Chapter 5

Logical Database Design and the Relational Model

Mapping ER/EER To Relations and Normalization

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Define the following key terms: **relation, primary key, composite key, foreign key, null, entity integrity rule, referential integrity constraint, well-structured relation, anomaly, recursive foreign key, normalization, normal form, functional dependency, determinant, candidate key, first normal form, second normal form, partial functional dependency, third normal form, transitive dependency, synonyms, alias, homonym,** and **enterprise key.**
- List five properties of relations.
- State two properties that are essential for a candidate key.
- Give a concise definition of each of the following: first normal form, second normal form, and third normal form.
- Briefly describe four problems that may arise when merging relations.
- Transform an E-R (or EER) diagram to a logically equivalent set of relations.
- Create relational tables that incorporate entity integrity and referential integrity constraints.
- Use normalization to decompose a relation with anomalies into well-structured relations.

INTRODUCTION

In this chapter we describe logical database design, with special emphasis on the relational data model. Logical database design is the process of transforming the conceptual data model (described in Chapters 3 and 4) into a logical data model. Although there are other data models, we have two reasons for emphasizing the relational data model in this chapter. First, the relational data model is most commonly used in contemporary database applications. Second, some of the principles of logical database design for the relational model apply to the other logical models as well.

We have introduced the relational data model informally through simple examples in earlier chapters. In this chapter, we first define the important terms and concepts for this model. (We will often use the abbreviated term *relational model* when referring to the rela-

tional data model.) We next describe and illustrate the process of transforming an E-R model into the relational model. Many CASE tools support this transformation today; however, it is important that you understand the underlying principles and procedures. We then describe the concepts of normalization in detail. Normalization, which is the process of designing well-structured relations, is an important component of logical design for the relational model. Finally, we describe how to merge relations while avoiding common pitfalls that may occur in this process.

The objective of logical database design is to translate the conceptual design (which represents an organization's requirements for data) into a logical database design that can be implemented on a chosen database management system. The resulting databases must meet user needs for data sharing, flexibility, and ease of access. The concepts presented in this chapter are essential to your understanding of the database development process.

THE RELATIONAL DATA MODEL

The relational data model was first introduced in 1970 by E. F. Codd, then of IBM (Codd, 1970). Two early research projects were launched to prove the feasibility of the relational model and to develop prototype systems. The first of these, at IBM's San Jose Research Laboratory, led to the development of System R (a prototype relational DBMS–RDBMS) during the late 1970s. The second, at the University of California at Berkeley, led to the development of Ingres, an academically oriented RDBMS. Commercial RDBMS products from numerous vendors started to appear about 1980 (see the Website for this book for links to RDBMS and other DBMS vendors). Today RDBMSs have become the dominant technology for database management, and there are literally hundreds of RDBMS products for computers ranging from personal computers to mainframes.

Basic Definitions

The relational data model represents data in the form of tables. The relational model is based on mathematical theory and therefore has a solid theoretical foundation. However, we need only a few simple concepts to describe the relational model, and it is therefore easily understood and used by those unfamiliar with the underlying theory. The relational data model consists of the following three components (Fleming and von Halle, 1989):

1. *Data structure* Data are organized in the form of tables with rows and columns.
2. *Data manipulation* Powerful operations (using the SQL language) are used to manipulate data stored in the relations.
3. *Data integrity* Facilities are included to specify business rules that maintain the integrity of data when they are manipulated.

We discuss data structure and data integrity in this section. Data manipulation is discussed in Chapters 7, 8, and 10.

Relation: A named two-dimensional table of data.

Relational Data Structure 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. An attribute, consistent with its definition in Chapter 3, is a named column of a relation. Each row of a relation corresponds to a record that contains data (attribute) values for a single entity. Figure 5-1 shows an example of a relation named EMPLOYEE1. This relation contains the following attributes describing employees: Emp_ID, Name, Dept_Name, and Salary. The five rows of the table correspond to five employees. It is important to understand that the sample data in Figure

Figure 5-1
EMPLOYEE1 relation with sample data

EMPLOYEE1

Emp_ID	Name	Dept_Name	Salary
100	Margaret Simpson	Marketing	48,000
140	Allen Beeton	Accounting	52,000
110	Chris Lucero	Info Systems	43,000
190	Lorenzo Davis	Finance	55,000
150	Susan Martin	Marketing	42,000

1 are intended to illustrate the structure of the EMPLOYEE1 relation; they are not art of the relation itself. Even if we add another row of data to the figure, it is still the ame EMPLOYEE1 relation. Nor does deleting a row change the relation. In fact, we ould delete *all* of the rows shown in Figure 5-1, and the EMPLOYEE1 relation would till exist. Stated differently, Figure 5-1 is an instance of the EMPLOYEE1 relation.

We 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 that relation. For EMPLOYEE1 we would have:

EMPLOYEE1(Emp_ID,Name,Dept_Name,Salary).

Relational Keys We must be able to store and retrieve a row of data in a relation, based on the data values stored in that row. To achieve this goal, every relation must have a primary key. A **primary key** is an attribute (or combination of attributes) that uniquely identifies each row in a relation. We designate a primary key by underlining e attribute name. For example, the primary key for the relation EMPLOYEE1 is Emp_ID. Notice that this attribute is underlined in Figure 5-1. In shorthand notation we express this relation as follows:

Primary key: An attribute (or combination of attributes) that uniquely identifies each row in a relation.

EMPLOYEE1(Emp_ID,Name,Dept_Name,Salary)

The concept of a primary key is related to the term "identifier" defined in Chapter 3. The same attribute (or attributes) indicated as an entity's identifier in an E-R diagram may be the same attributes that compose the primary key for the relation representing that entity. There are exceptions; for example, associative entities do not have to have an identifier and the identifier of a weak entity forms only part of a weak entity's primary key. In addition, there may be several attributes of an entity that may serve as the associated relation's primary key. All of these situations will be illustrated later in this chapter.

A **composite key** is a primary key that consists of more than one attribute. For example, the primary key for a relation DEPENDENT would likely consist of the combination Emp_ID and Dependent_Name. We show several examples of composite keys later in this chapter.

Composite key: A primary key that consists of more than one attribute.

Often we must represent the relationship between two tables or relations. This is accomplished through the use of foreign keys. A **foreign key** is an attribute (possibly composite) in a relation of a database that serves as the primary key of another relation in the same database. For example, consider the relations EMPLOYEE1 and DEPARTMENT:

Foreign key: An attribute in a relation of a database that serves as the primary key of another relation in the same database.

EMPLOYEE1(Emp_ID,Name,Dept_Name,Salary)

DEPARTMENT(Dept_Name,Location, Fax)

The attribute Dept_Name is a foreign key in EMPLOYEE1. It allows a user to associate any employee with the department to which he or she is assigned. Some authors emphasize the fact that an attribute is a foreign key by using a dashed underline, such as

EMPLOYEE1(Emp_ID,Name,Dept_Name,Salary)

We provide numerous examples of foreign keys in the remainder of this chapter and discuss the properties of foreign keys under the heading Referential Integrity.

Properties of Relations We have defined relations as two-dimensional tables of data. However, not all tables are relations. Relations have several properties that distinguish them from nonrelational tables. We summarize these properties below.

1. Each relation (or table) in a database has a unique name.
2. An entry at the intersection of each row and column is atomic (or single-valued). There can be no multivalued attributes in a relation.
3. Each row is unique; no two rows in a relation are identical.
4. Each attribute (or column) within a table has a unique name.
5. The sequence of columns (left to right) is insignificant. The columns of a relation can be interchanged without changing the meaning or use of the relation.
6. The sequence of rows (top to bottom) is insignificant. As with columns, the rows of a relation may be interchanged or stored in any sequence.

Removing Multivalued Attributes from Tables The second property of relations above states that there can be no multivalued attributes in a relation. Thus a table that contains one or more multivalued attributes is not a relation. As an example, Figure 5-2a shows the employee data from the EMPLOYEE1 relation extended to include courses that may have been taken by those employees. Since a given employee may have taken more than one course, the attributes Course_Title and Date_Completed are multivalued attributes. For example, the employee with

Figure 5-2
Eliminating multivalued attributes
(a) Table with repeating groups

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
				Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000	SPSS	1/12/200X
				C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/16/200X
				Java	8/12/200X

(b) EMPLOYEE2 relation

EMPLOYEE2

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
100	Margaret Simpson	Marketing	48,000	Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000	SPSS	1/12/200X
110	Chris Lucero	Info Systems	43,000	C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/19/200X
150	Susan Martin	Marketing	42,000	Java	8/12/200X

Emp_ID 100 has taken two courses. If an employee has not taken any courses, the Course_Title and Date_Completed attribute values are null (see employee with Emp_ID 190 for an example).

We show how to eliminate the multivalued attributes in Figure 5-2b by filling the relevant data values into the previously vacant cells of Figure 5-2a. As a result, the table in Figure 5-2b has only single-valued attributes and now satisfies the atomic property of relations. The name EMPLOYEE2 is given to this relation to distinguish it from EMPLOYEE1. However, as you will see below, this new relation does have some undesirable properties.

Example Database

A relational database consists of any number of relations. The structure of the database is described through the use of a conceptual schema (defined in Chapter 2), which is a description of the overall logical structure of the database. There are two common methods for expressing a (conceptual) schema:

a. Short text statements, in which each relation is named and the names of its attributes follow in parentheses (see EMPLOYEE1 and DEPARTMENT relations defined earlier in this chapter).

b. A graphical representation, in which each relation is represented by a rectangle containing the attributes for the relation.

Text statements have the advantage of simplicity. On the other hand, a graphical representation provides a better means of expressing referential integrity constraints (as you will see shortly). In this section we use both techniques for expressing a schema so that you can compare them.

We developed an entity-relationship diagram for Pine Valley Furniture in Chapter 3 (see Figure 3-22). In Chapter 1 we showed the following four relations from this database (see Figure 1-4): CUSTOMER, ORDER, ORDER LINE, and PRODUCT. In this section we show a schema for these four relations. A graphical representation is shown in Figure 5-3.

Figure 5-3
Schema for four relations (Pine Valley Furniture)

CUSTOMER

Customer_ID	Customer_Name	Address	City	State	Zip

ORDER

Order_ID	Order_Date	Customer_ID

ORDER LINE

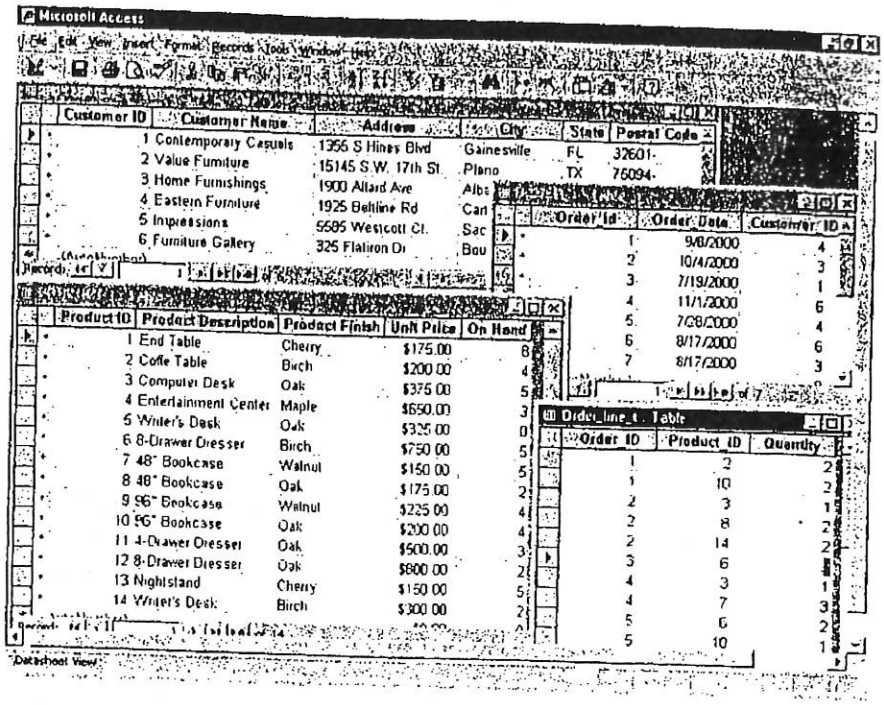
Order_ID	Product_ID	Quantity

PRODUCT

Product_ID	Product_Description	Product_Finish	Standard_Price	On_Hand

* Not in Figure 3-22 for simplicity.

Figure 5-4
Instance of a relational schema



Following is a text description for the relations:

CUSTOMER(Customer_ID,Customer_Name,Address,City, State,Zip)

ORDER(Order_ID,Order_Date,Customer_ID)

ORDER LINE(Order_ID,Product_ID,Quantity)

PRODUCT(Product_ID,Product_Description,Product_Finish,Standard_Price, On_Hand)

Notice that the primary key for ORDER LINE is a composite key consisting of the attributes Order_ID and Product_ID. Also Customer_ID is a foreign key in the ORDER relation; this allows the user to associate an order with the customer who submitted the order. ORDER LINE has two foreign keys: Order_ID, and Product_ID. These keys allow the user to associate each line on an order with the relevant order and product.

An instance of this database is shown in Figure 5-4. This figure shows four tables with sample data. Notice how the foreign keys allow us to associate the various tables. It is a good idea to create an instance of your relational schema with sample data for three reasons:

1. The sample data provide a convenient way to check the accuracy of your design.
2. The sample data help improve communications with users in discussing your design.
3. You can use the sample data to develop prototype applications and to test queries.

INTEGRITY CONSTRAINTS

The relational data model includes several types of constraints, or business rules, whose purpose is to facilitate maintaining the accuracy and integrity of data in the database. The major types of integrity constraints are domain constraints, entity integrity, referential integrity, and action assertions.

Domain Constraints

All of the values that appear in a column of a relation must be taken from the same domain. A domain is the set of values that may be assigned to an attribute. A domain definition usually consists of the following components: domain name, meaning, data type, size (or length), and allowable values or allowable range (if applicable). Table 5-1 shows domain definitions for the domains associated with the attributes in Figure 5-3.

Entity Integrity

The entity integrity rule is designed to assure that every relation has a primary key, and that the data values for that primary key are all valid. In particular, it guarantees that every primary key attribute is non-null.

In some cases a particular attribute cannot be assigned a data value. There are two situations where this is likely to occur: Either there is no applicable data value, or the applicable data value is not known when values are assigned. Suppose for example that you fill out an employment form that has a space reserved for a fax number. If you have no fax number, you leave this space empty since it does not apply to you. Or suppose that you are asked to fill in the telephone number of your previous employer. If you do not recall this number, you may leave it empty since that information is not known.

The relational data model allows us to assign a null value to an attribute in the just described situations. A **null** is a value that may be assigned to an attribute when no other value applies or when the applicable value is unknown. In reality, a null is not a value but rather the absence of a value. For example, it is not the same as a numeric zero or a string of blanks. The inclusion of nulls in the relational model is somewhat controversial, since it sometimes leads to anomalous results (Date, 1995). On the other hand, Codd advocates the use of nulls for missing values (Codd, 1990).

Null: A value that may be assigned to an attribute when no other value applies or when the applicable value is unknown.

One thing on which everyone agrees is that primary key values must not be allowed to be null. Thus the **entity integrity rule** states the following: No primary key attribute (or component of a primary key attribute) may be null.

Entity integrity rule: No primary key attribute (or component of a primary key attribute) can be null.

Referential Integrity

In the relational data model, associations between tables are defined through the use of foreign keys. For example in Figure 5-4, the association between the CUSTOMER and ORDER tables is defined by including the Customer_ID attribute as a foreign key in ORDER. This of course implies that before we insert a new row in the ORDER table, the customer for that order must already exist in the CUSTOMER

Table 5-1 Domain Definitions for Selected Attributes

Attribute	*Domain Name*	*Description*	*Domain*
Customer_ID	Customer_IDs	Set of all possible customer IDs	character: size 5
Customer_Name	Customer_Names	Set of all possible customer names	character: size 25
Customer_Address	Customer_Addresses	Set of all possible customer addresses	character: size 30
Customer_City	Cities	Set of all possible cities	character: size 20
Customer_State	States	Set of all possible states	character: size 2
Customer_Zip	Zips	Set of all possible zip codes	character: size 10
Order_ID	Order_IDs	Set of all possible order IDs	character: size 5
Order_Date	Order_Dates	Set of all possible order dates	date format mm-dd-yy
Product_ID	Product_IDs	Set of all possible product IDs	character: size 5
Product_Description	Product_Descriptions	Set of all possible product descriptions	character size 25
Product_Finish	Product_Finishes	Set of all possible product finishes	character: size 12
Standard_Price	Unit_Prices	Set of all possible unit prices	monetary: 6 digits
On_Hand	On_Hands	Set of all possible on hands	integer: 3 digits

Referential integrity constraint: A rule that states that either each foreign key value must match a primary key value in another relation or the foreign key value must be null.

table. If you examine the rows in the ORDER table in Figure 5-4, you will find that every customer number for an order already appears in the CUSTOMER table.

A **referential integrity constraint** is a rule that maintains consistency among the rows of two relations. The rule states that if there is a foreign key in one relation, either each foreign key value must match a primary key value in another relation or the foreign key value must be null. You should examine the tables in Figure 5-4 to check whether the referential integrity rule has been enforced.

The graphical version of the relational schema provides a simple technique for identifying associations where referential integrity must be enforced. Figure 5-5 shows the schema for the relations introduced in Figure 5-3. An arrow has been drawn from each foreign key to the associated primary key. A referential integrity constraint must be defined for each of these arrows in the schema.

How do you know if a foreign key is allowed to be null? A simple answer can be seen in Figure 3-22, the E-R diagram associated with Figures 5-4 and 5-5. In Figure 3-22, because each order must have a customer (the minimum cardinality next to CUSTOMER on the Submits relationship is one), the foreign key of Customer_ID cannot be null on the ORDER relation. If the minimum cardinality had been zero, then the foreign key could be null. Whether a foreign key can be null must be specified as a property of the foreign key attribute when the database is defined.

Actually, whether a foreign key can be null is more complex to model on an E-R diagram and to determine than we have shown so far. For example, what happens to order data if we choose to delete a customer who has submitted orders? We may want to see sales even if we do not care about the customer any more. Three choices are possible:

1. Delete the associated orders (called a cascading delete), in which case we lose not only the customer but also all the sales history.
2. Prohibit deletion of the customer until all associated orders are first deleted (a safety check).
3. Place a null value in the foreign key (an exception that says although an order must have a Customer_ID value when the order is created, Customer_ID can become null later if the associated customer is deleted).

We will see how each of these choices is implemented when we describe the SQL database query language in Chapter 7.

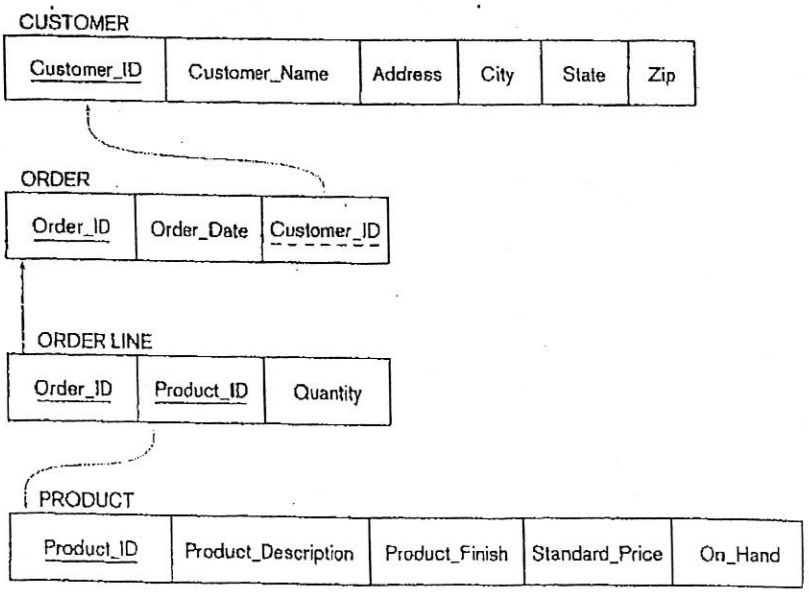


Figure 5-5
Referential integrity constraints (Pine Valley Furniture)

Action Assertions

In Chapter 4 we discussed business rules and introduced a new category of business rules we called action assertions. For example, a typical action assertion might state the following: "A person may purchase a ticket for the all-star game only if that person is a season-ticket holder." There are various techniques for defining and enforcing such rules. We discuss some of these techniques in later chapters.

Creating Relational Tables

In this section we create table definitions for the four tables shown in Figure 5-5. These definitions are created using **CREATE TABLE** statements from the SQL data definition language. In practice, these table definitions are actually created during the implementation phase later in the database development process. However, we show these sample tables in this chapter for continuity and especially to illustrate the way the integrity constraints described above are implemented in SQL.

The SQL table definitions are shown in Figure 5-6. One table is created for each of the four tables shown in the relational schema (Figure 5-5). Each attribute for a table is then defined. Notice that the data type and length for each attribute is taken from the domain definitions (Table 5-1). For example, the attribute Customer_Name in the CUSTOMER relation is defined as VARCHAR (variable character) data type with length 25. By specifying **NOT NULL**, each attribute can be constrained from being assigned a null value.

The primary key for each table is specified for each table using the **PRIMARY KEY** clause at the end of each table definition. The ORDER_LINE table illustrates how to specify a primary key when that key is a composite attribute. In this example,

Figure 5-6
SQL table definitions

```
CREATE TABLE CUSTOMER
        (CUSTOMER_ID            VARCHAR(5)      NOT NULL,
        CUSTOMER_NAME           VARCHAR(25)     NOT NULL,
        CUSTOMER ADDRESS        VARCHAR(30)     NOT NULL,
        CUSTOMER_CITY           VARCHAR(20)     NOT NULL,
        CUSTOMER_STATE          CHAR(2)         NOT NULL,
        CUSTOMER_ZIP            CHAR(10)        NOT NULL,
PRIMARY KEY (CUSTOMER_ID);

CREATE TABLE ORDER
        (ORDER_ID               CHAR(5)         NOT NULL,
        ORDER DATE              DATE            NOT NULL,
        CUSTOMER_ID             VARCHAR(5)      NOT NULL,
PRIMARY KEY (ORDER_ID),
FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (CUSTOMER_ID);

CREATE TABLE ORDER_LINE
        (ORDER_ID               CHAR(5)         NOT NULL,
        PRODUCT_ID              CHAR(5)         NOT NULL,
        QUANTITY                INT             NOT NULL,
PRIMARY KEY (ORDER_ID, PRODUCT_ID),
FOREIGN KEY (ORDER_ID) REFERENCES ORDER (ORDER_ID),
FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT (PRODUCT_ID);

CREATE TABLE PRODUCT
        (PRODUCT_ID             CHAR(5)         NOT NULL,
        PRODUCT_DESCRIPTION     VARCHAR(25),
        PRODUCT_FINISH          VARCHAR(12),
        STANDARD_PRICE          DECIMAL(8,2)    NOT NULL,
        ON_HAND                 INT             NOT NULL,
PRIMARY KEY (PRODUCT_ID);
```

the primary key of ORDER_LINE is the combination of Order_ID and Product_ID. Each primary key attribute in the four tables is constrained with **NOT NULL**. This enforces the entity integrity constraint described in the previous section. Notice that the **NOT NULL** constraint can also be used with non–primary key attributes.

Referential integrity constraints are easily defined, using the graphical schema shown in Figure 5-5. An arrow originates from each foreign key and "points to" the related primary key in the associated relation. In the SQL table definition, a **FOREIGN KEY REFERENCES** statement corresponds to each of these arrows. Thus for the table ORDER, the foreign key CUSTOMER_ID references the primary key of CUSTOMER, which is also CUSTOMER_ID. Although in this case the foreign key and primary keys have the same name, this need not be the case. For example, the foreign key attribute could be named CUST_NO instead of CUSTOMER_ID. However, the foreign and primary keys must be from the *same* domain.

The ORDER_LINE table provides an example of a table that has two foreign keys. Foreign keys in this table reference both the ORDER and PRODUCT tables.

Well-Structured Relations

Well-structured relation: A relation that contains minimal redundancy and allows users to insert, modify, and delete the rows in a table without errors or inconsistencies.

To prepare for our discussion of normalization, we need to address the following question: What constitutes a well-structured relation? Intuitively, a **well-structured relation** contains minimal redundancy and allows users to insert, modify, and delete the rows in a table without errors or inconsistencies. EMPLOYEE1 (Figure 5-1) 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 5-2b) is not a well-structured relation. If you examine the sample data in the table, you will notice considerable redundancy. For example, values for Emp_ID, Name, Dept_Name, and Salary 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).

Anomaly: An error or inconsistency that may result when a user attempts to update a table that contains redundant data. The three types of anomalies are insertion, deletion, and modification.

Redundancies in a table may result in errors or inconsistencies (called **anomalies**) when a user attempts to update the data in the table. Three types of anomalies are possible: insertion, deletion, and modification.

1. *Insertion anomaly* Suppose that we need to add a new employee to EMPLOYEE2. The primary key for this relation is the combination of Emp_ID and Course_Title (as noted earlier). Therefore, to insert a new row, the user must supply values for both Emp_ID and Course_Title (since primary key values cannot be null or nonexistent). This is an anomaly, since the user should be able to enter employee data without supplying course data.
2. *Deletion anomaly* Suppose that the data for employee number 140 are deleted from the table. This will result in losing the information that this employee completed a course (Tax Acc) on 12/8/200X. In fact, it results in losing the information that this course had an offering that completed on that date.
3. *Modification anomaly* Suppose that employee number 100 gets a salary increase. We must record the increase in each of the rows for that employee (two occurrences in Figure 5-2); otherwise the data will be inconsistent.

These anomalies indicate that EMPLOYEE2 is not a well-structured relation. The problem with this relation is that it contains data about two entities: EMPLOYEE and COURSE. We will use normalization theory (described below) to divide EMPLOYEE2 into two relations. One of the resulting relations is EMPLOYEE1 (Figure 5-1). The other we will call EMP_COURSE, which appears with sample data in Figure 5-7. The primary key of this relation is the combination of Emp_ID and

Emp_ID	Course_Title	Date_Completed
100	SPSS	6/19/200X
100	Surveys	10/7/200X
140	Tax Acc	12/8/200X
110	SPSS	1/12/200X
110	C++	4/22/200X
150	SPSS	6/19/200X
150	Java	8/12/200X

Figure 5-7
EMP_COURSE

Course_Title, and we underline these attribute names in Figure 5-7 to highlight this fact. Examine Figure 5-7 to verify that EMP_COURSE is free of the types of anomalies described above and is therefore well-structured.

TRANSFORMING EER DIAGRAMS INTO RELATIONS

During logical design you transform the E-R (and EER) diagrams that were developed during conceptual design into relational database schemas. The inputs to this process are the entity-relationship (and enhanced E-R) diagrams that you studied in Chapters 3 and 4. The outputs are the relational schemas described in the first two sections of this chapter.

Transforming (or mapping) E-R diagrams to relations is a relatively straightforward process with a well-defined set of rules. In fact, many CASE tools can automatically perform many of the conversion steps. However, it is important that you understand the steps in this process for three reasons:

1. CASE tools often cannot model more complex data relationships such as ternary relationships and supertype/subtype relationships. For these situations you may have to perform the steps manually.
2. There are sometimes legitimate alternatives where you will need to choose a particular solution.
3. You must be prepared to perform a quality check on the results obtained with a CASE tool.

In the following discussion we illustrate the steps in the transformation with examples taken from Chapters 3 and 4. It will help for you to recall that we discussed three types of entities in those chapters:

1. *Regular entities* are entities that have an independent existence and generally represent real-world objects such as persons and products. Regular entity types are represented by rectangles with a single line.
2. *Weak entities* are entities that cannot exist except with an identifying relationship with an owner (regular) entity type. Weak entities are identified by a rectangle with a double line.
3. *Associative entities* (also called gerunds) are formed from many-to-many relationships between other entity types. Associative entities are represented by a rectangle with a single line that encloses the diamond relationship symbol.

Step 1: Map Regular Entities

Each regular entity type in an ER diagram is transformed into a relation. The name given to the relation is generally the same as the entity type. Each simple attribute of the entity type becomes an attribute of the relation. The identifier of the

entity type becomes the primary key of the corresponding relation. You should check to make sure that this primary key satisfies the desirable properties of identifiers outlined in Chapter 3.

Figure 5-8a shows a representation of the CUSTOMER entity type for Pine Valley Furniture Company from Chapter 3 (see Figure 3-22). The corresponding CUSTOMER relation is shown in graphical form in Figure 5-8b. In this figure and those that follow in this section we show only a few key attributes for each relation to simplify the figures.

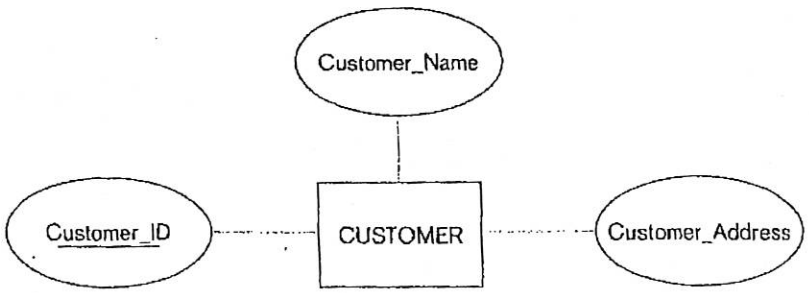
Composite Attributes When a regular entity type has a composite attribute, only the simple component attributes of the composite attribute are included in the new relation. Figure 5-9 shows a variation on the example of Figure 5-8, where Customer_Address is represented as a composite attribute with components Street, City, State, and Zip (see Figure 5-9a). This entity is mapped to the CUSTOMER relation, which contains the simple address attributes, as shown in Figure 5-9b.

Multivalued Attributes When the regular entity type contains a multivalued attribute, two new relations (rather than one) are created. The first relation contains all of the attributes of the entity type except the multivalued attribute. The second relation contains two attributes that form the primary key of the second relation. The first of these attributes is the primary key from the first relation, which becomes a foreign key in the second relation. The second is the multivalued attribute. The name of the second relation should capture the meaning of the multivalued attribute.

An example of this procedure is shown in Figure 5-10. This is the EMPLOYEE entity type for Pine Valley Furniture Company (Figure 3-22). As shown in Figure 5-10a, EMPLOYEE has Skill as a multivalued attribute. Figure 5-10b shows the two relations that are created. The first (called EMPLOYEE) has the primary key Employee_ID. The second relation (called EMPLOYEE_SKILL) has the two attributes Employee_ID and Skill, which form the primary key. The relationship between foreign and primary keys is indicated by the arrow in the figure.

The relation EMPLOYEE_SKILL contains no nonkey attributes (also called *descriptors*). Each row simply records the fact that a particular employee possesses a particular skill. This provides an opportunity for you to suggest to users that new attributes can be added to this relation. For example, the attributes Years_Experience and/or Certification_Date might be appropriate new values to add to this relation.

Figure 5-8
Mapping the regular entity CUSTOMER
(a) CUSTOMER entity type

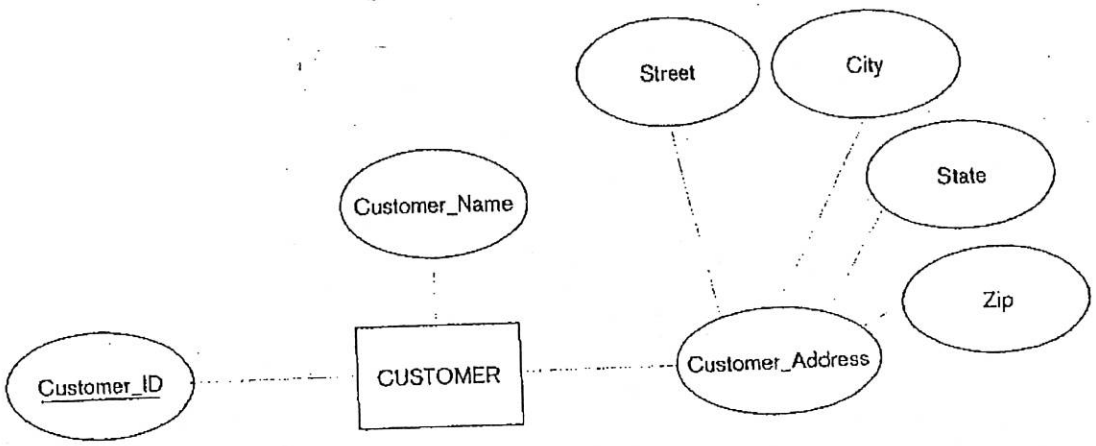


(b) CUSTOMER relation

CUSTOMER

Customer_ID	Customer_Name	Customer_Address

Figure 5-9
Mapping a composite attribute
(a) CUSTOMER entity type with composite attribute



(b) CUSTOMER relation with address detail

CUSTOMER

Customer_ID	Customer_Name	Street	City	State	Zip

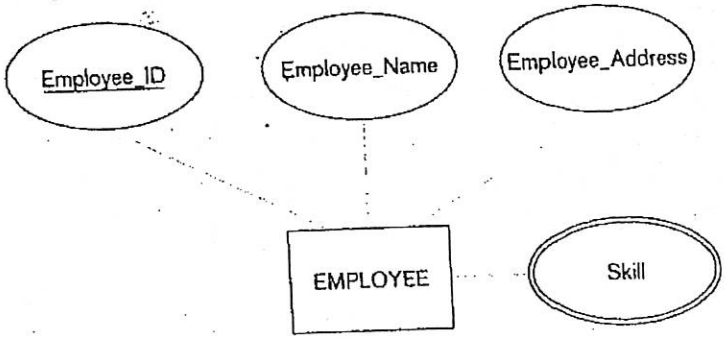


Figure 5-10
Mapping an entity with a multivalued attribute
(a) Employee entity type with multivalued attribute

(b) Mapping a multivalued attribute

EMPLOYEE

Employee_ID	Employee_Name	Employee_Address

EMPLOYEE_SKILL

Employee_ID	Skill

Step 2: Map Weak Entities

Recall that a weak entity type does not have an independent existence, but exists only through an identifying relationship with another entity type called the owner. A weak entity type does not have a complete identifier, but must have an attribute called a partial identifier that permits distinguishing the various occurrences of the weak entity for each owner entity instance.

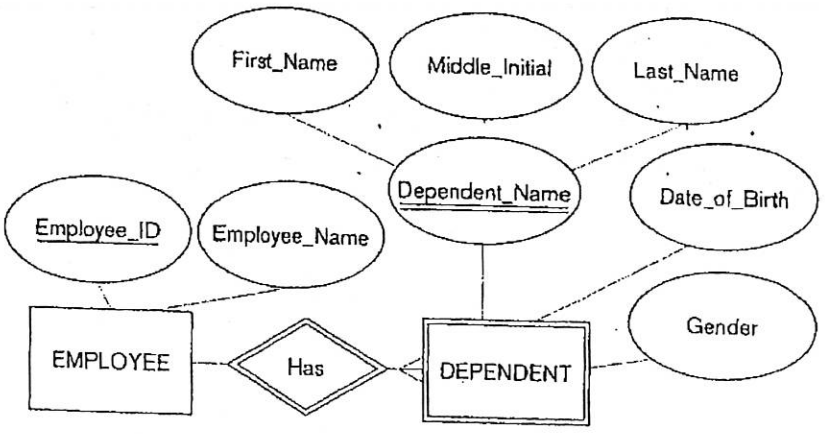
The following procedure assumes that you have already created a relation corresponding to the identifying entity type during Step 1. If you have not, you should create that relation now using the process described in Step 1.

For each weak entity type, create a new relation and include all of the simple attributes (or simple components of composite attributes) as attributes of this relation. Then include the primary key of the *identifying* relation as a foreign key attribute in this new relation. The primary key of the new relation is the combination of this primary key of the identifying and the partial identifier of the weak entity type.

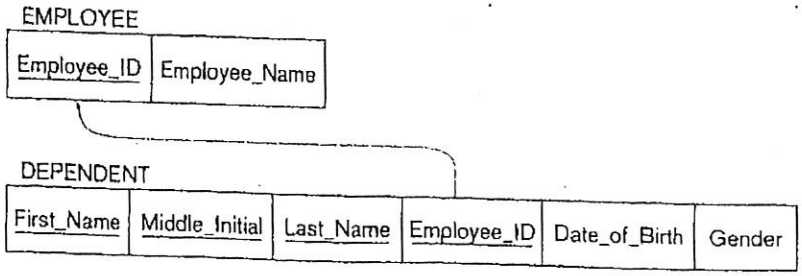
An example of this process is shown in Figure 5-11. Figure 5-11a shows the weak entity type DEPENDENT and its identifying entity type EMPLOYEE, linked by the identifying relationship Has (see Figure 3–5). Notice that the attribute Dependent_Name, which is the partial identifier for this relation, is a composite attribute with components First_Name, Middle_Initial, and Last_Name. Thus we assume that, *for a given employee*, these items will uniquely identify a dependent (a notable exception being the case of prizefighter George Foreman, who has named all his sons after himself!).

Figure 5-11b shows the two relations that result from mapping this E-R segment. The primary key of DEPENDENT consists of four attributes: Employee_ID, First_Name, Middle_Initial, and Last_Name. Date_of_Birth and Gender are the nonkey attributes. The foreign key relationship with its primary key is indicated by the arrow in the figure.

Figure 5-11
Example of mapping a weak entity
(a) Weak entity DEPENDENT



(b) Relations resulting from weak entity



Step 3: Map Binary Relationships

The procedure for representing relationships depends on both the degree of the relationships (unary, binary, ternary) and the cardinalities of the relationships. We describe and illustrate the important cases in the following discussion.

Map Binary One-to-Many Relationships For each binary 1:M relationship, first create a relation for each of the two entity types participating in the relationship, using the procedure described in Step 1. Next, include 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 (a mnemonic you can use to remember this rule is this: The primary key migrates to the many side).

To illustrate this simple process, we use the Submits relationship between customers and orders for Pine Valley Furniture Company (see Figure 3-22). This 1:M relationship is illustrated in Figure 5-12a. Figure 5-12b shows the result of applying the above rule to map the entity types with the 1:M relationship. The primary key Customer_ID of CUSTOMER (the one-side) is included as a foreign key in ORDER (the many-side). The foreign key relationship is indicated with an arrow.

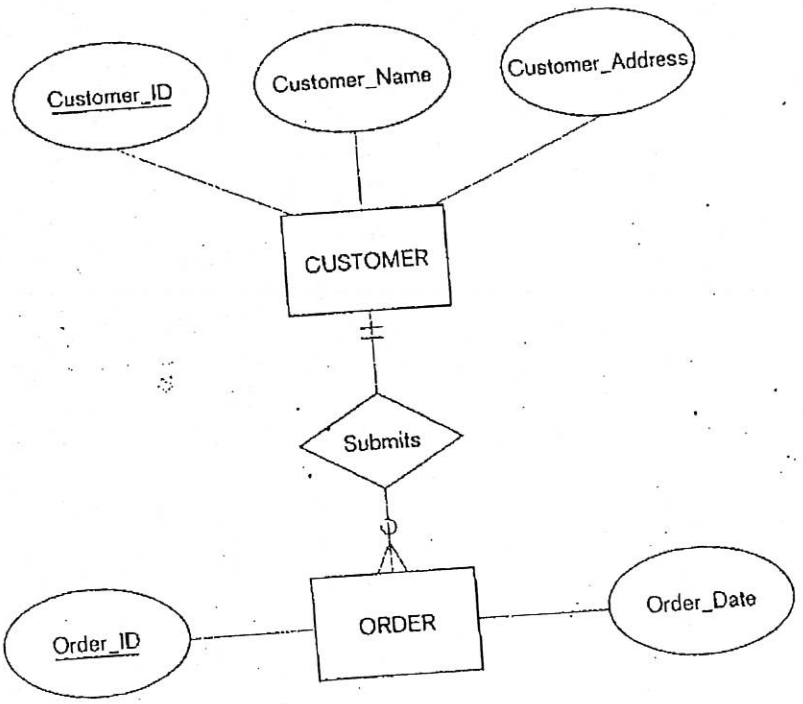
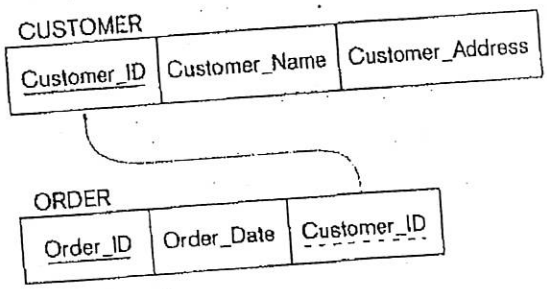


Figure 5-12
Example of mapping a 1:M relationship
(a) Relationship between customers and orders

(b) Mapping the relationship

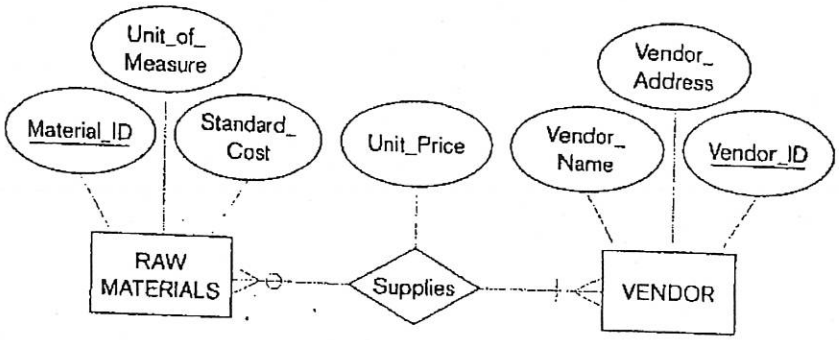


Map Binary Many-to-Many Relationships Suppose that there is a binary many-to-many (M:N) relationship between two entity types A and B. For such a relationship, we create a new relation C. Include as foreign key attributes in C the primary key for each of the two participating entity types. These attributes become the primary key of C. Any nonkey attributes that are associated with the M:N relationship are included with the relation C.

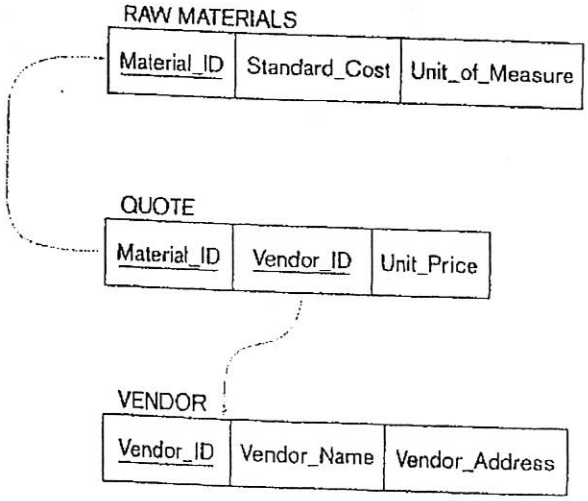
Figure 5-13 shows an example of applying this rule. Figure 5-13a shows the Supplies relationship between the entity types VENDOR and RAW MATERIALS for Pine Valley Furniture Company (introduced in Figure 3-22). Figure 5-13b shows the three relations (VENDOR, RAW MATERIALS, and QUOTE) that are formed from the entity types and the Supplies relationship. First, a relation is created for each of the two regular entity types VENDOR and RAW MATERIALS. Then a relation (named QUOTE in Figure 5-13b) is created for the Supplies relationship. The primary key of QUOTE is the combination of Vendor_ID and Material_ID, which are the respective primary keys of VENDOR and RAW MATERIALS. As indicated in the diagram, these attributes are foreign keys that "point to" the respective primary keys. The nonkey attribute Unit_Price also appears in QUOTE.

Map Binary One-to-One Relationships Binary one-to-one relationships can be viewed as a special case of one-to-many relationships. The process of mapping such a relationship to relations requires two steps. First, two relations are created, one for each of the participating entity types. Second, the primary key of one of the relations is included as a foreign key in the other relation.

Figure 5-13
Example of mapping an M:N relationship
(a) Requests relationship (M:N)



(b) Three resulting relations



In a 1:1 relationship, the association in one direction is nearly always optional one, while the association in the other direction is mandatory one (you can review the notation for these terms in Figure 3–1). You should include in the relation on the optional side of the relationship the foreign key of the entity type that has the mandatory participation in the 1:1 relationship. This approach will avoid the need to store null values in the foreign key attribute. Any attributes associated with the relationship itself are also included in the same relation as the foreign key.

An example of applying this procedure is shown in Figure 5-14. Figure 5-14a shows a binary 1:1 relationship between the entity types NURSE and CARE CENTER. Each care center must have a nurse who is in charge of that center. Thus the association from CARE CENTER to NURSE is a mandatory one, while the association from NURSE to CARE CENTER is an optional one (since any nurse may or may not be in charge of a care center). The attribute Date_Assigned is attached to the In_charge relationship.

The result of mapping this relationship to a set of relations is shown in Figure 5-14b. The two relations NURSE and CARE CENTER are created from the two entity types. Since CARE CENTER is the optional participant, the foreign key is placed in this relation. In this case the foreign key is called Nurse_in_Charge. It has the same domain as Nurse_ID, and the relationship with the primary key is shown in the figure. The attribute Date_Assigned is also located in CARE CENTER and would not be allowed to be null.

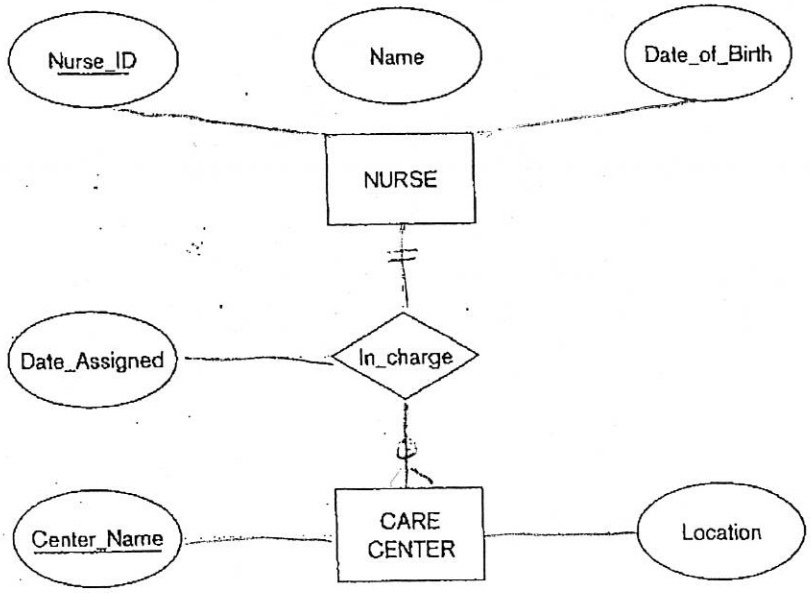


Figure 5-14
Mapping a binary 1:1 relationship
(a) Binary 1:1 relationship

(b) Resulting relations

NURSE

Nurse_ID	Name	Date_of_Birth

CARE CENTER

Center_Name	Location	Nurse_in_Charge	Date_Assigned

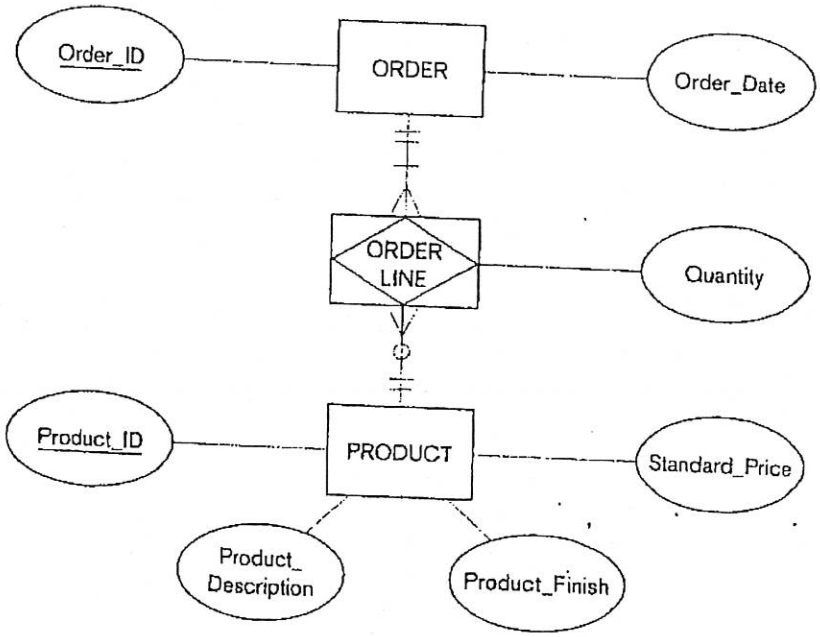
Step 4: Map Associative Entities

As explained in Chapter 3, when the data modeler encounters a many-to-many relationship, he or she may choose to model that relationship as an associative entity in the E-R diagram. This approach is most appropriate when the end user can best visualize the relationship as an entity type rather than as an M:N relationship. Mapping the associative entity involves essentially the same steps as mapping an M:N relationship, as described in Step 3.

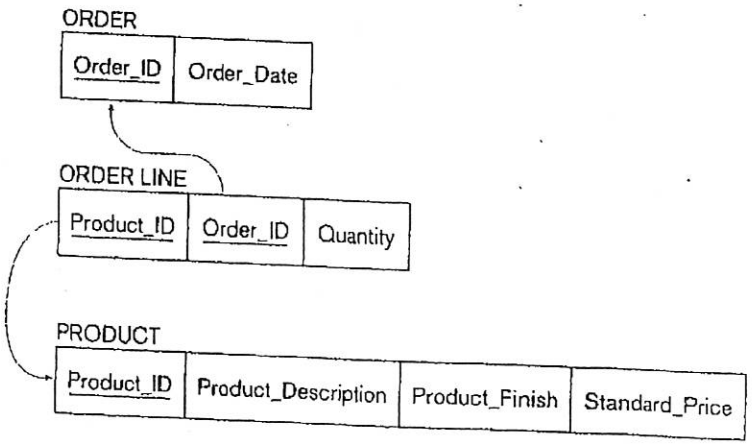
The first step is to create three relations: one for each of the two participating entity types, and the third for the associative entity. We refer to the relation formed from the associative entity as the *associative relation*. The second step then depends on whether on the E-R diagram an identifier was assigned to the associative entity.

Identifier Not Assigned If an identifier was not assigned, the default primary key for the associative relation consists of the two primary key attributes from the other two relations. These attributes are then foreign keys that reference the other two relations.

Figure 5-15
Mapping an associative entity
(a) An associative entity



(b) Three resulting relations



An example of this case is shown in Figure 5-15. Figure 5-15a shows the associative entity ORDER LINE that links the ORDER and PRODUCT entity types at Pine Valley Furniture Company (see Figure 3-22). Figure 5-15b shows the three relations that result from this mapping. Note the similarity of this example to that of an M:N relationship shown in Figure 5-13.

Identifier Assigned Sometimes the data modeler will assign an identifier (called a *surrogate* identifier or key) to the associative entity type on the E-R diagram. There are two reasons that may motivate this approach:

1. The associative entity type has a natural identifier that is familiar to end users.
2. The default identifier (consisting of the identifiers for each of the participating entity types) may not uniquely identify instances of the associative entity.

The process for mapping the associative entity in this case is now modified as follows. As before, a new (associative) relation is created to represent the associative entity. However, the primary key for this relation is the identifier assigned on the E-R diagram (rather than the default key). The primary keys for the two participating entity types are then included as foreign keys in the associative relation.

An example of this process is shown in Figure 5-16. Figure 5-16a shows the associative entity type SHIPMENT that links the CUSTOMER and VENDOR entity types. Shipment_No has been chosen as the identifier for SHIPMENT for two reasons:

1. Shipment_No is a natural identifier for this entity that is very familiar to end users.
2. The default identifier consisting of the combination of Customer_ID and Vendor_ID does not uniquely identify the instances of SHIPMENT. In fact, a

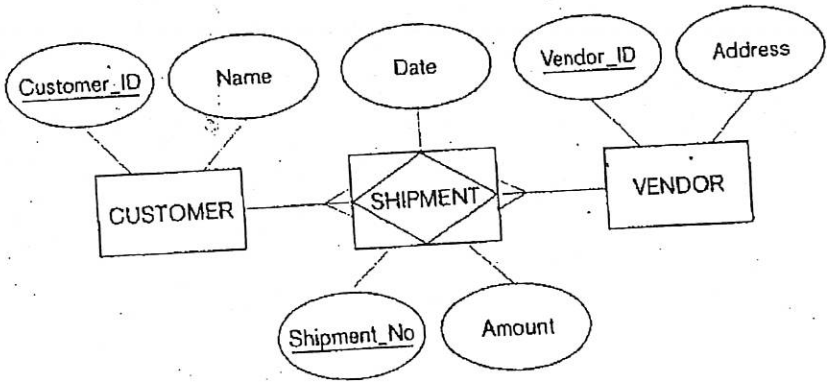
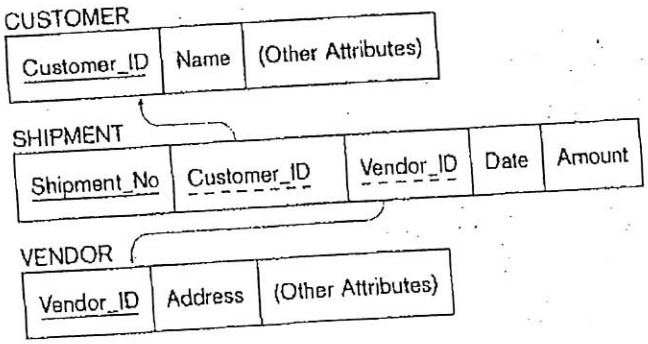


Figure 5-16
Mapping an associative entity with an identifier
(a) Associative entity (SHIPMENT)

(b) Three relations



given vendor will make many shipments to a given customer. Even including the attribute Date does not guarantee uniqueness, since there may be more than one shipment by a particular vendor on a given date, but the surrogate key Shipment_No will uniquely identify each shipment.

Two nonkey attributes associated with SHIPMENT are Date and Amount.

The result of mapping this entity to a set of relations is shown in Figure 5-16b. The new associative relation is named SHIPMENT. The primary key is Shipment_No. Customer_ID and Vendor_ID are included as foreign keys in this relation, and Date and Amount are nonkey attributes.

Step 3: Map Unary Relationships

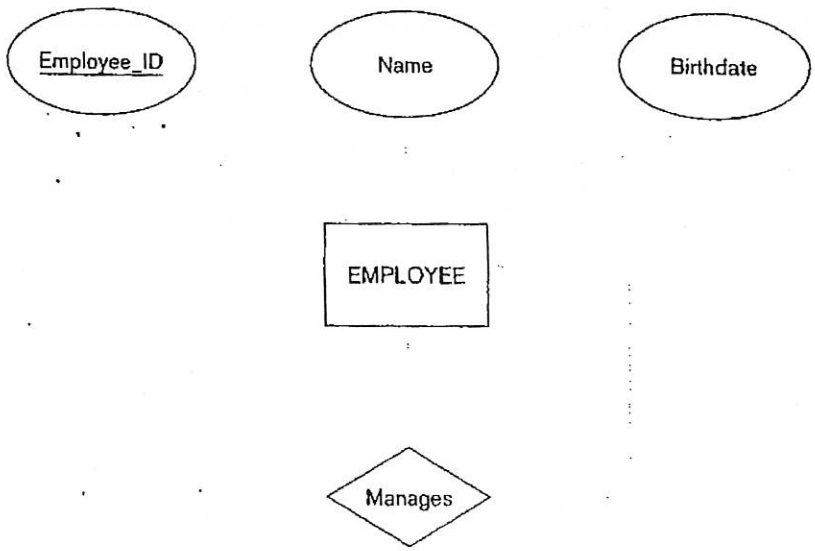
In Chapter 3 we defined a unary relationship as a relationship between the instances of a single entity type. Unary relationships are also called recursive relationships. The two most important cases of unary relationships are one-to-many and many-to-many. We discuss these two cases separately since the approach to mapping is somewhat different for the two types.

Unary One-to-Many Relationships The entity type in the unary relationship is mapped to a relation using the procedure described in Step 1. Then a foreign key attribute is added *within* the same relation that references the primary key values (this foreign key must have the same domain as the primary key). A **recursive foreign key** is a foreign key in a relation that references the primary key values of that same relation.

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

Figure 5-17a shows a unary one-to-many relationship named Manages that associates each employee of an organization with another employee who is his or her manager. Each employee has exactly one manager; a given employee may manage zero to many employees.

Figure 5-17
Mapping a unary 1:N relationship
(a) EMPLOYEE entity with Manages relationship



(b) EMPLOYEE relation with recursive foreign key

EMPLOYEE

Employee_ID	Name	Birthdate	Manager_ID

The EMPLOYEE relation that results from mapping this entity and relationship is shown in Figure 5-17b. The (recursive) foreign key in the relation is named Manager_ID. This attribute has the same domain as the primary key Employee_ID. Each row of this relation stores the following data for a given employee: Employee_ID, Name, Birthdate, and Manager_ID (that is, the Employee_ID for this employee's manager). Notice that since it is a foreign key, Manager_ID references Employee_ID.

Unary Many-to-Many Relationships With this type of relationship, two relations are created: one to represent the entity type in the relationship and the other an associative relation to represent the M:N relationship itself. The primary key of the associative relation consists of two attributes. These attributes (which need not have the same name) both take their values from the primary key of the other relation. Any nonkey attribute of the relationship is included in the associative relation.

An example of mapping a unary M:N relationship is shown in Figure 5-18. Figure 5-18a shows a bill-of-materials relationship among items that are assembled from other items or components (this structure was described in Chapter 3 and an example appears in Figure 3–13). The relationship (called Contains) is M:N since a given item can contain numerous component items, and conversely an item can be used as a component in numerous other items.

The relations that result from mapping this entity and its relationship are shown in Figure 5-18b. The ITEM relation is mapped directly from the same entity type. COMPONENT is an associative relation whose primary key consists of two attributes that are arbitrarily named Item_No and Component_No. The attribute Quantity is a nonkey attribute of this relation that for a given item, records the quantity of a particular component item used in that item. Notice that both Item_No and Component_No reference the primary key (Item_No) of the ITEM relation.

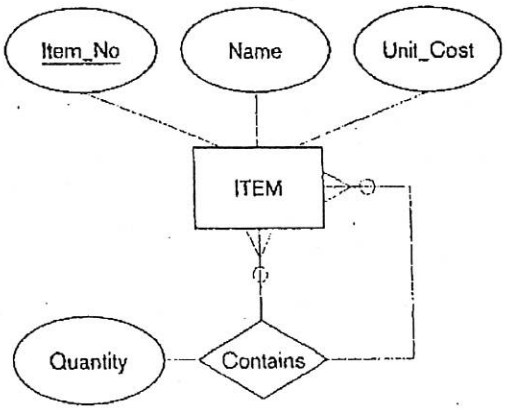
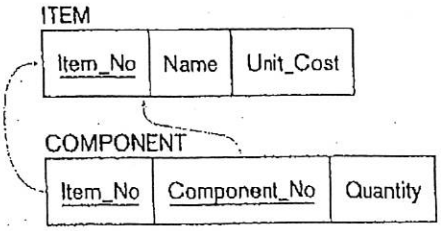


Figure 5-18
Mapping a unary M:N relationship
(a) Bill-of-materials relationships (M:N)

(b) ITEM and COMPONENT relations



We can easily query the above relations to determine (for example) the components of a given item. The following SQL query will list the immediate components (and their quantity) for item number 100:

```
SELECT Component_No, Quantity
FROM COMPONENT
WHERE Item_No = 100;
```

Step 6: Map Ternary (and *n*-ary) Relationships

Recall from Chapter 3 that a ternary relationship is a relationship among three entity types. In that chapter we recommended that you convert a ternary relationship to an associative entity in order to represent participation constraints more accurately.

To map an associative entity type that links three regular entity types, we create a new associative relation. The default primary key of this relation consists of the three primary key attributes for the participating entity types (in some cases, additional attributes are required to form a unique primary key). These attributes then act in the role of foreign keys that reference the individual primary keys of the participating entity types. Any attributes of the associative entity type become attributes of the new relation.

An example of mapping a ternary relationship (represented as an associative entity type) is shown in Figure 5-19. Figure 5-19a is an E-R segment (or view) that represents a *patient* receiving a *treatment* from a *physician*. The associative entity type PATIENT TREATMENT has the attributes Date, Time, and Results; values are recorded for these attributes for each instance of PATIENT TREATMENT.

The result of mapping this view is shown in Figure 5-19b. The primary key attributes Patient_ID, Physician_ID, and Treatment_Code become foreign keys in PATIENT TREATMENT. These attributes are components of the primary key of PATIENT TREATMENT. However, they do not uniquely identify a given treatment, since a patient may receive the same treatment from the same physician on more than one occasion. Does including the attribute Date as part of the primary key (along with the other three attributes) result in a primary key? This would be so if a given patient receives only one treatment from a particular physician on a given date. However, this is not likely to be the case. For example, a patient may receive a treatment in the morning, then the same treatment in the afternoon. To resolve this issue, we include Time as part of the primary key. Therefore, the primary key of PATIENT TREATMENT consists of the five attributes shown in Figure 5-19b: Patient_ID, Physician_ID, Treatment_Code, Date, and Time. The only nonkey attribute in the relation is Results.

Step 7: Map Supertype/Subtype Relationships

The relational data model does not yet directly support supertype/subtype relationships. Fortunately, there are various strategies that database designers can use to represent these relationships with the relational data model (Chouinard, 1989). For our purposes we use the following strategy, which is the one most commonly employed:

1. Create a separate relation for the supertype and for each of its subtypes.
2. Assign to the relation created for the supertype the attributes that are common to all members of the supertype, including the primary key.
3. Assign to the relation for each subtype the primary key of the supertype, and only those attributes that are unique to that subtype.
4. Assign one (or more) attributes of the supertype to function as the subtype discriminator (the role of the subtype discriminator was discussed in Chapter 4).

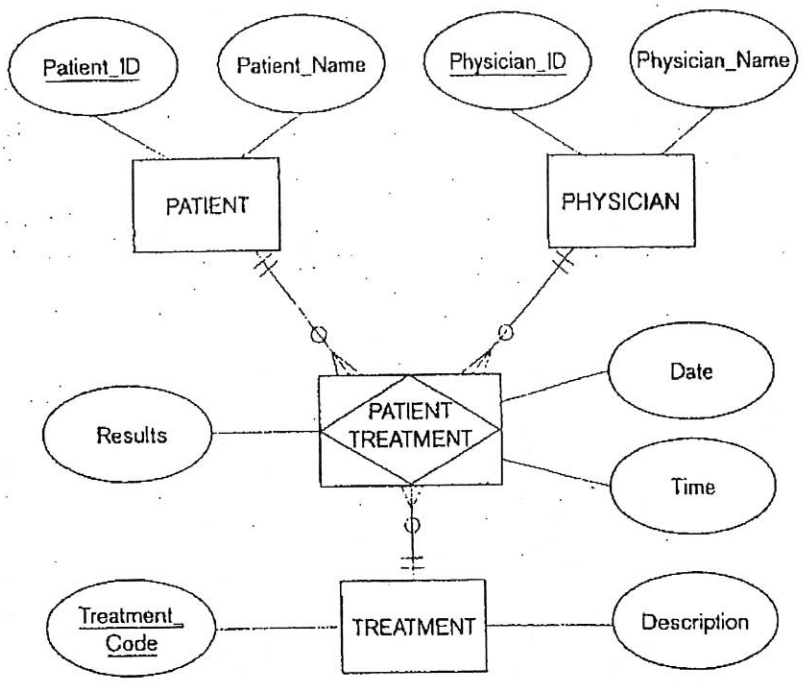
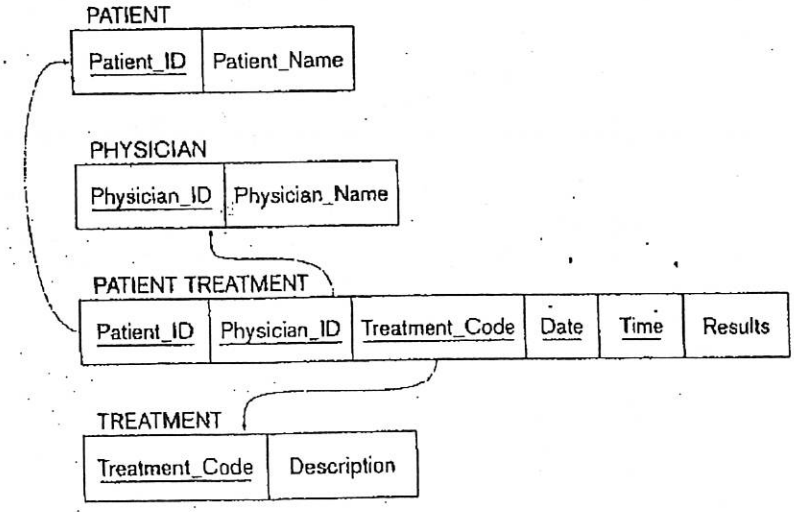


Figure 5-19
Mapping a ternary relationship
(a) Ternary relationship with associative entity

(b) Mapping the ternary relationship



An example of applying this procedure is shown in Figures 5-20 and 5-21. Figure 5-20 shows the supertype EMPLOYEE with subtypes HOURLY EMPLOYEE, SALARIED EMPLOYEE, and CONSULTANT (this example is described in Chapter 4, and Figure 5-20 is a repeat of Figure 4–8). The primary key of EMPLOYEE is Employee_Number, and the attribute Employee_Type is the subtype discriminator.

The result of mapping this diagram to relations using the above rules is shown in Figure 5-21. There is one relation for the supertype (EMPLOYEE) and one for each of the three subtypes. The primary key for each of the four relations is Employee_Number. A prefix is used to distinguish the name of the primary key for each subtype. For example, S_Employee_Number is the name for the primary key of

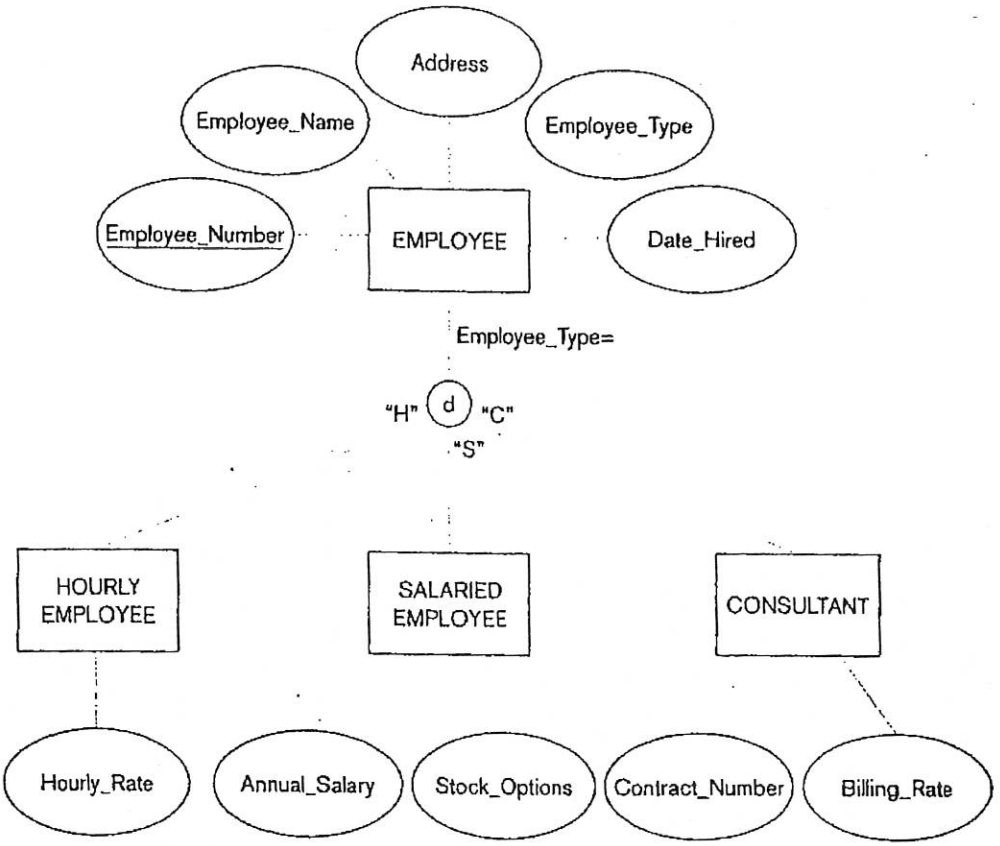


Figure 5-20
Supertype/subtype relationships

Figure 5-21
Mapping supertype/subtype relationships to relations

EMPLOYEE

Employee_Number	Employee_Name	Address	Employee_Type	Date_Hired

HOURLY_EMPLOYEE

H_Employee_Number	Hourly_Rate

SALARIED_EMPLOYEE

S_Employee_Number	Annual_Salary	Stock_Options

CONSULTANT

C_Employee_Number	Contract_Number	Billing_Rate

the relation SALARIED_EMPLOYEE. Each of these attributes is a foreign key that references the supertype primary key, as indicated by the arrows in the diagram. Each subtype relation contains only those attributes peculiar to the subtype.

For each subtype, a relation can be produced that contains *all* of the attributes of that subtype (both specific and inherited) by using an SQL command that joins the subtype with its supertype. For example, suppose that we want to display a table that contains all of the attributes for SALARIED_EMPLOYEE. The following command is used:

```
SELECT *
FROM EMPLOYEE, SALARIED_EMPLOYEE
WHERE Employee_Number = S_Employee_Number;
```

Although you have not yet formally studied such commands, you can intuitively see that this command will join the two tables and produce a larger table that contains all of the attributes from both tables.

INTRODUCTION TO NORMALIZATION

Normalization is a formal process for deciding which attributes should be grouped together in a relation. In Chapters 3 and 4 you used common sense to group attributes into entity types during conceptual data modeling. In the previous section of this chapter you learned how to map E-R diagrams to relations. Before proceeding with physical design, we need a method to validate the logical design to this point. Normalization is primarily a tool to validate and improve a logical design, so that it satisfies certain constraints that avoid unnecessary duplication of data.

We have presented an intuitive discussion of well-structured relations; however, we need formal definitions of such relations, together with a process for designing them. **Normalization** is the process of decomposing relations with anomalies to produce smaller, well-structured relations. For example, we used the principles of normalization to convert the EMPLOYEE2 table (with its redundancy) to EMPLOYEE1 (Figure 5-1) and EMP_COURSE (Figure 5-7).

Normalization: The process of decomposing relations with anomalies to produce smaller, well-structured relations.

Steps in Normalization

Normalization can be accomplished and understood in stages, each of which corresponds to a normal form (see Figure 5-22). A **normal form** is a state of a relation that results from applying simple rules regarding functional dependencies (or relationships between attributes) to that relation. We describe these rules briefly in this section and illustrate them in detail in the following sections.

Normal form: A state of a relation that results from applying simple rules regarding functional dependencies (or relationships between attributes) to that relation.

1. *First normal form* Any multivalued attributes (also called repeating groups) have been removed, so there is a single value (possibly null) at the intersection of each row and column of the table (as in Figure 5-2b).
2. *Second normal form* Any partial functional dependencies have been removed.
3. *Third normal form* Any transitive dependencies have been removed.
4. *Boyce/Codd normal form* Any remaining anomalies that result from functional dependencies have been removed.
5. *Fourth normal form* Any multivalued dependencies have been removed.
6. *Fifth normal form* Any remaining anomalies have been removed.

We describe and illustrate first through third normal forms in this chapter. The remaining normal forms are described in Appendix B.

the relation SALARIED_EMPLOYEE. Each of these attributes is a foreign key that references the supertype primary key, as indicated by the arrows in the diagram. Each subtype relation contains only those attributes peculiar to the subtype.

For each subtype, a relation can be produced that contains *all* of the attributes of that subtype (both specific and inherited) by using an SQL command that joins the subtype with its supertype. For example, suppose that we want to display a table that contains all of the attributes for SALARIED_EMPLOYEE. The following command is used:

```
SELECT *
FROM EMPLOYEE, SALARIED_EMPLOYEE
WHERE Employee_Number = S_Employee_Number;
```

Although you have not yet formally studied such commands, you can intuitively see that this command will join the two tables and produce a larger table that contains all of the attributes from both tables.

INTRODUCTION TO NORMALIZATION

Normalization is a formal process for deciding which attributes should be grouped together in a relation. In Chapters 3 and 4 you used common sense to group attributes into entity types during conceptual data modeling. In the previous section of this chapter you learned how to map E-R diagrams to relations. Before proceeding with physical design, we need a method to validate the logical design to this point. Normalization is primarily a tool to validate and improve a logical design, so that it satisfies certain constraints that avoid unnecessary duplication of data.

We have presented an intuitive discussion of well-structured relations; however, we need formal definitions of such relations, together with a process for designing them. **Normalization** is the process of decomposing relations with anomalies to produce smaller, well-structured relations. For example, we used the principles of normalization to convert the EMPLOYEE2 table (with its redundancy) to EMPLOYEE1 (Figure 5-1) and EMP_COURSE (Figure 5-7).

Normalization: The process of decomposing relations with anomalies to produce smaller, well-structured relations.

Steps in Normalization

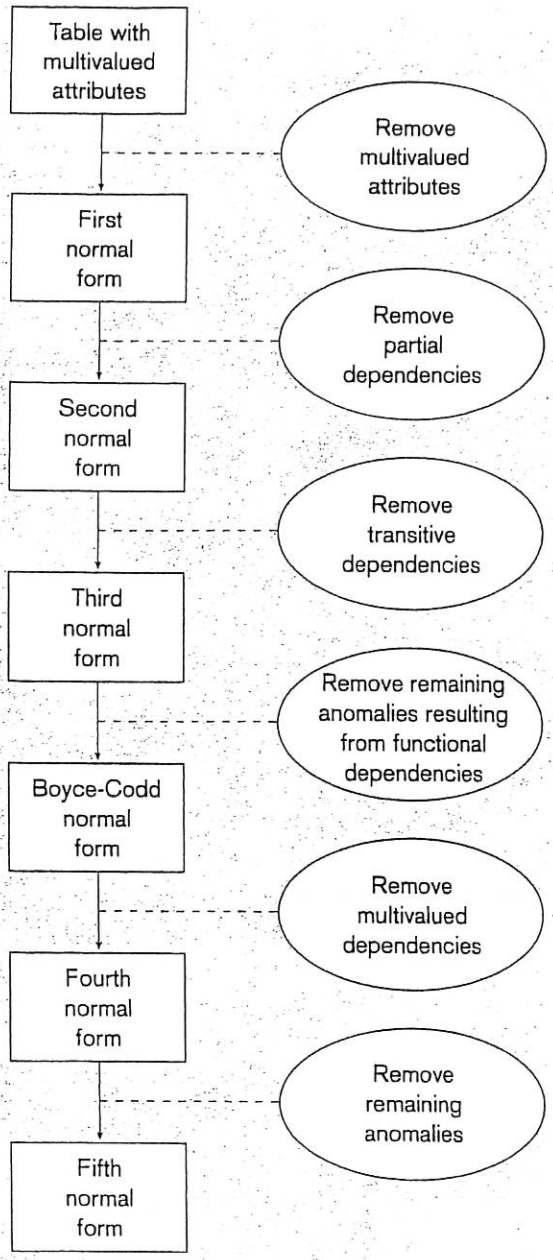
Normalization can be accomplished and understood in stages, each of which corresponds to a normal form (see Figure 5-22). A **normal form** is a state of a relation that results from applying simple rules regarding functional dependencies (or relationships between attributes) to that relation. We describe these rules briefly in this section and illustrate them in detail in the following sections.

Normal form: A state of a relation that results from applying simple rules regarding functional dependencies (or relationships between attributes) to that relation.

1. *First normal form* Any multivalued attributes (also called repeating groups) have been removed, so there is a single value (possibly null) at the intersection of each row and column of the table (as in Figure 5-2b).
2. *Second normal form* Any partial functional dependencies have been removed.
3. *Third normal form* Any transitive dependencies have been removed.
4. *Boyce/Codd normal form* Any remaining anomalies that result from functional dependencies have been removed.
5. *Fourth normal form* Any multivalued dependencies have been removed.
6. *Fifth normal form* Any remaining anomalies have been removed.

We describe and illustrate first through third normal forms in this chapter. The remaining normal forms are described in Appendix B.

Figure 5-22
Steps in normalization



Functional Dependencies and Keys

Functional dependency: A constraint between two attributes or two sets of attributes.

Normalization is based on the analysis of functional dependencies. A **functional dependency** is a constraint between two attributes or two sets of attributes. For any relation R, attribute B is functionally dependent on attribute A if, for every valid instance of A, that value of A uniquely determines the value of B (Dutka and Hanson, 1989). The functional dependency of B on A is represented by an arrow, as follows: A → B. 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_Title,Date_Completed) shown in Figure 5-7. We represent the functional dependency in this relation as follows:

Emp_ID,Course_Title → Date_Completed.

↓
Page 175

The functional dependency in this statement implies that the date a course is completed is completely determined by the identity of the employee and the title of the course. Common examples of functional dependencies are the following:

1. SSN → Name, Address, Birthdate A person's name, address, and birthdate are functionally dependent on that person's Social Security number.
2. VIN → Make, Model, Color The make, model, and color of a vehicle are functionally dependent on the vehicle identification number.
3. ISBN → Title, First_Author_Name The title of a book and the name of the first author are functionally dependent on the book's International Standard Book Number (ISBN).

Determinants The attribute on the left-hand side of the arrow in a functional dependency is called a **determinant**. SSN, VIN, and ISBN are determinants (respectively) in the preceding three examples. In the EMP_COURSE relation (Figure 5-7) the combination of Emp_ID and Course_Title is a determinant.

Determinant: The attribute on the left-hand side of the arrow in a functional dependency.

Candidate Keys A **candidate key** is an attribute, or combination of attributes, that uniquely identifies a row in a relation. A candidate key must satisfy the following properties (Dutka and Hanson, 1989), which are a subset of the six properties of a primary key previously listed:

Candidate key: An attribute, or combination of attributes, that uniquely identifies a row in a relation.

1. *Unique identification* For every row, the value of the key must uniquely identify that row. *This property implies that each nonkey attribute is functionally dependent on that key.*
2. *Nonredundancy* No attribute in the key can be deleted without destroying the property of unique identification.

Let's apply the preceding definition to identify candidate keys in two of the relations described in this chapter. The EMPLOYEE1 relation (Figure 5-1) has the following schema: EMPLOYEE1(Emp_ID,Name,Dept_Name,Salary). Emp_ID is the only determinant in this relation. All of the other attributes are functionally dependent on Emp_ID. Therefore Emp_ID is a candidate key and (since there are no other candidate keys) also is the primary key.

We represent the functional dependencies for a relation using the notation shown in Figure 5-23. Figure 5-23a shows the representation for EMPLOYEE1. The horizontal line in the figure portrays the functional dependencies. A vertical line drops from the primary key (Emp_ID) and connects to this line. Vertical arrows then point to each of the nonkey attributes that are functionally dependent on the primary key.

For the relation EMPLOYEE2 (Figure 5-2b), notice that (unlike EMPLOYEE1), Emp_ID does not uniquely identify a row in the relation. For example, there are two rows in the table for Emp_ID number 100. There are two functional dependencies in this relation:

1. Emp_ID → Name, Dept_Name, Salary
2. Emp_ID, Course_Title → Date_Completed

The functional dependencies indicate that the combination of Emp_ID and Course_Title is the only candidate key (and therefore the primary key) for EMPLOYEE2. In other words, the primary key of EMPLOYEE2 is a composite key. Neither Emp_ID nor Course_Title uniquely identifies a row in this relation, and therefore property 1 cannot by itself be a candidate key. Examine the data in Figure 5-2b to verify that the combination of Emp_ID and Course_Title does uniquely identify each row of EMPLOYEE2. We represent the functional dependencies in this relation in Figure 5-23b. Notice that Date_Completed is the only attribute that is functionally dependent on the full primary key consisting of the attributes Emp_ID and Course_Title.

Figure 5-2
Eliminating multivalued attributes
(a) Table with repeating groups

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
				Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000.	SPSS	1/12/200X
				C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/16/200X
				Java	8/12/200X

(b) EMPLOYEE2 relation

EMPLOYEE2

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
100	Margaret Simpson	Marketing	48,000	Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000	SPSS	1/12/200X
110	Chris Lucero	Info Systems	43,000	C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/19/200X
150	Susan Martin	Marketing	42,000	Java	8/12/200X

Figure 5-1
EMPLOYEE1 relation with sample data

EMPLOYEE1

Emp_ID	Name	Dept_Name	Salary
100	Margaret Simpson	Marketing	48,000
140	Allen Beeton	Accounting	52,000
110	Chris Lucero	Info Systems	43,000
190	Lorenzo Davis	Finance	55,000
150	Susan Martin	Marketing	42,000

5-1 are intended to illustrate the structure of the EMPLOYEE1 relation; they are not part of the relation itself. Even if we add another row of data to the figure, it is still the same EMPLOYEE1 relation. Nor does deleting a row change the relation. In fact, we could delete *all* of the rows shown in Figure 5-1, and the EMPLOYEE1 relation would still exist. Stated differently, Figure 5-1 is an instance of the EMPLOYEE1 relation.

We 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 that relation. For EMPLOYEE1 we would have:

EMPLOYEE1 (Emp_ID,Name,Dept_Name,Salary).

Relational Keys We must be able to store and retrieve a row of data in a relation, n the data values stored in that row. To achieve this goal, every relation must hav. . primary key. A **primary key** is an attribute (or combination of attributes) that uniquely identifies each row in a relation. We designate a primary key by underlining ... attribute name. For example, the primary key for the relation EMPLOYEE1 is

Primary key: An attribute (or combination of attributes) that uniquely identifies each row in a relation.

tional databases, it is important that you understand how to eliminate multivalued attributes.

We illustrated a table with repeating groups in Figure 5-2a and then converted it to the relation EMPLOYEE2 in Figure 5-2b by removing the multivalued attributes. Thus EMPLOYEE2 is a relation in first normal form. A table with multivalued attributes is converted to a relation in first normal form by extending the data in each column to fill cells that are empty because of the multivalued attributes.

Second Normal Form

Second normal form: A relation in first normal form in which every nonkey attribute is fully functionally dependent on the primary key.

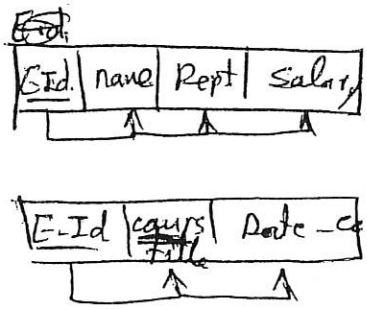
A relation is in **second normal form** (2NF) if it is in first normal form and every nonkey attribute is fully functionally dependent on the primary key. Thus no nonkey attribute is functionally dependent on part (but not all) of the primary key. A relation that is in first normal form will be in second normal form if any-one of the following conditions applies:

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

EMPLOYEE2 (Figure 5-2b) is an example of a relation that is not in second normal form. The primary key for this relation is the composite key Emp_ID, Course_Title. Therefore the nonkey attributes Name, Dept_Name, and Salary are functionally dependent on part of the primary key (Emp_ID) but not on Course_Title. These dependencies are shown graphically in Figure 5-23b.

Partial functional dependency: A functional dependency in which one or more nonkey attributes are functionally dependent on part (but not all) of the primary key.

A **partial functional dependency** is a functional dependency in which one or more nonkey attributes (such as Name) are functionally dependent on part (but not all) of the primary key. The partial functional dependency in EMPLOYEE2 creates redundancy in that relation, which results in anomalies when the table is updated as we noted in a previous section.



To convert a relation to second normal form, we decompose the relation into new relations that satisfy one (or more) of the conditions described above. EMPLOYEE2 is decomposed into the following two relations:

1. EMPLOYEE1(Emp_ID,Name,Dept_Name, Salary) This relation satisfies condition 1 above and is in second normal form (sample data are shown in Figure 5-1).
2. EMP_COURSE(Emp_ID,Course_Title,Date_Completed) This relation satisfies property 3 above and is also in second normal form (sample data appear in Figure 5-7).

You should examine these new relations to verify that they are free of the anomalies associated with EMPLOYEE2.

Third Normal Form

Third normal form: A relation that is in second normal form and has no transitive dependencies present.

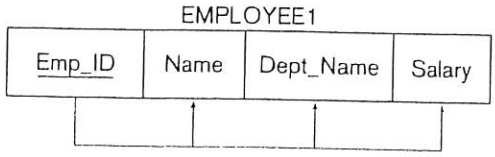
Transitive dependency: A functional dependency between two (or more) nonkey attributes.

A relation is in **third normal form** (3NF) if it is in second normal form and no transitive dependencies exist. A **transitive dependency** in a relation is a functional dependency between two (or more) nonkey attributes. For example, consider the relation

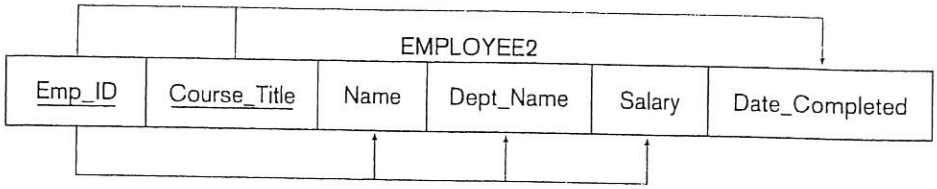
SALES(Cust_ID,Name,Salesperson,Region)

(Sample data for this relation appear in Figure 5-24a.)

Figure 5-23
Representing functional dependencies
(a) Functional dependencies in EMPLOYEE1



(b) Functional dependencies in EMPLOYEE2



We can summarize the relationship between determinants and candidate keys as follows. A candidate key is always a determinant, while a determinant may or may not be a candidate key. For example, in EMPLOYEE2, Emp_ID is a determinant but not a candidate key. A candidate key is a determinant that uniquely identifies the remaining (nonkey) attributes in a relation. A determinant may be a candidate key (such as Emp_ID in EMPLOYEE1), part of a composite candidate key (such as Emp_ID in EMPLOYEE2), or a nonkey attribute. We will describe examples of this shortly.

THE BASIC NORMAL FORMS

Now that we have examined functional dependencies and keys, we are ready to describe and illustrate first through third normal forms. We also describe the normalization of summary data that appear in information bases.

First normal form: A relation that contains no multivalued attributes.

First Normal Form

A relation is in **first normal form** (1NF) if it contains no multivalued attributes. Recall that the first property of a relation is that the value at the intersection of each row and column must be atomic. Thus a table that contains multivalued attributes or repeating groups is not a relation.

In the previous section, in describing how to map E-R diagrams to relations, we described a procedure for removing multivalued attributes from entity types on the E-R diagram. Thus if you developed a logical design by transforming E-R diagrams to relations, there should not be any multivalued attributes remaining. However, many older legacy systems written in languages such as COBOL supported multivalued attributes. Since you may participate in efforts to convert these older systems to rela-

Figure 5-2
Eliminating multivalued attributes
(a)Table with repeating groups

Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
				Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000.	SPSS	1/12/200X
				C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/16/200X
				Java	8/12/200X

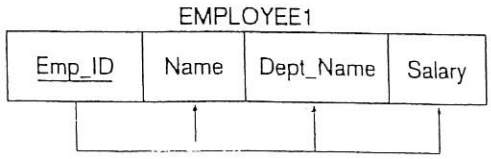
(b) EMPLOYEE2 relation

EMPLOYEE2

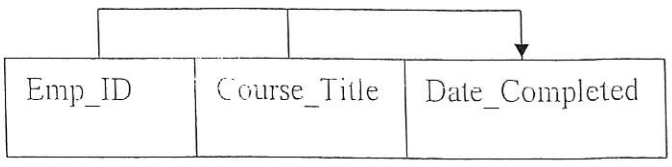
Emp_ID	Name	Dept_Name	Salary	Course_Title	Date_Completed
100	Margaret Simpson	Marketing	48,000	SPSS	6/19/200X
100	Margaret Simpson	Marketing	48,000	Surveys	10/7/200X
140	Alan Beeton	Accounting	52,000	Tax Acc	12/8/200X
110	Chris Lucero	Info Systems	43,000	SPSS	1/12/200X
110	Chris Lucero	Info Systems	43,000	C++	4/22/200X
190	Lorenzo Davis	Finance	55,000		
150	Susan Martin	Marketing	42,000	SPSS	6/19/200X
150	Susan Martin	Marketing	42,000	Java	8/12/200X

Figure 5-23
Representing functional dependencies
(a) Functional dependencies in EMPLOYEE1

1st → 2nd Normal Form



(b) Functional dependencies EMPLOYEE2



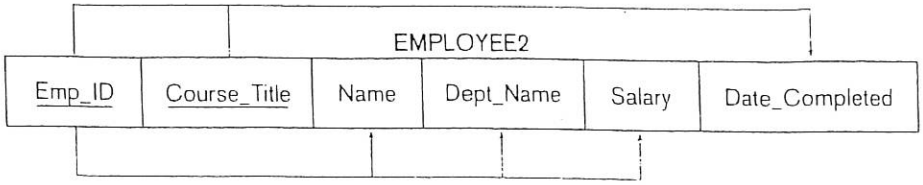


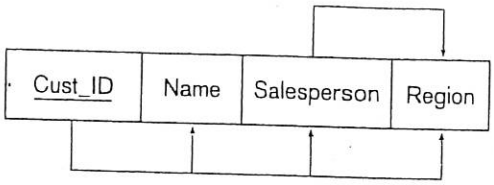
Figure 5-24
Relation with transitive dependency
(a) SALES relation with sample data

should not be underlined

SALES

Cust_ID	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
—	—	Robinson	North

(b) Transitive dependency in SALES relation



The functional dependencies in the SALES relation are shown graphically in Figure 5-24b. Cust_ID is the primary key, so that all of the remaining attributes are functionally dependent on this attribute. However, there is a transitive dependency (Region is functionally dependent on Salesperson and Salesperson is functionally dependent on Cust_ID). As a result, there are update anomalies in SALES.

Anomalies

1. *Insertion anomaly* A new salesperson (Robinson) assigned to the North region cannot be entered until a customer has been assigned to that salesperson (since a value for Cust_ID must be provided to insert a row in the table).
2. *Deletion anomaly* If customer number 6837 is deleted from the table, we lose the information that salesperson Hernandez is assigned to the East region.
3. *Modification anomaly* If salesperson Smith is reassigned to the East region, several rows must be changed to reflect that fact (two rows are shown in Figure 5-24a).

These anomalies arise as a result of the transitive dependency. The transitive dependency can be removed by decomposing SALES into two relations, as shown in Figure 5-25a. Note that Salesperson, which is the determinant in the transitive dependency in SALES, becomes the primary key in SPERSON. Salesperson[1] becomes a foreign key in SALES1. Often the normalized relations of Figure 5-25 will also use less storage space when implemented than will a database based on the relation in Figure 5-24a because the dependent data (Region and any other salesperson data) do not have to repeat for each customer. Thus, normalization, besides solving anomalies, also often reduces redundant data storage.

[1]Because Salesperson is an intelligent business key, it is a poor choice for a primary key. This was briefly explained in Chapter 3, and we elaborate on this issue later in this chapter.

SALES1

Cust_ID	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

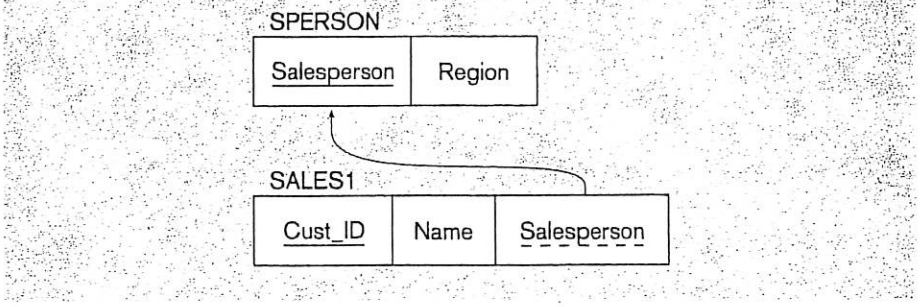


Figure 5-25
Removing a transitive dependency
(a) Decomposing the SALES relation

(b) Relations in 3NF

As shown in Figure 5-25b, the new relations are now in third normal form, since no transitive dependencies exist. You should verify that the anomalies that exist in SALES are not present in SALES1 and SPERSON.

Transitive dependencies may also occur between sets of attributes in a relation. For example, the relation SHIPMENT (Snum,Origin,Destination,Distance) could be used to record shipments according to origin, destination, and distance (Dutka and Hanson, 1989). Sample data for this relation appear in Figure 5-26a. The functional dependencies in the SHIPMENT relation are shown in Figure 5-26b. The primary key of the SHIPMENT relation is the attribute Snum (for Shipment Number). As a result, we know that the relation is in second normal form (why?). However, there is a transitive dependency in this relation: The Distance attribute is functionally dependent on the *pair* of nonkey attributes Origin and Destination. As a result there are anomalies in SHIPMENT (as an exercise, you should examine Figure 5-26a and identify insertion, deletion, and modification anomalies). We can remove the transitive dependency in SHIPMENT by decomposing it into two relations (both in 3NF):

SHIPTO(Snum,Origin,Destination)

DISTANCES(Origin,Destination,Distance)

The first of these relations provides the origin and destination for any given shipment, while the second provides the distance between an origin and destination pair. Sample data for these relations appear in Figure 5-26c.

Normalizing Summary Data

Databases to support managerial decision making in an organization often contain subsets and summaries of data from operational databases. These "data warehouses" used to support higher levels of management are often normalized (at least to a certain extent) to avoid the same anomalies that arise in operational databases. We describe the issues associated with normalizing data in data warehouses in Chapter 11.

Figure 5-24
Relation with transitive dependency
(a) SALES relation with sample data

should not be underlined

2nd → 3rd Normal Form

SALES

Cust_ID	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
—	—	Robinson	North

(b) Transitive dependency in SALES relation

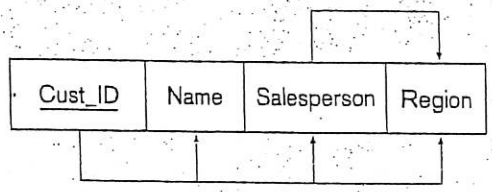


Figure 5-25
Removing a transitive dependency
(a) Decomposing the SALES relation

SALES1

Cust_ID	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

(b) Relations in 3NF

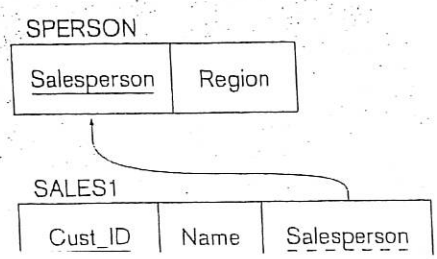


Figure 5-26
Another example of transitive dependencies

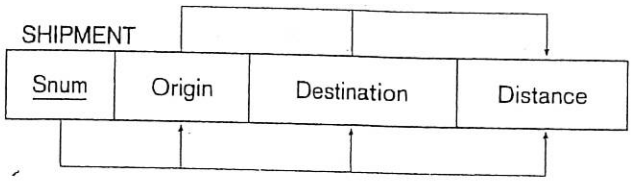
(a) SHIPMENT relation with sample data

2nd → 3rd Normal Form

SHIPMENT

Snum	Origin	Destination	Distance
409	Seattle	Denver	1,537
618	Chicago	Dallas	1,058
723	Boston	Atlanta	1,214
824	Denver	Los Angeles	1,150
629	Minneapolis	St. Louis	587

(b) Functional dependencies in SHIPMENT



(c) Relations in 3NF

SHIPTO

Snum	Origin	Destination
409	Seattle	Denver
618	Chicago	Dallas
723	Boston	Atlanta
824	Denver	Los Angeles
629	Minneapolis	St. Louis

DISTANCES

Origin	Destination	Distance
Seattle	Denver	1,537
Chicago	Dallas	1,058
Boston	Atlanta	1,214
Denver	Los Angeles	1,150
Minneapolis	St. Louis	587

MERGING RELATIONS

In a previous section, we described how to transform E-R diagrams to relations. We then described how to check the resulting relations to determine whether they are in third normal form, and perform normalization steps if necessary.

As part of the logical design process, normalized relations may have been created from a number of separate E-R diagrams and (possibly) other user views. Some of the relations may be redundant; that is, they may refer to the same entities. If so, we should merge those relations to remove the redundancy. This section describes merging relations (also called *view integration*). An understanding of how to merge relations is important for three reasons:

Why we need to Merge.

1. On large projects, the work of several subteams comes together during logical design, so there is a need to merge relations.
2. Integrating existing databases with new information requirements often leads to the need to integrate different views.
3. New data requirements may arise during the life cycle, so there is a need to merge any new relations with what has already been developed.

An Example

Suppose that modeling a user view results in the following 3NF relation:

EMPLOYEE1 (Employee_ID,Name,Address,Phone)

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

EMPLOYEE2 (Employee_ID,Name,Address,Jobcode,No_Years)

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

EMPLOYEE (Employee_ID,Name,Address,Phone,Jobcode,No_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 as in the preceding example, the database analyst 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 briefly illustrate four problems that arise in view integration: *synonyms, homonyms, transitive dependencies,* and *supertype/subtype relationships.*

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

Synonyms: Two (or more) attributes having different names but the same meaning, as when they describe the same characteristic of an entity.

STUDENT1 (Student_ID,Name)

STUDENT2 (Matriculation_No,Name,Address)

In this case, the analyst recognizes that both the Student_ID and Matriculation_No are synonyms for a person's Social Security number and are identical attributes. (Another possibility is that these are both candidate keys, and only one of them should be selected as the primary keys.) 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)

Often when there are synonyms, there is a need to allow some database users to refer to the same data by different names. Users may need to use familiar names that are consistent with terminology in their part of the organization. An **alias** is an alternative name used for an attribute. Many database management systems allow the definition of an alias that may be used interchangeably with the primary attribute label.

Alias: An alternative name used fc an attribute.

Homonyms An attribute that may have more than one meaning is called a **homonym**. 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).

Homonym: An attribute that m have more than one meaning.

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_No,Address)

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

STUDENT(Student_ID,Name,Phone_No,Campus_Address,Permanent_Address).

Transitive Dependencies When two 3NF relations are merged to form a single relation, transitive dependencies (described earlier in this chapter) may result. For example, consider the following two relations:

STUDENT1(Student_ID,Major)

STUDENT2(Student_ID,Advisor)

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

STUDENT(Student_ID,Major,Advisor)

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

Major → Advisor

If the preceding functional dependency exists, then STUDENT is in 2NF but not in 3NF, since it contains a transitive dependency. The analyst can create 3NF relations by removing the transitive dependency (Major becomes a foreign key in STUDENT):

STUDENT(Student_ID,Major)

MAJOR ADVISOR(Major,Advisor)

Supertype/Subtype Relationships These relationships may be hidden in user views or relations. Suppose that we have the following two hospital relations:

PATIENT1(Patient_ID,Name,Address)

PATIENT2(Patient_ID,Room_No)

Initially, it appears that these two relations can be merged into a single PATIENT relation. However, the analyst correctly suspects that there are two different types of patients: resident patients and outpatients. PATIENT1 actually contains attributes common to *all* patients. PATIENT2 contains an attribute (Room_No) that is a characteristic only of resident patients. In this situation, the analyst should create supertype/subtype relationships for these entities:

PATIENT(Patient_ID,Name,Address)

RESIDENT PATIENT(Patient_ID,Room_No)

OUTPATIENT(Patient_ID,Date_Treated)

For an extended discussion of view integration in database design, see Navathe, Elmasri, and Larson (1986).

A FINAL STEP FOR DEFINING RELATIONAL KEYS

In Chapter 3 we provided some criteria for selecting *identifiers*: does not change alue over time, must be unique and known, nonintelligent, and use a single ttribute surrogate for composite identifier. Actually, none of these criteria must pply until the database is implemented (that is, when the identifier becomes a pri- nary key and is defined as a field in the physical database). Before the relations are lefined as tables, the primary keys of relations should, if necessary, be changed to conform to these criteria.

Recently database experts (Johnston, 2000) have strengthened the criteria for primary key specification. Experts now also recommend that a primary key be unique across *the whole database* (a so-called **enterprise key**), not just unique within the relational table to which it applies. This criterion makes a primary key more like what in object-oriented databases is called an *object identifier* (see Chapters 14 and 15). With this recommendation, the primary key of a relation becomes a value internal to the database system and has no business meaning.

Enterprise key: A primary key whose value is unique across all relations.

A candidate primary key, such as Emp_ID in the EMPLOYEE1 relation of Figure 5-1 or Salesperson in the SPERSON relation of Figure 5-25, if ever used in the orga- nization, is called a "business key" and would be included in the relation as a nonkey attribute. The EMPLOYEE1 and SPERSON relations (and every other relation in the database) then have a new enterprise key attribute (called, say, Object_ID), which has no business meaning.

Why create this extra attribute? One of the main motivations for an enterprise l database evolvability—merging new relations into a database once the database is created. For example, consider the following two relations:

EMPLOYEE(Emp_ID,Emp_Name,Dept_Name,Salary)

CUSTOMER(Cust_ID,Cust_Name,Address)

In this example without an enterprise key, Emp_ID and Cust_ID may or may not have the same format, length, and datatype, whether they are intelligent or nonintelli- gent. Suppose the organization evolves its information processing needs and recog- nizes that employees can also be customers, so employee and customer are simply two subtypes of the same PERSON supertype. Thus, the organization would then like to have three relations:

PERSON(Person_ID,Person_Name)

EMPLOYEE(Person_ID,Dept_Name,Salary)

CUSTOMER(Person_ID,Address)

In this case Person_ID is supposed to be the same value for the same person throughout. But if values for Emp_ID and Cust_ID were selected before relation PER- SON was created, the values for Emp_ID and Cust_ID probably will not match. More- over, if we change the values of Emp_ID and Cust_ID to match the new Person_ID, then how do we ensure all Emp_IDs and Cust_IDs are unique should another employee or customer already have the associated Person_ID value? Even worse, if there are other tables that relate to, say, EMPLOYEE, then foreign keys in these other tables have to change, creating a ripple effect of foreign key changes. The only way to guarantee that each primary key of a relation is unique across the database is to cre- ? n enterprise key from the very beginning so primary keys never have to change.

n our example, the original database (without PERSON) with an enterprise key is shown in Figures 5-27a (the relations) and 5-27b (sample data). In this figure, and OBJECT is the supertype of all other

or any other internal system attributes for an object instance. Then when PERSON is needed, the database evolves to the design shown in Figures 5-27c (the relations) and 5-27d (sample data). Evolution to the database with PERSON still requires some alterations to existing tables, but not to primary key values—the name attribute is moved to PERSON since it is common to both subtypes and a foreign key is added to EMPLOYEE and CUSTOMER to point to the common person instance. As you will see in Chapter 7, it is easy to add and delete nonkey columns, even foreign keys, to table definitions. In contrast, changing the primary key of a relation is not allowed by most database management systems because of the extensive cost of the foreign key ripple effect.

Figure 5-27
Enterprise key

(a) Relations with enterprise key

OBJECT (OID, Object_Type)
EMPLOYEE (OID, Emp_ID, Emp_Name, Dept_Name, Salary)
CUSTOMER (OID, Cust_ID, Cust_Name, Address)

(b) Sample data with enterprise key

OBJECT

OID	Object_Type
1	EMPLOYEE
2	CUSTOMER
3	CUSTOMER
4	EMPLOYEE
5	EMPLOYEE
6	CUSTOMER
7	CUSTOMER

EMPLOYEE

OID	Emp_ID	Emp_Name	Dept_Name	Salary
1	100	Jennings, Fred	Marketing	50000
4	101	Hopkins, Dan	Purchasing	45000
5	102	Huber, Ike	Accounting	45000

CUSTOMER

OID	Cust_ID	Cust_Name	Address
2	100	Fred's Warehouse	Greensboro, NC
3	101	Bargain Bonanza	Moscow, ID
6	102	Jasper's	Tallahassee, FL
7	103	Desks 'R Us	Kettering, OH

(c) Relations after adding PERSON relation

OBJECT (OID, Object_Type)
EMPLOYEE (OID, Emp_ID, Dept_Name, Salary, Person_ID)
CUSTOMER (OID, Cust_ID, Address, Person_ID)
PERSON (OID, Name)

OBJECT

OID	Object_Type
1	EMPLOYEE
2	CUSTOMER
3	CUSTOMER
4	EMPLOYEE
5	EMPLOYEE
6	CUSTOMER
7	CUSTOMER
8	PERSON
9	PERSON
10	PERSON
11	PERSON
12	PERSON
13	PERSON
14	PERSON

PERSON

OID	Name
8	Jennings, Fred
9	Fred's Warehouse
10	Bargain Bonanza
11	Hopkins, Dan
12	Huber, Ike
13	Jasper's
14	Desks 'R Us

EMPLOYEE

OID	Emp_ID	Dept_Name	Salary	Person_ID
1	100	Marketing	50000	8
4	101	Purchasing	45000	11
5	102	Accounting	45000	12

CUSTOMER

OID	Cust_ID	Address	Person_ID
2	100	Greensboro, NC	9
3	101	Moscow, ID	10
6	102	Tallahassee, FL	13
7	103	Kettering, OH	14

Figure 5-27
(continued)
(d) Sample data after adding PERSON relation

Summary

Logical database design is the process of transforming the conceptual data model into a logical data model. The emphasis in this chapter has been on the relational data model, because of its importance in contemporary database systems. The relational data model represents data in the form of tables called relations. A relation is a named, two-dimensional table of data. A key property of relations is that they cannot contain multivalued attributes.

In this chapter, we described the major steps in the logdatabase design process. This process is based on transforming E-R diagrams to normalized relations. The three steps in this process are the following: Transform E-R (and tions in third normal form that can be implemented using any contemporary relational database management system.

Each entity type in the E-R diagram is transformed to a relation that has the same primary key as the entity type. A one-to-many relationship is represented by adding a foreign key to the relation that represents the entity on the many-side of the relationship. (This foreign key is the primary key of the entity on the one-side of the relationship.) A many-to-many relationship is represented by creating a separate relation. The primary key of this relation is a composite key, consisting of the primary key of each of the entities that participate in the relationship.

The relational model does not directly support super-

tionships by creating a separate table (or relation) for the supertype and for each subtype. The primary key of each subtype is the same (or at least from the same domain) as for the supertype. The supertype must have an attribute called the subtype discriminator that indicates to which subtype (or subtypes) each instance of the supertype belongs. The purpose of normalization is to derive well-structured relations that are free of anomalies (inconsistencies or errors) that would otherwise result when the relations are updated or modified. Normalization is based on the analysis of functional dependencies, which are constraints between two attributes (or two sets of attributes). It may be accomplished in several stages. Relations in first normal form (1NF) contain no multivalued attributes or repeating groups. Relations in 2NF contain no partial dependencies, and relations in 3NF contain no transitive dependencies. We can use diagrams that show the functional dependencies in a relation to help decompose that relation (if necessary) to obtain relations in third normal form. Higher normal forms (beyond 3NF) have also been defined; we discuss these normal forms in Appendix B.

We must be careful when combining relations to deal with problems such as synonyms, homonyms, transitive dependencies, and supertype/subtype relationships. In addition, before relations are defined to the database management system, all primary keys should be described as single attribute nonintelligent keys, and preferably, as enterprise keys.

CHAPTER REVIEW

Key Terms

Alias
Anomaly
Candidate key
Composite key
Determinant
Enterprise key
Entity integrity rule
First normal form
Foreign key
Functional dependency
Homonym
Normal form
Normalization
Null
Partial functional dependency
Primary key
Recursive foreign key
Referential integrity constraint
Relation
Second normal form
Synonyms
Third normal form
Transitive dependency
Well-structured relation

Review Questions

1. Define each of the following terms:
   a. determinant
   b. functional dependency
   c. transitive dependency
   d. recursive foreign key
   e. normalization
   f. composite key
   g. relation
   h. normal form
   i. partial functional dependency
   j. enterprise key

2. Match the following terms to the appropriate definitions:

____ well-structured relation	a. constraint between two attributes
____ anomaly	b. functional dependency between nonkey attributes
____ functional dependency	c. references primary key in same relation
____ determinant	d. multivalued attributes removed
____ composite key	e. inconsistency or error
____ 1NF	f. contains little redundancy
____ 2NF	g. contains two (or more) attributes
____ 3NF	h. contains no partial functional
____ recursive foreign key	i. transitive dependencies eliminated
____ relation	j. attribute on left-hand side of functional dependency
____ transitive dependency	k. named two-dimensional table of data

3. Contrast the following terms:
   a. normal form; normalization
   b. candidate key; primary key
   c. functional dependency; transitive dependency
   d. composite key; recursive foreign key
   e. determinant; candidate key
   f. foreign key; primary key

4. Summarize six important properties of relations.

5. Describe two properties that must be satisfied by candidate keys.

6. Describe three types of anomalies that can arise in a table.

7. Fill in the blanks in each of the following statements:
   a. A relation that has no partial functional dependencies is in _____ normal form.
   b. A relation that has no multivalued attributes is in _____ normal form.

le 5-2 Sample Data for Parts and Vendors

Part_No.	Description	Vendor_Name	Address	Unit_Cost
1234	Logic chip	Fast Chips	Cupertino	10.00
		Smart Chips	Phoenix	8.00
5678	Memory chip	Fast Chips	Cupertino	3.00
		Quality Chips	Austin	2.00
		Smart Chips	Phoenix	5.00

b. List the functional dependencies in PART SUPPLIER and identify a candidate key.

c. For the relation PART SUPPLIER, identify each of the following: an insert anomaly, a delete anomaly, and a modification anomaly.

d. Draw a relational schema for PART SUPPLIER (similar to Figure 5-23) and show the functional dependencies.

e. In what normal form is this relation?

f. Develop a set of 3NF relations from PART SUPPLIER.

7. Table 5-3 shows a relation called GRADE REPORT for a university. Following is a description of the functional dependencies in GRADE REPORT:

- Student_ID → Student_Name,Campus_Address,Major
- Course_ID → Course_Title,Instructor_Name, Instructor_Location
- Student_ID,Course_ID → Grade
- Instructor_Name → Instructor_Location

Following is your assignment:

a. Draw a relational schema (similar to Figure 5-23) and diagram the functional dependencies in the relation.

b. In what normal form is this relation?

c. Decompose GRADE REPORT into a set of 3NF relations.

d. Draw a relational schema for your 3NF relations (similar to Figure 5-5) and show the referential integrity constraints.

8. Transform Figure 3-15b, attribute version, to 3NF relations. Transform Figure 3-15b, relationship version, to 3NF relations. Compare these two sets of 3NF relations to those in Figure 5-10. What observations and conclusions do you reach by comparing these different sets of 3NF relations?

Field Exercises

1. Interview systems and database designers at several organizations. Ask them to describe the process they use for logical design. How do they transform their conceptual data models (e.g., E-R diagrams) to relational schema? What is the role of CASE tools in this process? Do they use normalization? If so, to what level?

2. Obtain a common document such as a sales slip, customer invoice from an auto repair shop, credit card statement, etc. Then do the following:

a. List the attribute names on the document.

b. List the functional dependencies among the attributes (make any assumptions you find necessary).

c. Draw a schema for the relation (similar to Figure 5-23) and diagram the functional dependencies.

d. Decompose the relation into a set of 3NF relations, and draw the relational schema (similar to Figure 5-5).

3. Obtain documentation for three popular PC-based CASE tools and compare the notation they use for representing relational schemas. You may use the following Internet

Table 5-3 GRADE REPORT Relation

GRADE REPORT								
Student_ID	Student_ Name	Campus_ Address	Major	Course_ID	Course_ Title	Instructor_ Name	Instructor_ Location	Grade
168300458	Williams	208 Brooks	IS	IS 350	Database Mgt	Codd	B 104	A
268300458	Williams	208 Brooks	IS	IS 465	Systems Analysis	Parsons	B 317	B
543291073	Baker	104 Phillips	Acctg	IS 350	Database Mgt	Codd	B 104	C
3291073	Baker	104 Phillips	Acctg	Acct 201	Fund Acctg	Miller	H 310	B
543291073	Baker	104 Phillips	Acctg	Mktg 300	Intro Mktg	Bennett	B 212	A

addresses to obtain on-line information for the following three tools:

a. Microsoft Access: www.microsoft.com/Visual Tools/
b. Erwin: www.logicworks.com
c. EasyER: www.esti.com

4. Find a form or report from a business organization, possibly a statement, bill, or document you have received. Draw an EER diagram of the data in this form or report. Transform the diagram into a set of 3NF relations.

Appendix B

Advanced Normal Forms

In Chapter 5 we introduced the topic of normalization and described first through third normal forms in detail. Relations in third normal form (3NF) are sufficient for most practical database applications. However, 3NF does not guarantee that all anomalies have been removed. As indicated in Chapter 5, several additional normal forms are designed to remove these anomalies: Boyce-Codd normal form, fourth normal form, and fifth normal form (see Figure 5-22). We describe Boyce-Codd normal form and fourth normal form in this appendix.

BOYCE-CODD NORMAL FORM

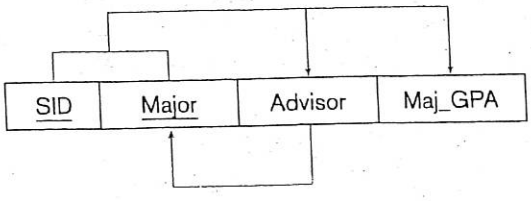
When a relation has more than one candidate key, anomalies may result even though that relation is in 3NF. For example, consider the STUDENT_ADVISOR relation shown in Figure B-1. This relation has the following attributes: SID (student ID), Major, Advisor, and Maj_GPA. Sample data for this relation are shown in Figure B-1a, and the functional dependencies are shown in Figure B-1b.

Figure B-1
Relation in 3NF, but not BCNF
(a) Relation with sample data

STUDENT_ADVISOR

SID	Major	Advisor	Maj_GPA
123	Physics	Hawking	4.0
123	Music	Mahler	3.3
456	Literature	Michener	3.2
789	Music	Bach	3.7
678	Physics	Hawking	3.5

(b) Functional dependencies in STUDENT_ADVISOR



As shown in Figure B-1b, the primary key for this relation is the composite key consisting of SID and Major. Thus the two attributes Advisor and Maj_GPA are functionally dependent on this key. This reflects the constraint that although a given student may have more than one major, for each major a student has exactly one advisor and one GPA.

There is a second functional dependency in this relation: Major is functionally dependent on Advisor. That is, each advisor advises in exactly one major. Notice that this is *not* a transitive dependency. In Chapter 5 we defined a transitive dependency as a functional dependency between two nonkey attributes. In contrast, in this example a key attribute (Major) is functionally dependent on a nonkey attribute (Advisor).

Anomalies in STUDENT_ADVISOR

The STUDENT_ADVISOR relation is clearly in 3NF, since there are no partial functional dependencies and no transitive dependencies. Nevertheless, because of the functional dependency between Major and Advisor there are anomalies in this relation. Consider the following examples:

1. Suppose that in Physics the advisor Hawking is replaced by Einstein. This change must be made in two (or more) rows in the table (update anomaly).
2. Suppose we want to insert a row with the information that Babbage advises in Computer Science. This, of course, cannot be done until at least one student majoring in Computer Science is assigned Babbage as an advisor (insertion anomaly).
3. Finally, if student number 789 withdraws from school, we lose the information that Bach advises in Music (deletion anomaly).

Definition of Boyce-Codd Normal Form (BCNF)

The anomalies in STUDENT_ADVISOR result from the fact that there is a determinant (Advisor) that is not a candidate key in the relation. R. F. Boyce and E. F. Codd identified this deficiency and proposed a stronger definition of 3NF that remedies the problem. We say a relation is in **Boyce-Codd normal form (BCNF)** if and only if every determinant in the relation is a candidate key. STUDENT_ADVISOR is not in BCNF because although the attribute Advisor is a determinant, it is not a candidate key (only Major is functionally dependent on Advisor).

Boyce-Codd normal form (BCNF): A relation in which every determinant is a candidate key.

Converting a Relation to BCNF

A relation that is in 3NF (but not BCNF) can be converted to relations in BCNF using a simple two-step process. This process is shown in Figure B-2.

In this first step, the relation is modified so that the determinant in the relation that is not a candidate key becomes a component of the primary key of the revised relation. The attribute that is functionally dependent on that determinant becomes a nonkey attribute. This is a legitimate restructuring of the original relation because of the functional dependency.

The result of applying this rule to STUDENT_ADVISOR is shown in Figure B-2a. The determinant Advisor becomes part of the composite primary key. The attribute Major, which is functionally dependent on Advisor, becomes a nonkey attribute.

If you examine Figure B-2a, you will discover that the new relation has a partial functional dependency (Major is functionally dependent on Advisor, which is just one component of the primary key). Thus the new relation is in first (but not second) normal form.

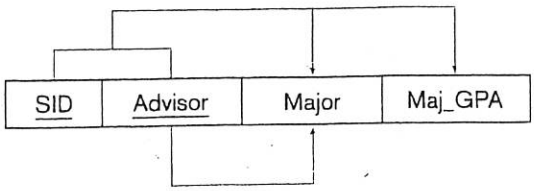
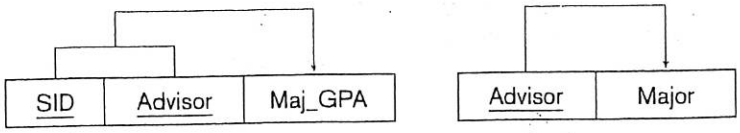


Figure B-2
Converting a relation to BCNF relations
(a) Revised STUDENT_ADVISOR relation (2NF)

1NF

(b) Two relations in BCNF



(c) Relations with sample data

STUDENT

SID	Advisor	Maj_GPA
123	Hawking	4.0
123	Mahler	3.3
456	Michener	3.2
789	Bach	3.7
678	Hawking	3.5

ADVISOR

Advisor	Major
Hawking	Physics
Mahler	Music
Michener	Literature
Bach	Music

The second step in the conversion process is to decompose the relation to eliminate the partial functional dependency, as we learned in Chapter 5. This results in two relations, as shown in Figure B-2b. These relations are in 3NF. In fact the relations are also in BCNF, since there is only one candidate key (the primary key) in each relation. Thus we see that if a relation has only one candidate key (which therefore becomes the primary key), then 3NF and BCNF are equivalent.

The two relations (now named STUDENT and ADVISOR) with sample data are shown in Figure B-2c. You should verify that these relations are free of the anomalies that were described for STUDENT_ADVISOR. You should also verify that you can recreate the STUDENT_ADVISOR relation by joining the two relations STUDENT and ADVISOR.

Another common situation in which BCNF is violated is when there are two (or more) overlapping candidate keys of the relation. Consider the relation in Figure B-3a. In this example, there are two candidate keys: (SID,COURSE_ID) and (SNAME,COURSE_ID), in which COURSE_ID appears in both candidate keys. The problem with this relationship is that we cannot record student data (SID and SNAME) unless the student has taken a course. Figure B-3b shows two possible solutions, each of which creates two relations that are in BCNF.

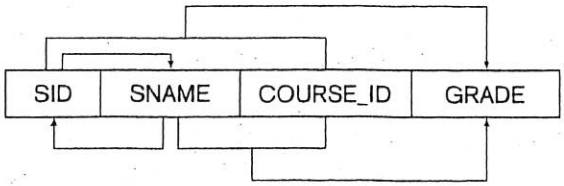
Candidate keys are overlapped.

#2

FOURTH NORMAL FORM

When a relation is in BCNF, there are no longer any anomalies that result from ıctional dependencies. However, there may still be anomalies that result from multivalued dependencies (defined below). For example, consider the table shown in

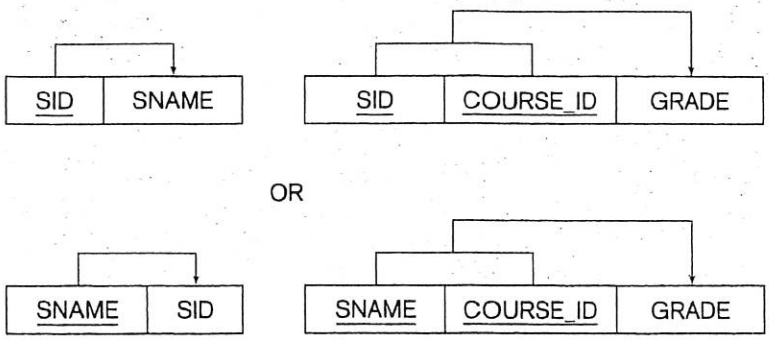
Figure B-3
Converting a relation with overlapping candidate keys to BCNF
(a) Relation with overlapping candidate keys



(b) Two alternative pairs of relations in BCNF

I prefer this →

Because SNAME May need to change.



SID	SName	CourseID	Grade

Key1 : SName CourseID

SID → SName

Nonkey determines Key

Key2 : SID CourseID

SName → SID

Nonkey determines Key

Multivalued dependency: The type of dependency that exists when there are at least three attributes (e.g., A, B, and C) in a relation, with a well-defined set of B and C values for each A value, but those B and C values are independent of each other.

Multivalued Dependencies

The type of dependency shown in this example is called a **multivalued dependency**, which exists when there are at least three attributes (for example, A, B, and C) in a relation, and for each value of A there is a well-defined set of values of B and a well-defined set of values of C. However, the set of values of B is independent of set C, and vice versa.