize this table. Based on the suggestions for possible denormalization presented in the chapter, the only possible denormalization of this table would be to combine it with the VENDOR table. Because each invoice must have a vendor, and the only additional data about vendors not in the INVOICE table is the Vendor_Name attribute, it is a good candidate for denormalization. Because Vendor_Name is not especially volatile, repeating Vendor_Name in each invoice for the same vendor will not cause excessive update maintenance. If Vendor_Name is often used with other invoice data when invoice data are displayed, then, indeed, it would be a good candidate for denormalization. So, the denormalized relation to be transformed into a physical table is:

INVOICE(Vendor_ID, Invoice_Number, Invoice_Date, Paid?, Vendor_Name)

The next decision can be what indexes to create. The guidelines presented in this chapter suggest creating an index for the primary key, all foreign keys, and secondary keys used for sorting and qualifications in queries. So, we create a primary key index on the combined fields Vendor_ID and Invoice_Number. INVOICE has no foreign keys. To determine what fields are used as secondary keys in query sorting and qualification clauses, we would need to know the content of queries. Also, it would be helpful to know query frequency, because indexes do not provide much performance efficiency for infrequently run queries. For simplicity, suppose only two frequently run queries reference the INVOICE table, as follows:

1. Display all the data about all unpaid invoices due this week.
2. Display all invoices sorted by vendor: show all unpaid invoices first, then all paid invoices, and order the invoices of each category in reverse sequence by invoice date.

In the first query, both the Paid? and Invoice_Date fields are used for qualification. Paid?, however, may not be a good candidate for an index because this field contains only two values. The systems analyst would need to discover what percentage of invoices on file are unpaid. If this value is more than 10 percent,

TABLE 9-4: INVOICE Table Field Design Parameters for Hoosier Burger

	Physical Design Parameter				
Field	**Data Type and Size**	**Format and Input Mask**	**Default Value**	**Validation Rule**	**Required, Zero Length**
Vendor_ID	Number	Fixed with 0 decimals, 9999	N/A	< 0	Required, not 0 length
Invoice_Number	Text, 10	LL99–99999	N/A	N/A	Required, not 0 length
Invoice_Date	Date/Time	Medium date	= Date( )	> #1/1/2000	Not required
Paid?	Yes/No	N/A	False	N/A	Required
Vendor_Name	Text, 30	N/A	N/A	N/A	Required, may be 0 length

then an index on Paid? would not likely be helpful. Invoice_Date is a more discriminating field, so an index on this field would be helpful.

In the second query, Vendor_ID, Paid?, and Invoice_Date are used for sorting. Vendor_ID and Invoice_Date are discriminating fields (most values occur in less than 10 percent of the rows), so indexes on these fields will be helpful. Assuming less than 10 percent of the invoices on file are unpaid, then it would make sense to create the following indexes to make these two queries run as efficiently as possible:

Primary key index: Vendor_ID and Invoice_Number
Secondary key indices: Vendor_ID, Invoice_Date, and Paid?

Table 9-4 shows the decisions made for the properties of each field, based on reasonable assumptions about invoice data. Figure 9-4 illustrates a Microsoft Access table definition screen for the SHIPMENT table that includes the Invoice_Number field. It is the parameters on such a screen that must be specified for each field. Table 9-4 summarizes the field design parameters for the Invoice_Number field: size (width), format and input mask (picture), default value, validation rule (integrity control), and whether the field is required or is allowed zero length (null value controls); we have already indicated the indexing decision. Recall from Table 9-2 that the data type of Lookup Wizard implements referential integrity, but no foreign keys are in the INVOICE table because we combined the VENDOR table into the INVOICE table. We do not specify properties under the Lookup tab, which relates to additional data entry and display properties peculiar to Microsoft Access. Remember, we specify these parameters in physical database design, and it is in implementation that the Access tables would be defined using forms such as in Figure 9-4.

We do not illustrate security and other types of controls because these decisions are dependent on unique capabilities of the technology and a complex analysis of what data which users have the right to read, modify, add, or delete. This section illustrates the process of making many key physical database design decisions within the Microsoft Access environment.

Pine Valley Furniture WebStore: Designing Databases

Like many other analysis and design activities, designing the database for an Internet-based electronic commerce application is no different from the process followed when designing the database for other types of applications. In the last chapter, you read how Jim Woo and the Pine Valley Furniture development team designed the human interface for the WebStore. In this section, we

examine the processes Jim followed when transforming the conceptual data model for the WebStore into a set of normalized relations.

Designing Databases for Pine Valley Furniture's WebStore

The first step Jim took when designing the database for the WebStore was to review the conceptual data model—the entity-relationship diagram—developed during the analysis phase of the SDLC (see Figure 7-13 for a review). Given that the diagram contained no associative entities or many-to-many relationships, he began by identifying four distinct entity types that he named:

CUSTOMER
ORDER
INVENTORY
SHOPPING_CART

Once reacquainted with the conceptual data model, he examined the lists of attributes for each entity. He noted that three types of customers were identified during conceptual data modeling, namely: corporate customers, home-office customers, and student customers. Yet, all were simply referred to as a "customer." Nonetheless, because each type of customer had some unique information (attributes) that other types of customers did not, Jim created three additional entity types, or subtypes, of customers:

CORPORATE
HOME_OFFICE
STUDENT

Table 9-5 lists the common and unique information about each customer type. As Table 9-5 implies, four separate relations are needed to keep track of customer information without having anomalies. The CUSTOMER relation is used to capture common attributes, whereas the additional relations are used to capture

TABLE 9-5: Common and Unique Information about Each Customer Type*

Common Information about ALL Customer Types		
Corporate Customer	**Home-Office Customer**	**Student Customer**
Customer ID	Customer ID	Customer ID
Address	Address	Address
Phone	Phone	Phone
E-mail	E-mail	E-mail
Unique Information about EACH Customer Type		
Corporate Customer	**Home-Office Customer**	**Student Customer**
Corporate name	Customer name	Customer name
Shipping method	Corporate name	School
Buyer name	Fax	
Fax		

*Having multiple "types" of an entity, with some sharing common attributes and each having unique attributes, is modeled in E-R diagrams as a subclass entity and is commonly referred to as an "is a" relationship (e.g., a customer is a corporate customer, a customer is a home-office customer, or a customer is a student customer). Please see a comprehensive database management text such as Hoffer, Ramesh, and Topi (2011) for more information on subclass entities and "is a" relationships.

information unique to each distinct customer type. In order to identify the type of customer within the CUSTOMER relation easily, a Customer_Type attribute is added to the CUSTOMER relation. Thus, the CUSTOMER relation consists of:

CUSTOMER(Customer_ID, Address, Phone, E-mail, Customer_Type)

In order to link the CUSTOMER relation to each of the separate customer types—CORPORATE, HOME_OFFICE, and STUDENT—they all share the same primary key, Customer_ID, in addition to the attributes unique to each, which results in the following relations:

CORPORATE(Customer_ID, Corporate_Name, Shipping_Method, Buyer_Name, Fax)
HOME_OFFICE(Customer_ID, Customer_Name, Corporate_Name, Fax)
STUDENT(Customer_ID, Customer_Name, School)

In addition to identifying all the attributes for customers, Jim also identified the attributes for the other entity types. The results of this investigation are summarized in Table 9-6. As described in Chapter 7, much of the order-related information is captured and tracked within PVF's Purchasing Fulfillment System. Therefore, the ORDER relation does not need to track all the details of the order because the Purchasing Fulfillment System produces a detailed invoice that contains all order details, such as the list of ordered products, materials used, colors, quantities, and other such information. In order to access this invoice information, a foreign key, Invoice_ID, is included in the ORDER relation. Additionally, to easily identify which orders belong to a specific customer, the Customer_ID attribute is also included in ORDER. Two additional attributes, Return_Code and Order_Status, are also included in ORDER. The Return_Code is used to track the return of an order more easily—or a product within an order—whereas Order_Status is a code used to represent the state of an order as it moves through the purchasing fulfillment process. These attributes result in the following ORDER relation:

ORDER(Order_ID, Invoice_ID, Customer_ID, Return_Code, Order_Status)

In the INVENTORY entity, two attributes—Materials and Colors—could take on multiple values but were represented as single attributes. For example, Materials represents the range of materials from which a particular inventory item could be constructed. Likewise, Colors is used to represent the range of possible

TABLE 9-6: Attributes for Order, Inventory, and Shopping Cart Entities

Order	Inventory	Shopping_Cart
Order_ID (primary key)	Inventory_ID (primary key)	Cart_ID (primary key)
Invoice_ID (foreign key)	Name	Customer_ID (foreign key)
Customer_ID (foreign key)	Description	Inventory_ID (foreign key)
Return_Code	Size	Material
Order_Status	Weight	Color
	Materials	Quantity
	Colors	
	Price	
	Lead_Time	

product colors. PVF has a long-established set of codes for representing materials and colors; each of these *complex* attributes is represented as a single attribute. For example, the value "A" in the Colors field represents walnut, dark oak, light oak, and natural pine, whereas the value "B" represents cherry and walnut. Using this coding scheme, PVF can use a single character code to represent numerous combinations of colors and results in the following INVENTORY relation:

INVENTORY(Inventory_ID, Name, Description, Size, Weight, Materials, Colors, Price, Lead_Time)

Finally, in addition to Cart_ID, each shopping cart contains the Customer_ID and Inventory_ID attributes so that each item in a cart can be linked to a particular inventory item and to a specific customer. In other words, both the Customer_ID and Inventory_ID attributes are foreign keys in the SHOPPING_CART relation. Recall that the SHOPPING_CART is temporary and is kept only while a customer is shopping. When a customer actually places the order, the ORDER relation is created and the line items for the order—the items in the shopping cart—are moved to the Purchase Fulfillment System and stored as part of an invoice. Because we also need to know the selected material, color, and quantity of each item in the SHOPPING_CART, these attributes are included in this relation, which results in the following:

SHOPPING_CART(Cart_ID, Customer_ID, Inventory_ID, Material, Color, Quantity)

Now that Jim had completed the database design for the WebStore, he shared all the design information with his project team so that the design could be turned into a working database during implementation. We read more about the WebStore's implementation in the next chapter.

Key Points Review

1. **Concisely define each of the following key database design terms: *relation, primary key, functional dependency, foreign key, referential integrity, field, data type, null value, denormalization, file organization, index,* and *secondary key.***

   A relation is a named, two-dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of unnamed rows. In logical database design, a relation corresponds to an entity or a many-to-many relationship from an E-R data model. One or more columns of each relation compose the primary key of the relation, values for which distinguish each row of data in the relation. A functional dependency is a particular relationship between two attributes. For a given relation, attribute B is functionally dependent on attribute A if, for every valid value of A, that value of A uniquely determines the value of B. The functional dependence of B on A is represented by A→B. The primary goal of logical database design is to develop relations in which all the nonprimary key attributes of a relation functionally depend on the whole primary key and nothing but the primary key. Relationships between relations are represented by placing the primary key of the table on the one side of the relationship as an attribute (also known as a foreign key) in the relation on the many side of the relationship. Foreign keys must satisfy referential integrity, which means that the value (or existence) of an attribute depends on the value (or existence) of the same attribute in another relation. The specifications for a database in terms of relations must be transformed into technology-related terms before the database can be implemented. A field is the smallest unit of stored data in a database and typically corresponds to an attribute in a relation. Each field has a data type, which is a coding scheme recognized by system software for representing organizational data. A null value for a field is a special field value, distinct from 0, blank, or any other value, that indicates that the value for the field is missing or otherwise unknown. Denormalization is an important process in designing a physical database, by which normalized relations are split or combined into

physical tables based on affinity of use of rows and fields. A file organization is a technique for physically arranging the records of a physical file. Many types of file organizations utilize an index, which is a table (not related to the E-R diagram for the application) used to determine the location of rows in a file that satisfy some condition. An index can be created on a primary or a secondary key, which is one or a combination of fields for which more than one row may have the same combination of values.

2. **Explain the role of designing databases in the analysis and design of an information system.**

   Databases are defined during the systems design phase of the systems development life cycle. They are designed usually in parallel with the design of system interfaces. To design a database, a systems analyst must understand the conceptual database design for the application, usually specified by an E-R diagram, and the data requirements of each system interface (report, form, screen, etc.). Thus, database design is a combination of top-down (driven by an E-R diagram) and bottom-up (driven by specific information requirements in system interfaces) processes. Besides data requirements, systems analysts must also know physical data characteristics (e.g., length and format), frequency of use of the system interfaces, and the capabilities of database technologies.

3. **Transform an entity-relationship (E-R) diagram into an equivalent set of well-structured (normalized) relations.**

   An E-R diagram is transformed into normalized relations by following well-defined principles summarized in Table 9-1. For example, each entity becomes a relation and each many-to-many relationship or associative entity also becomes a relation. The principles also specify how to add foreign keys to relations to represent one-to-many relationships. You may want to review Table 9-1 at this point.

4. **Merge normalized relations from separate user views into a consolidated set of well-structured relations.**

   Separate sets of normalized relations are merged (this process is also called *view integration*) to create a consolidated logical database design. The different sets of relations come from the conceptual E-R diagram for the application, known human system interfaces (reports, screens, forms, etc.), and known or anticipated queries for data that meet certain qualifications. The result of merging is a comprehensive, normalized set of relations for the application. Merging is not simply a mechanical process. A systems analyst must address issues of synonyms, homonyms, and dependencies between nonkeys during view integration.

5. **Choose storage formats for fields in database tables.**

   Fields in the physical database design represent the attributes (columns) of relations in the logical database design. Each field must have a data type and potentially other characteristics, such as a coding scheme to simplify the storage of business data, default value, input mask, range control, referential integrity control, or null value control. A storage format is chosen to balance four objectives: (1) minimize storage space, (2) represent all possible values of the field, (3) improve data integrity for the field, and (4) support all data manipulations desired on the field.

6. **Translate well-structured relations into efficient database tables.**

   Whereas normalized relations possess properties of well-structured relations, the design of a physical table attempts to achieve two goals different from those of normalization: efficient use of secondary storage and data-processing speed. Efficient use of storage means that the amount of extra (or overhead) information is minimized. So, file organizations, such as sequential, are efficient in the use of storage because little or no extra information, besides the meaningful business data, are kept. Data-processing speed is achieved by keeping storage data close together that are used together and by building extra information in the database that allows data to be quickly found based on primary or secondary key values or by sequence.

7. **Explain when to use different types of file organizations to store computer files.**

   Table 9-3 summarizes the performance characteristics of different types of file organizations. The systems analyst must decide which performance factors are most important for each application and the associated database. These factors are storage space, sequential retrieval speed, random-row retrieval speed, speed of retrieving data based on multiple key qualifications, and the speed to perform data maintenance activities of row deletion, addition, and updating.

8. **Describe the purpose of indexes and the important considerations in selecting attributes to be indexed.**

   An index is information about the primary or secondary keys of a file. Each index entry contains the key value and a pointer to the row that contains that key value. Using indexes involves a trade-off between improved performance for retrievals and

degrading performance for inserting, deleting, and updating the rows in a file. Thus, indexes should be used generously for databases intended primarily to support data retrievals, such as for decision support applications. Because they impose additional overhead, indexes should be used judiciously for databases that support transaction processing and other applications with heavy updating requirements. Typically, you create indexes on a file for its primary key, foreign keys, and other attributes used in qualification and sorting clauses in queries, forms, reports, and other system interfaces.

Key Terms Checkpoint

Here are the key terms from the chapter. The page where each term is first explained is in parentheses after the term.

1. **Calculated (or computed or derived) field (p. 295)**
2. **Data type (p. 294)**
3. **Default value (p. 296)**
4. **Denormalization (p. 298)**
5. **Field (p. 294)**
6. **File organization (p. 299)**
7. **Foreign key (p. 283)**
8. **Functional dependency (p. 282)**
9. **Hashed file organization (p. 303)**
10. **Homonym (p. 290)**
11. **Index (p. 302)**
12. **Indexed file organization (p. 302)**
13. **Input mask (p. 296)**
14. **Normalization (p. 281)**
15. **Null value (p. 297)**
16. **Physical file (p. 299)**
17. **Physical table (p. 297)**
18. **Pointer (p. 299)**
19. **Primary key (p. 276)**
20. **Recursive foreign key (p. 288)**
21. **Referential integrity (p. 284)**
22. **Relation (p. 280)**
23. **Relational database model (p. 280)**
24. **Second normal form (2NF) (p. 282)**
25. **Secondary key (p. 302)**
26. **Sequential file organization (p. 301)**
27. **Synonyms (p. 290)**
28. **Third normal form (3NF) (p. 283)**
29. **Well-structured relation (or table) (p. 280)**

Match each of the key terms above with the definition that best fits it.

____ 1. A named, two-dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of unnamed rows.

____ 2. A relation that contains a minimum amount of redundancy and allows users to insert, modify, and delete the rows without errors or inconsistencies.

____ 3. The process of converting complex data structures into simple, stable data structures.

____ 4. A particular relationship between two attributes.

____ 5. A relation for which every nonprimary key attribute is functionally dependent on the whole primary key.

____ 6. A relation that is in second normal form and has no functional (transitive) dependencies between two (or more) nonprimary key attributes.

____ 7. An attribute that appears as a nonprimary key attribute in one relation and as a primary key attribute (or part of a primary key) in another relation.

____ 8. An integrity constraint specifying that the value (or existence) of an attribute in one relation depends on the value (or existence) of the same attribute in another relation.

____ 9. A foreign key in a relation that references the primary key values of that same relation.

____ 10. Two different names that are used for the same attribute.

____ 11. A single attribute name that is used for two or more different attributes.

____ 12. The smallest unit of named application data recognized by system software.

____ 13. A coding scheme recognized by system software for representing organizational data.

____ 14. A field that can be derived from other database fields.

____ 15. The value a field will assume unless an explicit value is entered for that field.

____ 16. A pattern of codes that restricts the width and possible values for each position of a field.

____ 17. A special field value, distinct from 0, blank, or any other value, that indicates that the value for the field is missing or otherwise unknown.

____ 18. A named set of rows and columns that specifies the fields in each row of the table.

____ 19. The process of splitting or combining normalized relations into physical

tables based on affinity of use of rows and fields.

____ 20. A named set of table rows stored in a contiguous section of secondary memory.

____ 21. A technique for physically arranging the records of a file.

____ 22. A field of data that can be used to locate a related field or row of data.

____ 23. The rows in the file are stored in sequence according to a primary key value.

____ 24. The rows are stored either sequentially or nonsequentially, and an index is created that allows software to locate individual rows.

____ 25. A table used to determine the location of rows in a file that satisfy some condition.

____ 26. One or a combination of fields for which more than one row may have the same combination of values.

____ 27. The address for each row is determined using an algorithm.

____ 28. An attribute whose value is unique across all occurrences of a relation.

____ 29. Data represented as a set of related tables or relations.

Review Questions

1. What is the purpose of normalization?
2. List five properties of relations.
3. What problems can arise during view integration or merging relations?
4. How are relationships between entities represented in the relational data model?
5. What is the relationship between the primary key of a relation and the functional dependencies among all attributes within that relation?
6. How is a foreign key represented in relational notation?
7. Can instances of a relation (sample data) prove the existence of a functional dependency? Why or why not?
8. In what way does the choice of a data type for a field help to control the integrity of that field?
9. Contrast the differences between range control and referential integrity when controlling data integrity.
10. What is the purpose of denormalization? Why might you not want to create one physical table or file for each relation in a logical data model?
11. What factors influence the decision to create an index on a field?
12. Explain the purpose of data compression techniques.
13. What are the goals of designing physical tables?
14. What are the seven factors that should be considered in selecting a file organization?
15. What are the four key steps in logical database modeling and design?
16. What are the four steps in transforming an E-R diagram into normalized relations?

Problems and Exercises

1. Assume that at Pine Valley Furniture products consist of components, products are assigned to salespersons, and components are produced by vendors. Also assume that in the relation PRODUCT(Prodname, Salesperson, Compname, Vendor) Vendor is functionally dependent on Compname, and Compname is functionally dependent on Prodname. Eliminate the transitive dependency in this relation and form 3NF relations.
2. Transform the E-R diagram of Figure 7-20 into a set of 3NF relations. Make up a primary key and one or more nonkeys for each entity that does not already have them listed.
3. Transform the E-R diagram of Figure 9-21 into a set of 3NF relations.
4. Consider the list of individual 3NF relations that follow. These relations were developed from several separate normalization activities.

   PATIENT(Patient_ID, Room_Number, Admit_Date,_Address)
   ROOM(Room_Number, Phone, Daily_Rate)
   PATIENT(Patient_Number, Treatment_Description, Address)
   TREATMENT(Treatment_ID, Description, Cost)
   PHYSICIAN(Physician_ID, Name, Department)
   PHYSICIAN(Physician_ID, Name, Supervisor_ID)

   a. Merge these relations into a consolidated set of 3NF relations. Make and state whatever

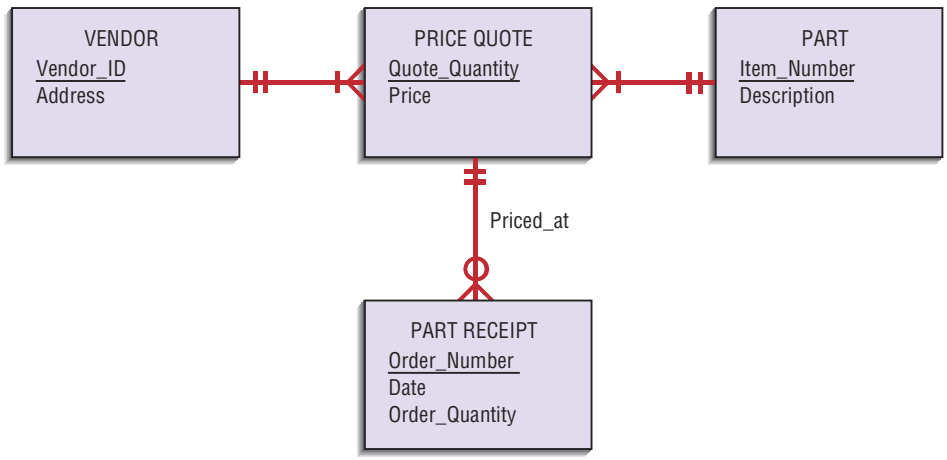


FIGURE 9-21
E-R diagram for Problem and Exercise 3.

assumptions you consider necessary to resolve any potential problems you identify in the merging process.

b. Draw an E-R diagram for your answer to part a.

5. Consider the following 3NF relations about a sorority or fraternity:

MEMBER(Member_ID, Name, Address, Dues_Owed)
OFFICE(Office_Name, Officer_ID, Term_Start_Date, Budget)
EXPENSE(Ledger_Number, Office_Name, _Expense_Date, Amt_Owed)
PAYMENT(Check_Number, Expense_Ledger_Number, Amt_Paid)
RECEIPT(Member_ID, Receipt_Date, Dues_Received)
COMMITTEE(Committee_ID, Officer_in_Charge)
WORKERS(Committee_ID, Member_ID)

a. Foreign keys are not indicated in these relations. Decide which attributes are foreign keys and justify your decisions.
b. Draw an E-R diagram for these relations, using your answer to part a.
c. Explain the assumptions you made about cardinalities in your answer to part b. Explain why it is said that the E-R data model is more expressive or more semantically rich than the relational data model.

6. Consider the following functional dependencies:

Applicant_ID→Applicant_Name
Applicant_ID→Applicant_Address
Position_ID→Position_Title
Position_ID→Date_Position_Opens
Position_ID→Department
Applicant_ID + Position_ID→Date_Applied
Applicant_ID + Position_ID + Date_Interviewed?→

a. Represent these attributes with 3NF relations. Provide meaningful relation names.
b. Represent these attributes using an E-R diagram. Provide meaningful entity and relationship names.

7. Suppose you were designing a file of student records for your university's placement office. One of the fields that would likely be in this file is the student's major. Develop a coding scheme for this field that achieves the objectives outlined in this chapter for field coding.
8. In Problem and Exercise 3, you developed integrated normalized relations. Choose primary keys for the files that would hold the data for these relations. Did you use attributes from the relations for primary keys or did you design new fields? Why or why not?
9. Suppose you created a file for each relation in your answer to Problem and Exercise 3. If the following queries represented the complete set of accesses to this database, suggest and justify what primary and secondary key indexes you would build.

a. For each PART, list all vendors and their associated prices for that part.
b. List all PART RECEIPTs, including related PART fields for all the parts received on a particular day.
c. For a particular VENDOR, list all the PARTs and their associated prices that VENDOR can supply.

10. Suppose you were designing a default value for the marriage status field in a student record at your university. What possible values would you consider and why? How would the default value change depending on other factors, such as type of student (undergraduate, graduate, professional)?
11. Consider Figure 9-19B. Explain a query that would likely be processed more quickly using the denormalized relations rather than the normalized relations.

12. Consider your answers to parts a and b of Problem and Exercise 11 in Chapter 7.
    a. Transform the E-R diagram you developed in part a into a set of 3NF relations. Clearly identify primary and foreign keys. Explain how you determined the primary key for any many-to-many relationships or associative entities.
    b. Transform the E-R diagram you developed in part b into a set of 3NF relations. Clearly identify primary and foreign keys. Explain how you determined the primary key for any many-to-many relationships or associative entities.
13. Model a set of typical family relationships—spouse, father, and mother—in a single 3NF relation. Also include nonkey attributes name and birth date. Assume that each person has only one spouse, one father, and one mother. Show foreign keys with dashed underlining.

Discussion Questions

1. Many database management systems offer the ability to enforce referential integrity. Why would using such a feature be a good idea? Are there any situations in which referential integrity might not be important?
2. Assume you are part of the systems development team at a medium-sized organization. You have just completed the database design portion of the systems design phase, and the project sponsor would like a status update. Assuming the project sponsor is a VP in the marketing department, with only a high-level understanding of technical subjects, how would you go about presenting the database design you have just completed? How would your presentation approach change if the project sponsor were the manager of the database team?
3. Discuss what additional information should be collected during requirements analysis that is needed for file and database design and that is not especially useful for earlier phases of systems development.
4. Find out what database management systems are available at your university for student use. Investigate which data types these DBMSs support. Compare these DBMSs based upon data types supported and suggest which types of applications each DBMS is best suited for based on this comparison.
5. Find out what database management systems are available at your university for student use. Investigate what physical file and database design decisions need to be made. Compare this list of decisions to those discussed in this chapter. For physical database and design decisions (or options) not discussed in the chapter, investigate what choices you have and how you should choose among these choices. Submit a report to your instructor with your findings.

Case Problems

1. Pine Valley Furniture

   Development work on Pine Valley Furniture's new Customer Tracking System is proceeding according to plan and is on schedule. The project team has been busy designing the human interfaces, and you have just completed the new tracking system's Customer Profile Form, Products by Demographics Summary Report, and Customer Purchasing Frequency Report.

   Because you are now ready for a new task, Jim Woo asks you to prepare logical data models for the form and two reports that you have just designed and drop them by his office this afternoon. At that time, the two of you will prepare a consolidated database model, translate the Customer Tracking System's E-R data model into normalized relations, and then integrate the logical data models into a final logical data model for the Customer Tracking System.

   a. Develop logical data models for the form and two reports mentioned in the case scenario.
   b. Perform view integration on the logical models developed for part a.
   c. What view integration problems, if any, exist? How should you correct these problems?
   d. Have a fellow classmate critique your logical data model. Make any necessary corrections.

2. Hoosier Burger

   As the lead analyst on the Hoosier Burger project, you have had the opportunity to learn more about the systems development process, work with project team members, and interact with the system's end users, especially with Bob and

Thelma. You have just completed the design work for the various forms and reports that will be used by Bob, Thelma, and their employees. Now it is time to prepare logical and physical database designs for the new Hoosier Burger system.

During a meeting with Hoosier Burger project team members, you review the four steps in logical database modeling and design. It will be your task to prepare the logical models for the Customer Order Form, Customer Account Balance Form, Daily Delivery Sales Report, and Inventory Low-in-Stock Report. At the next meeting, the E-R model will be translated and a final logical model produced.

a. Develop logical models for each of the interfaces mentioned in the case scenario.
b. Integrate the logical models prepared for part a into a consolidated logical model.
c. What types of problems can arise from view integration? Did you encounter any of these problems when preparing the consolidated logical model?
d. Using your newly constructed logical model, determine which fields should be indexed. Which fields should be designated as calculated fields?

3. PlowMasters

PlowMasters is a locally owned and operated snow removal business. PlowMasters provides residential and commercial snow removal for clients throughout a large metropolitan area. Typical services include driveway and walkway snow removal, as well as parking lot snow maintenance for larger commercial clients.

PlowMasters' clientele has grown over the past several snow seasons. Recent heavy snowfall, coupled with a successful advertising campaign, has increased current demand even more, and this increase in demand is expected to continue. In order to provide faster, more efficient service, PlowMasters has hired your consulting company to design, develop, and implement a computer-based system. Your development team is currently preparing the logical and physical database designs for PlowMasters.

a. What are the four steps in logical database modeling and design?
b. Several relations have been identified for this project, including removal technician, customer, service provided, equipment inventory, and services offered. What relationships exist among these relations? How should these relationships be represented?
c. Think of the attributes that would most likely be associated with the relations identified in the part b. For each data integrity control method discussed in the chapter, provide a specific example.
d. What are the guidelines for choosing indexes? Identify several fields that should be indexed.

CASE: PETRIE'S ELECTRONICS

Designing Databases

Jim Watanabe, assistant director of IT for Petrie's Electronics and the manager of the "No Customer Escapes" customer loyalty system project, was walking down the hall from his office to the cafeteria. It was 4 P.M., but Jim was nowhere close to going home yet. The deadlines he had imposed for the project were fast approaching. His team was running behind, and he had a lot of work to do over the next week to try to get things back on track. He needed to get some coffee for the start of what was going to be a late night.

As Jim approached the cafeteria, he saw Sanjay Agarwal and Sam Waterston walking toward him. Sanjay was in charge of systems integration for Petrie's, and Sam was one of the company's top interface designers. They were both on the customer loyalty program team. They were having an intense conversation as Jim approached.

"Hi guys," Jim said.

"Oh, hi, Jim," Sanjay replied. "Glad I ran into you—we are moving ahead on the preliminary database designs. We're translating the earlier conceptual designs into physical designs."

"Who's working on that? Stephanie?" Jim asked. Stephanie Welch worked for Petrie's database administrator.

"Yes," Sanjay replied. "But she is supervising a couple of interns who have been assigned to her for this task."

"So how is that going? Has she approved their work?"

"Yeah, I guess so. It all seems to be under control."

"I don't want to second-guess Stephanie, but I'm curious about what they've done."