3 The Relational Model and Normalization

Chapter Objectives

- To understand basic relational terminology
- To understand the characteristics of relations
- To understand alternative terminology used in describing the relational model
- To be able to identify functional dependencies, determinants, and dependent attributes
- To identify primary, candidate, and composite keys
- To be able to identify possible insertion, deletion, and update anomalies in a relation
- To be able to place a relation into BCNF normal form
- To understand the special importance of domain/key normal form
- To be able to identify multivalued dependencies
- To be able to place a relation in fourth normal form

As we discussed in Chapter 1, databases arise from three sources: from existing data, from the development of new information systems, and from the redesign of existing databases. In this chapter and the next, we consider the design of databases from existing data, such as data from spreadsheets or extracts of existing databases.

The premise of Chapters 3 and 4 is that you have received one or more tables of data from some source that are to be stored in a new database. The question is: Should this data be stored as is, or should it be transformed in some way before it is stored? For example, consider the two tables in the top part of Figure 3-1. These are the SKU_DATA and ORDER_ITEM tables extracted from the Cape Codd Outdoor Sports database as used in the database in Chapter 2.

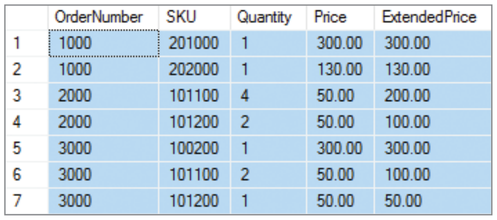
You can design the new database to store this data as two separate tables, or you can join the tables together and design the database with just one table. Each alternative has advantages and disadvantages. When you make the decision to use one design, you obtain certain advantages at the expense of certain costs. The purpose of this chapter is to help you understand those advantages and costs.

Such questions do not seem difficult, and you may be wondering why we need two chapters to answer them. In truth, even a single table can have surprising complexity. Consider, for example, the table in Figure 3-2, which shows sample data

FIGURE 3-1

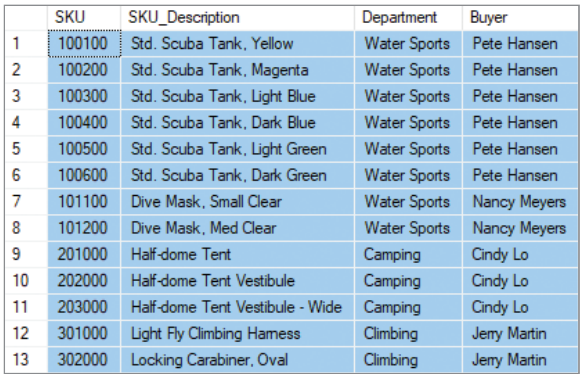
How Many Tables?

ORDER_ITEM



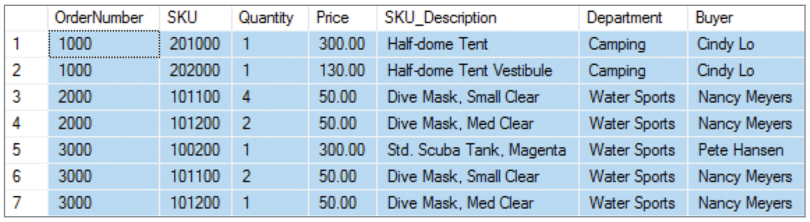
	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	100200	1	300.00	300.00
6	3000	101100	2	50.00	100.00
7	3000	101200	1	50.00	50.00

SKU_DATA



	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	100300	Std. Scuba Tank, Light Blue	Water Sports	Pete Hansen
4	100400	Std. Scuba Tank, Dark Blue	Water Sports	Pete Hansen
5	100500	Std. Scuba Tank, Light Green	Water Sports	Pete Hansen
6	100600	Std. Scuba Tank, Dark Green	Water Sports	Pete Hansen
7	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
8	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
9	201000	Half-dome Tent	Camping	Cindy Lo
10	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
11	203000	Half-dome Tent Vestibule - Wide	Camping	Cindy Lo
12	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
13	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

SKU_ITEM

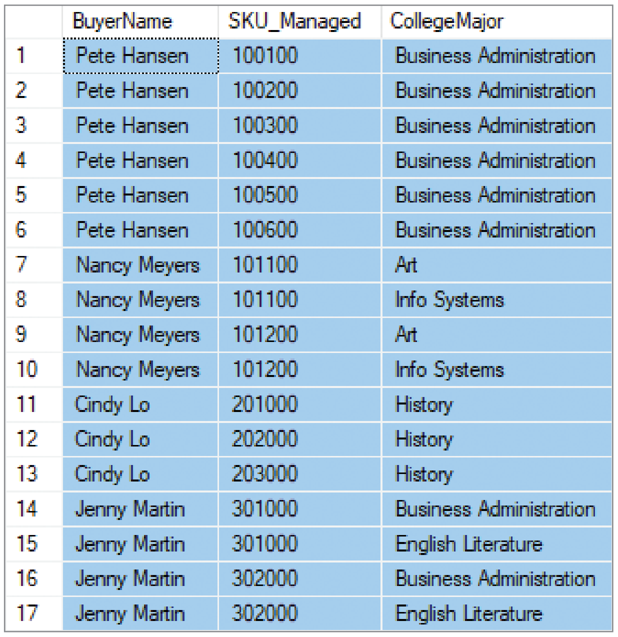


	OrderNumber	SKU	Quantity	Price	SKU_Description	Department	Buyer
1	1000	201000	1	300.00	Half-dome Tent	Camping	Cindy Lo
2	1000	202000	1	130.00	Half-dome Tent Vestibule	Camping	Cindy Lo
3	2000	101100	4	50.00	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	2000	101200	2	50.00	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	3000	100200	1	300.00	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
6	3000	101100	2	50.00	Dive Mask, Small Clear	Water Sports	Nancy Meyers
7	3000	101200	1	50.00	Dive Mask, Med Clear	Water Sports	Nancy Meyers

FIGURE 3-2

PRODUCT_BUYER—
A Very Strange Table

PRODUCT_BUYER



	BuyerName	SKU_Managed	CollegeMajor
1	Pete Hansen	100100	Business Administration
2	Pete Hansen	100200	Business Administration
3	Pete Hansen	100300	Business Administration
4	Pete Hansen	100400	Business Administration
5	Pete Hansen	100500	Business Administration
6	Pete Hansen	100600	Business Administration
7	Nancy Meyers	101100	Art
8	Nancy Meyers	101100	Info Systems
9	Nancy Meyers	101200	Art
10	Nancy Meyers	101200	Info Systems
11	Cindy Lo	201000	History
12	Cindy Lo	202000	History
13	Cindy Lo	203000	History
14	Jenny Martin	301000	Business Administration
15	Jenny Martin	301000	English Literature
16	Jenny Martin	302000	Business Administration
17	Jenny Martin	302000	English Literature

extracted from a corporate database. This simple table has three columns: the buyer's name, the **SKU (stock keeping unit)** of the products that the buyer purchases, and the names of the buyer's college major(s). Buyers manage more than one SKU, and they can have multiple college majors.

To understand why this is an odd table, suppose that Nancy Meyers is assigned a new SKU, say 101300. What addition should we make to this table? Clearly, we need to add a row for the new SKU, but if we add just one row, say the row ('Nancy Meyers', 101300, 'Art'), it will appear that she manages product 101300 as an Art major, but not as an Info Systems major. To avoid such an illogical state, we need to add two rows: ('Nancy Meyers', 101300, 'Art') and ('Nancy Meyers', 101300, 'Info Systems').

This is a strange requirement. Why should we have to add two rows of data simply to record the fact that a new SKU has been assigned to a buyer? Further, if we assign the product to Pete Hansen instead, we would only have to add one row, but if we assigned the product to a buyer who had four majors, we would have to add four new rows.

The more one thinks about the table in Figure 3-2, the stranger it becomes. What changes should we make if SKU 101100 is assigned to Pete Hansen? What changes should we make if SKU 100100 is assigned to Nancy Meyers? What should we do if all the SKU values in Figure 3-2 are deleted? Later in this chapter, you will learn that these problems arise because this table has a problem called a *multivalued dependency.* Even better, you will learn how to remove that problem.

Tables can have many different patterns; some patterns are susceptible to serious problems and other patterns are not. Before we can address this question, however, you need to learn some basic terms.

Relational Model Terminology

Figure 3-3 lists the most important terms used by the relational model. By the time you finish Chapters 3 and 4, you should be able to define each of these terms and explain how each pertains to the design of relational databases. Use this list of terms as a check on your comprehension.

Relations

So far, we have used the terms *table* and *relation* interchangeably. In fact, a relation is a special case of a table. This means that all relations are tables, but not all tables are relations. Codd defined the characteristics of a relation in his 1970 paper that laid the foundation for the relational model.[1] Those characteristics are summarized in Figure 3-4.

BY THE WAY In Figure 3-4 and in this discussion, we use the term **entity** to mean some identifiable thing. A customer, a salesperson, an order, a part, and a lease are all examples of what we mean by an entity. When we introduce the entity-relationship model in Chapter 5, we will make the definition of entity more precise. For now, just think of an entity as some identifiable thing that users want to track.

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, June 1970, pp. 377–387. A downloadable copy of this paper in PDF format is available at *http://dl.acm.org/citation.cfm?id=362685*.

FIGURE 3-3
Important Relational Model Terms

Important Relational Terms
Relation
Functional dependency
Determinant
Candidate key
Composite key
Primary key
Surrogate key
Foreign key
Referential integrity constraint
Normal form
Multivalued dependency

Characteristics of Relations

A **relation** has a specific definition, as shown in Figure 3-4, and for a table to be a relation, the criteria of this definition must be met. First, the rows of the table must store data about an entity, and the columns of the table must store data about the characteristics of those entities. Next, the names of the columns are unique; no two columns in the same relation may have the same name.

Further, in a relation, all of the values in a column are of the same kind. If, for example, the second column of the first row of a relation has FirstName, then the second column of every row in the relation has FirstName. This is an important requirement that is known as the **domain integrity constraint**, where the term **domain** means a grouping of data that meets a specific type definition. For example, FirstName would have a domain of names such as *Albert*, *Bruce*, *Cathy*, *David*, *Edith*, and so forth, and all values of FirstName *must* come from the names in that domain. The EMPLOYEE table shown in Figure 3-5 meets these criteria and is a relation.

FIGURE 3-4
Characteristics of Relations

Characteristics of Relations
Rows contain data about an entity.
Columns contain data about attributes of the entities.
All entries in a column are of the same kind.
Each column has a unique name.
Cells of the table hold a single value.
The order of the columns is unimportant.
The order of the rows is unimportant.
No two rows may be identical.

EmployeeNumber	FirstName	LastName	Department	EmailAddress	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	518-834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	518-834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	518-834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	518-834-1102
500	Tom	Jackson	Production	TJ@somewhere.com	518-834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	518-834-3101
700	Richard	Bandalone	Legal	RB@somewhere.com	518-834-3102

FIGURE 3-5

Sample EMPLOYEE Relation

BY THE WAY Columns in different relations may have the same name. In Chapter 2, for example, two relations had a column named SKU. When there is risk of confusion, we precede the column name with the relation name followed by a period. Thus, the name of the SKU column in the SKU_DATA relation is SKU_DATA.SKU, and column C1 of relation R1 is named R1.C1. Because relation names are unique within a database and because column names are unique within a relation, the combination of relation name and column name uniquely identifies every column in the database.

Each cell of a relation has only a single value or item; multiple entries are not allowed. The table in Figure 3-6 is *not* a relation because the Phone values of employees Caruthers and Bandalone store multiple phone numbers.

In a relation, the order of the rows and the order of the columns are immaterial. No information can be carried by the ordering of rows or columns. The table in Figure 3-7 is not a relation because the entries for employees Caruthers and Caldera require a particular row arrangement. If the rows in this table were rearranged, we would not know which employee has the indicated Fax and Home numbers.

Finally, according to the last characteristic in Figure 3-4, for a table to be a relation, no two rows can be identical. As you learned in Chapter 2, some SQL statements do produce tables with duplicate rows. In such cases, you can use the DISTINCT keyword to force uniqueness. Such row duplication occurs only as a result of SQL manipulation. Tables that you design to be stored in the database should never contain duplicate rows.

FIGURE 3-6

Nonrelational Table—Multiple Entries per Cell

EmployeeNumber	FirstName	LastName	Department	EmailAddress	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	518-834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	518-834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	518-834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	518-834-1102, 518-834-1191, 518-834-1192
500	Tom	Jackson	Production	TJ@somewhere.com	518-834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	518-834-3101
700	Richard	Bandalone	Legal	RB@somewhere.com	518-834-3102, 518-834-3191

EmployeeNumber	FirstName	LastName	Department	EmailAddress	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	518-834-1101
200	Mary	Abernathy	Finance	MA@somewhere.com	518-834-2101
300	Liz	Smathers	Finance	LS@somewhere.com	518-834-2102
400	Tom	Caruthers	Accounting	TC@somewhere.com	518-834-1102
				Fax:	518-834-9911
				Home:	518-723-8795
500	Tom	Jackson	Production	TJ@somewhere.com	518-834-4101
600	Eleanore	Caldera	Legal	EC@somewhere.com	518-834-3101
				Fax:	518-834-9912
				Home:	518-723-7654
700	Richard	Bandalone	Legal	RB@somewhere.com	518-834-3102

FIGURE 3-7

Nonrelational Table—Order of Rows Matters and Kind of Column Entries Differs in Email

BY THE WAY Do not fall into a common trap. Even though every cell of a relation must have a single value, this does not mean that all values must have the same length. The table in Figure 3-8 is a relation even though the length of the Comment column varies from row to row. It is a relation because, even though the comments have different lengths, there is only *one* comment per cell.

Alternative Terminology

As defined by Codd, the columns of a relation are called **attributes** and the rows of a relation are called **tuples** (rhymes with "couples"). Most practitioners, however, do not use these academic-sounding terms and instead use the terms *column* and *row*. Also, even though a

FIGURE 3-8

Relation with Variable-Length Column Values

EmployeeNumber	FirstName	LastName	Department	EmailAddress	Phone	Comments
100	Jerry	Johnson	Accounting	JJ@somewhere.com	518-834-1101	Joined the Accounting Department in March after completing his MBA. Will take the CPA exam this fall.
200	Mary	Abernathy	Finance	MA@somewhere.com	518-834-2101	
300	Liz	Smathers	Finance	LS@somewhere.com	518-834-2102	
400	Tom	Caruthers	Accounting	TC@somewhere.com	518-834-1102	
500	Tom	Jackson	Production	TJ@somewhere.com	518-834-4101	
600	Eleanore	Caldera	Legal	EC@somewhere.com	518-834-3101	
700	Richard	Bandalone	Legal	RB@somewhere.com	518-834-3102	Is a full-time consultant to Legal on a retainer basis.

FIGURE 3-9

Three Sets of Equivalent Terms

Table	Column	Row
Relation	Attribute	Tuple
File	Field	Record

table is not necessarily a relation, most practitioners mean *relation* when they say *table*. Thus, in most conversations the terms *relation* and *table* are synonymous. In fact, for the rest of this book *table* and *relation* will be used synonymously.

Additionally, a third set of terminology can be used. Some practitioners use the terms *file, field*, and *record* for the terms *table, column*, and *row*, respectively. These terms arose from traditional data processing and are common in connection with legacy systems. Sometimes people mix and match these terms. You might hear someone say, for example, that a relation has a certain column and contains 47 records. These three sets of terms are summarized in Figure 3-9.

To Key, or Not to Key—That Is the Question!

Again as defined by Codd, the rows of a relation must be unique (no two rows may be identical), but there is no requirement for a designated *primary key* in the relation. You will recall that in Chapter 1, we described a *primary key* as a column (or columns) with a set of values that uniquely identify each row.

However, the requirement that no two rows be identical *implies* that a primary key *can* be defined for the relation. Further, in the "real world" of databases, every relation (or table as they are more often referred to in daily use) *does* have a defined primary key.

To understand how to designate or assign a primary key for a relation, we need to learn about the different types of keys used in relational databases, and this means we need to learn about functional dependencies, which are the foundation upon which keys are built. We will then discuss specifically how to assign primary keys in relations.

Functional Dependencies

Functional dependencies are the heart of the database design process, and it is vital for you to understand them. We will first explain the concept in general terms and then examine two examples. We will then be able to define exactly what a *functional dependency* is.

We begin with a short excursion into the world of algebra. Suppose you are buying boxes of cookies and someone tells you that each box costs $5.00. With this fact, you can compute the cost of several boxes with the following formula:

CookieCost = NumberOfBoxes × $5

A more general way to express the relationship between CookieCost and NumberOfBoxes is to say that CookieCost *depends on* NumberOfBoxes. Such a statement tells us the character of the relationship between CookieCost and NumberOfBoxes, even though it doesn't give us the formula. More formally, we can say that CookieCost is **functionally dependent** on NumberOfBoxes. Such a statement can be written as:

NumberOfBoxes → CookieCost

This expression can be read as "NumberOfBoxes *determines* CookieCost." The variable on the left, here NumberOfBoxes, is called the **determinant**.

Using another formula, we can compute the extended price of a part order by multiplying the quantity of the item by its unit price, or:

$$\text{ExtendedPrice} = \text{Quantity} \times \text{UnitPrice}$$

In this case, we say that ExtendedPrice is functionally dependent on Quantity and UnitPrice, or:

(Quantity, UnitPrice) → ExtendedPrice

Here the determinant is the composite (Quantity, UnitPrice).

Functional Dependencies That Are Not Equations

In general, a **functional dependency** exists when the value of one or more attributes determines the value of another attribute. Many functional dependencies exist that do not involve equations.

Consider an example. Suppose you know that a sack contains red, blue, or yellow objects. Further, suppose you know that the red objects weigh 5 pounds, the blue objects weigh 5 pounds, and the yellow objects weigh 7 pounds. If a friend looks into the sack, sees an object, and tells you the color of the object, you can tell her the weight of the object. We can formalize this as:

ObjectColor → Weight

Thus, we can say that Weight is functionally dependent on ObjectColor and that ObjectColor determines Weight. The relationship here does not involve an equation, but the functional dependency holds. Given a value for ObjectColor, you can determine the object's weight.

If we also know that the red objects are balls, the blue objects are cubes, and the yellow objects are cubes, we can also say:

ObjectColor → Shape

Thus, ObjectColor determines Shape. We can put these two together to state:

ObjectColor → (Weight, Shape)

Thus, ObjectColor determines Weight and Shape.

Another way to represent these facts is to put them into a table:

ObjectColor	Weight	Shape
Red	5	Ball
Blue	5	Cube
Yellow	7	Cube

This table meets all of the conditions listed in Figure 3-4, and therefore it is a relation. You may be thinking that we performed a trick or sleight of hand to arrive at this relation, but in truth, the only reason for having relations is *to store instances of functional dependencies*. If there were a formula by which we could take ObjectColor and somehow compute Weight and Shape, then we would not need the table. We would just make the computation. Similarly, if there were a formula by which we could take EmployeeNumber and compute EmployeeName and HireDate, then we would not need an EMPLOYEE relation. However, because there is no such formula, we must store the combinations of EmployeeNumber, EmployeeName, and HireDate in the rows of a relation.

BY THE WAY Perhaps the easiest way to understand functional dependencies is:

If I tell you one specific fact, can you respond with a unique associated fact?

Using the earlier table, if I tell you that the ObjectColor is Red, can you uniquely tell me the associated Shape? *Yes*, you can, and it is Ball. Therefore, ObjectColor *determines* Shape, and a functional dependency exists with ObjectColor as the determinant.

Now, if I tell you that that the Shape is Cube, can you tell me the uniquely associated ObjectColor? *No*, you cannot because it could be either Blue or Yellow. Therefore, Shape *does not determine* ObjectColor, and ObjectColor is *not* functionally dependent on Shape.

Composite Functional Dependencies

The determinant of a functional dependency can consist of more than one attribute. For example, a grade in a class is determined by both the student and the class, or:

(StudentNumber, ClassNumber) → Grade

In this case, the determinant is called a **composite determinant**.

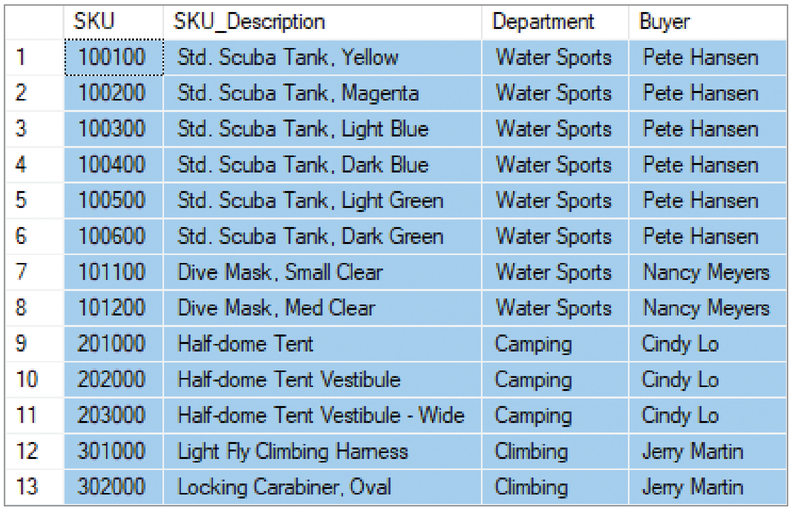
Notice that both the student and the class are needed to determine the grade. In general, if (A, B) → C, then neither A nor B will determine C by itself. However, if A → (B, C), then it is true that A → B and A → C (this is known as the **decomposition rule**). Work through examples of your own for both of these cases so that you understand why this is true. Also note that if A → B and A → C, then it is true that A → (B, C) (this is known as the **union rule**).

Finding Functional Dependencies

To fix the idea of functional dependency in your mind, consider what functional dependencies exist in the SKU_DATA and ORDER_ITEM tables in Figure 3-1.

Functional Dependencies in the SKU_DATA Table

To find functional dependencies in a table, we must ask "Does any column determine the value of another column?" For example, consider the values of the SKU_DATA table in Figure 3-1:



	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	100300	Std. Scuba Tank, Light Blue	Water Sports	Pete Hansen
4	100400	Std. Scuba Tank, Dark Blue	Water Sports	Pete Hansen
5	100500	Std. Scuba Tank, Light Green	Water Sports	Pete Hansen
6	100600	Std. Scuba Tank, Dark Green	Water Sports	Pete Hansen
7	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
8	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
9	201000	Half-dome Tent	Camping	Cindy Lo
10	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
11	203000	Half-dome Tent Vestibule - Wide	Camping	Cindy Lo
12	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
13	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

Consider the last two columns. If we know the value of Department, can we determine a unique value of Buyer? No, we cannot, because a Department may have more than one Buyer. In these sample data, 'Water Sports' is associated with Pete Hansen and Nancy Meyers. Therefore, Department does not functionally determine Buyer.

What about the reverse? Does Buyer determine Department? In every row, for a given value of Buyer, do we find the same value of Department? Every time Jerry Martin appears, for example, is he paired with the same department? The answer is yes. Further, every time Cindy Lo appears, she is paired with the same department. The same is true for the other buyers. Therefore, assuming that these data are representative, Buyer does determine Department, and we can write:

Buyer → Department

Does Buyer determine any other column? If we know the value of Buyer, do we know the value of SKU? No, we do not, because a given buyer has many SKUs assigned to him or her. Does Buyer determine SKU_Description? No, because a given value of Buyer occurs with many values of SKU_Description.

BY THE WAY As stated, for the Buyer → Department functional dependency, a Buyer is paired with one and only one value of Department. Notice that a buyer can appear more than once in the table, but, if so, that buyer is always paired with the same department. This is true for all functional dependencies. If A → B, then each value of A will be paired with one and only one value of B. A particular value of A may appear more than once in the relation, but, if so, it is always paired with the same value of B. Note, too, that the reverse is not necessarily true. If A → B, then a value of B may be paired with many values of A.

What about the other columns? It turns out that if we know the value of SKU, we also know the values of all of the other columns. In other words:

SKU → SKU_Description

because a given value of SKU will have just one value of SKU_Description. Next,

SKU → Department

because a given value of SKU will have just one value of Department. And, finally,

SKU → Buyer

because a given value of SKU will have just one value of Buyer.

We can combine these three statements as:

SKU → (SKU_Description, Department, Buyer)

For the same reasons, SKU_Description determines all of the other columns, and we can write:

SKU_Description → (SKU, Department, Buyer)

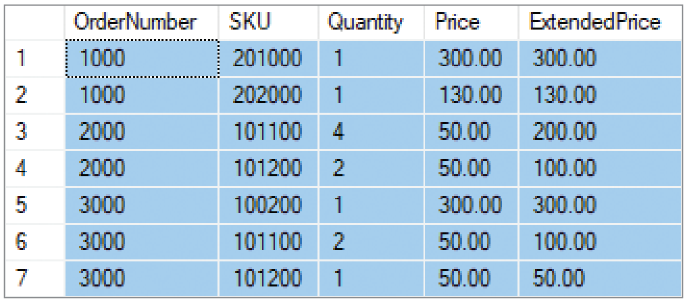
In summary, the functional dependencies in the SKU_DATA table are:

SKU → (SKU_Description, Department, Buyer)
SKU_Description → (SKU, Department, Buyer)
Buyer → Department

BY THE WAY You cannot always determine functional dependencies from sample data. You may not have any sample data, or you may have just a few rows that are not representative of all of the data conditions. In such cases, you must ask the users who are experts in the application that creates the data. For the SKU_DATA table, you would ask questions such as, “Is a Buyer always associated with the same Department?” and “Can a Department have more than one Buyer?” In most cases, answers to such questions are more reliable than sample data. When in doubt, trust the users.

Functional Dependencies in the ORDER_ITEM Table

Now consider the ORDER_ITEM table in Figure 3-1. For convenience, here is a copy of the data in that table:



	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	100200	1	300.00	300.00
6	3000	101100	2	50.00	100.00
7	3000	101200	1	50.00	50.00

What are the functional dependencies in this table? Start on the left. Does OrderNumber determine another column? It does not determine SKU because several SKUs are associated with a given order. For the same reasons, it does not determine Quantity, Price, or ExtendedPrice.

What about SKU? SKU does not determine OrderNumber because several OrderNumbers are associated with a given SKU. It does not determine Quantity or ExtendedPrice for the same reason.

What about SKU and Price? From this data, it does appear that

SKU → Price

but that might not be true in general. In fact, we know that prices can change after an order has been processed. Further, an order might have special pricing due to a sale or promotion. To keep an accurate record of what the customer actually paid, we need to associate a particular SKU price with a particular order. Thus:

(OrderNumber, SKU) → Price

Considering the other columns, Quantity, Price, and ExtendedPrice do not determine anything else. You can decide this by looking at the sample data. You can reinforce this conclusion by thinking about the nature of sales. Would a Quantity of 2 ever determine an OrderNumber or an SKU? This makes no sense. At the grocery store, if I tell you I bought two of something, you have no reason to conclude that my OrderNumber was 1010022203466 or that I bought carrots. Quantity does not determine OrderNumber or SKU.

Similarly, if I tell you that the price of an item was $3.99, there is no logical way to conclude what my OrderNumber was or that I bought a jar of green olives. Thus, Price does not determine OrderNumber or SKU. Similar comments pertain to ExtendedPrice. It turns out that no single column is a determinant in the ORDER_ITEM table.

What about pairs of columns? We already know that

(OrderNumber, SKU) → Price

Examining the data, (OrderNumber, SKU) determines the other two columns as well. Thus:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)

This functional dependency makes sense. It means that given a particular order and a particular item on that order, there is only one quantity, one price, and one extended price.

Notice, too, that because ExtendedPrice is computed from the formula ExtendedPrice = (Quantity * Price) we have:

(Quantity, Price) → ExtendedPrice

In summary, the functional dependencies in ORDER_ITEM are:

(OrderNumber, SKU) → (Quantity, Price, ExtendedPrice)

(Quantity, Price) → ExtendedPrice

No single skill is more important for designing databases than the ability to identify functional dependencies. Make sure you understand the material in this section. Work through Review Questions 3.58 and 3.59, the Regional Labs case questions, and The Queen Anne Curiosity Shop and Morgan Importing project questions at the end of the chapter. Ask your instructor for help if necessary. You *must* understand functional dependencies and be able to work with them.

When Are Determinant Values Unique?

In the previous section, you may have noticed an irregularity. Sometimes the determinants of a functional dependency are unique in a relation, and sometimes they are not. Consider the SKU_DATA relation, with determinants SKU, SKU_Description, and Buyer. In SKU_DATA, the values of both SKU and SKU_Description are unique in the table. For example, the SKU value 100100 appears just once. Similarly, the SKU_Description value 'Half-dome Tent' occurs just once. From this, it is tempting to conclude that values of determinants are always unique in a relation. However, this is *not* true.

For example, Buyer is a determinant, but it is not unique in SKU_DATA. The buyer 'Cindy Lo' appears in two different rows. In fact, for these sample data, all of the buyers occur in two different rows.

In truth, a determinant is unique in a relation only if it determines every other column in the relation. For the SKU_DATA relation, SKU determines all of the other columns. Similarly, SKU_Description determines all of the other columns. Hence, they both are unique. Buyer, however, only determines the Department column. It does not determine SKU or SKU_Description.

The determinants in ORDER_ITEM are (OrderNumber, SKU) and (Quantity, Price). Because (OrderNumber, SKU) determines all of the other columns, it will be unique in the relation. The composite (Quantity and Price) only determines ExtendedPrice. Therefore, it will not be unique in the relation.

This fact means that you cannot find the determinants of all functional dependencies simply by looking for unique values. Some of the determinants will be unique, but some will not. Instead, to determine if column A determines column B, look at the data and ask, "Every time a value of column A appears, is it matched with the same value of Column B?" If so, it can be a determinant of B. Again, however, sample data can be incomplete, so the best strategies are to think about the nature of the business activity from which the data arise and to ask the users.

Keys

The relational model has more keys than a locksmith. There are candidate keys, composite keys, primary keys, surrogate keys, and foreign keys. In this section, we will define each of these types of keys. Because key definitions rely on the concept of functional dependency, make sure you understand that concept before reading on.

In general, a **key** is a combination of one or more columns that is used to identify particular rows in a relation. Keys that have two or more columns are called **composite keys**.

Candidate Keys

A **candidate key** is a determinant that determines all of the other columns in a relation. The SKU_DATA relation has two candidate keys: SKU and SKU_Description. Buyer is a determinant, but it is not a candidate key because it determines only Department.

The ORDER_ITEM table has just one candidate key: (OrderNumber, SKU). The other determinant in this table, (Quantity, Price), is not a candidate key because it determines only ExtendedPrice.

Candidate keys identify a unique row in a relation. Given the value of a candidate key, we can find one and only one row in the relation that has that value. For example, given the SKU value of 100100, we can find one and only one row in SKU_DATA. Similarly, given the OrderNumber and SKU values (2000, 101100), we can find one and only one row in ORDER_ITEM.

Primary Keys

When designing a database, one of the candidate keys is selected to be the **primary key**. This term is used because this key will be defined to the database management system (DBMS), and the DBMS will use it as its primary means for finding rows in a table. A table has only one primary key. The primary key can have one column, or it can be a composite.

In this text, to clarify discussions we will sometimes indicate table structure by showing the name of a table followed by the names of the table's columns enclosed in parentheses. When we do this, we will underline the column(s) that comprise the primary key. For example, we can show the structure of SKU_DATA and ORDER_ITEM as follows:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

ORDER_ITEM (OrderNumber, *SKU*, Quantity, Price, ExtendedPrice)

This notation indicates that SKU is the primary key of SKU_DATA and that (OrderNumber, SKU) is the primary key of ORDER_ITEM.

In order to function properly, a primary key, whether it is a single column or a composite key, *must* have unique data values inserted into every row of the table. Although this fact may seem obvious, it is significant enough to be named the **entity integrity constraint** and is a fundamental requirement for the proper functioning of a relational database.

BY THE WAY What do you do if a table has no candidate keys? In that case, define the primary key as the collection of all of the columns in the table. Because there are no duplicate rows in a stored relation, the combination of all of the columns of the table will always be unique. Again, although tables generated by SQL manipulation may have duplicate rows, the tables that you design to be stored should never be constructed to have data duplication. Thus, the combination of all columns is always a candidate key.

Surrogate Keys

A **surrogate key** is an artificial column that is added to a table to serve as the primary key. The DBMS assigns a unique value to a surrogate key when the row is created. The assigned value never changes. Surrogate keys are used when the primary key is large and unwieldy. For example, consider the relation RENTAL_PROPERTY:

RENTAL_PROPERTY (Street, City, State/Province, ZIP/PostalCode, Country, Rental_Rate)

The primary key of this table is (Street, City, State/Province, ZIP/PostalCode, Country). As we will discuss further in Chapter 6, for good performance, a primary key should be short and, if possible, numeric. The primary key of RENTAL_PROPERTY is neither.

In this case, the designers of the database would likely create a surrogate key. The structure of the table would then be:

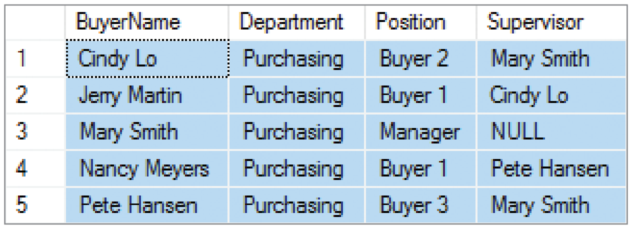
RENTAL_PROPERTY (PropertyID, Street, City, State/Province, ZIP/PostalCode, Country, Rental_Rate)

The DBMS can then be used to assign a numeric value to PropertyID when a row is created (exactly *how* this is done depends upon which DBMS product is being used). Using that key will result in better performance than using the original key. Note that surrogate key values are artificial and have no meaning to the users. In fact, surrogate key values are normally hidden in forms and reports.

For another example, let's look at the Cape Codd BUYER table we created in Chapter 2. The structure of the BUYER table is:

BUYER (BuyerName, Department, Position, *Supervisor*)

The primary key is BuyerName, Supervisor is a foreign key referencing BuyerName in a recursive relationship as discussed in Chapter 2, and the data in the table is:



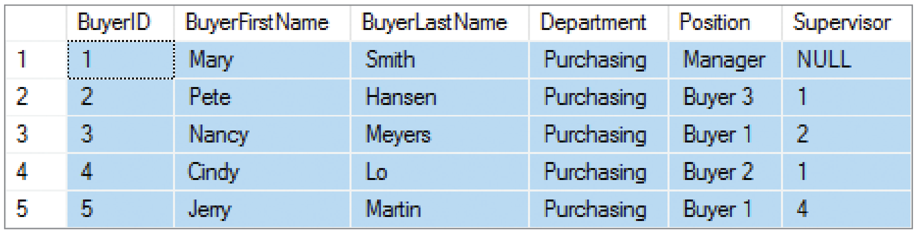
	BuyerName	Department	Position	Supervisor
1	Cindy Lo	Purchasing	Buyer 2	Mary Smith
2	Jerry Martin	Purchasing	Buyer 1	Cindy Lo
3	Mary Smith	Purchasing	Manager	NULL
4	Nancy Meyers	Purchasing	Buyer 1	Pete Hansen
5	Pete Hansen	Purchasing	Buyer 3	Mary Smith

But a primary key must be unique, and BuyerName is only unique because we have so few records in this table—for example, Mary Smith is a common name, and we could easily have multiple Mary Smiths working at Cape Codd as buyers. Another problem is that the BuyerName column actually holds two pieces of data: the Buyer's first name and the Buyer's last name. Good database design dictates that we should split this column into two separate columns: BuyerFirstName and BuyerLastName. Using these two columns as a composite primary key doesn't solve the problem of possible name duplication.

The solution is to use a surrogate primary key and split BuyerName into its components. Thus we get a revised structure of the BUYER table as:

BUYER (BuyerID, BuyerFirstName, BuyerLastName, Department, Position, *Supervisor*)

Note that because we are now using the surrogate primary key BuyerID, the Supervisor column must now hold the numeric values that point to the appropriate BuyerID! Our data in the table now looks like this:



	BuyerID	BuyerFirstName	BuyerLastName	Department	Position	Supervisor
1	1	Mary	Smith	Purchasing	Manager	NULL
2	2	Pete	Hansen	Purchasing	Buyer 3	1
3	3	Nancy	Meyers	Purchasing	Buyer 1	2
4	4	Cindy	Lo	Purchasing	Buyer 2	1
5	5	Jerry	Martin	Purchasing	Buyer 1	4

For the time being, we will continue to use the original BUYER table as the basis for our discussions in this chapter. The techniques we would use to convert the original BUYER table into the redesigned BUYER table are discussed in Chapter 8 on database redesign.

Foreign Keys

A **foreign key** is a column or composite of columns that is the primary key of a table other than the one in which it appears. The term arises because it is a key of a table *foreign* to the one in which it appears as the primary key. In the following two tables, DEPARTMENT.DepartmentName is the primary key of DEPARTMENT, and EMPLOYEE .Department is a foreign key. In this text, we will show foreign keys in italics:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, DepartmentPhone)
EMPLOYEE (EmployeeNumber, LastName, FirstName, *Department*)

Foreign keys express relationships between rows of tables. In this example, the foreign key EMPLOYEE.Department stores the relationship between an employee and his or her department. Note that the foreign key does *not* need to have the same name as the primary key it references—it only has to contain the same type of data!

Consider the SKU_DATA and ORDER_ITEM tables. SKU_DATA.SKU is the primary key of SKU_DATA, and ORDER_ITEM.SKU is a foreign key.

SKU_DATA (SKU, SKU_Description, Department, Buyer)
ORDER_ITEM (OrderNumber, *SKU*, Quantity, Price, ExtendedPrice)

Notice that ORDER_ITEM.SKU is both a foreign key and part of the primary key of ORDER_ITEM. This condition sometimes occurs, but it is not required. In the earlier example, EMPLOYEE.Department is a foreign key, but it is not part of the EMPLOYEE primary key. You will see some uses for foreign keys later in this chapter and the next, and you will study them at length in Chapter 6.

In most cases, we need to ensure that the values of a foreign key match a valid value of a primary key. For the SKU_DATA and ORDER_ITEM tables, we need to ensure that all of the values of ORDER_ITEM.SKU match a value of SKU_DATA.SKU. To accomplish this, we create a **referential integrity constraint**, which is a statement that limits the values of the foreign key. In this case, we create the constraint:

SKU in ORDER_ITEM must exist in SKU in SKU_DATA

This constraint stipulates that every value of SKU in ORDER_ITEM must match a value of SKU in SKU_DATA.

Note that we can have a referential integrity constraint on a recursive relationship between two columns in the same table. The referential integrity constraint for the redesigned BUYER table (the one with BuyerID) discussed earlier in this chapter is:

Supervisor in BUYER must exist in BuyerID in BUYER

BY THE WAY While we have defined a *referential integrity constraint* to require a corresponding primary key value in the linked table, the technical definition of the referential integrity constraint allows for one other option—that the foreign key cell in the table is *empty* and *does not have a value*.[2] If a cell in a table does not have a value, it is said to have a **null value**—for example, see the null value for Mary Smith's Supervisor in the BUYER tables earlier (where it appears in all uppercase as NULL.) We will discuss null values in Chapter 4.

Except for recursive relationships like the one in the BUYER table, it is difficult to imagine a foreign key having null values in a real database when the referential integrity constraint is actually in use, and we will stick with our basic definition of the referential integrity constraint in this book. At the same time, be aware that the complete formal definition of the referential integrity constraint does allow for null values in foreign key columns, and our BUYER table data provides one example of how this can happen.

[2] For example, see the Wikipedia article on referential integrity at *http://en.wikipedia.org/wiki/Referential_integrity*.

BY THE WAY We have defined three constraints so far in our discussion:

- The *domain integrity constraint*
- The *entity integrity constraint*
- The *referential integrity constraint*

The purpose of these three constraints, taken as a whole, is to create **database integrity**, which means that the data in our database will be useful, meaningful data.[3]

Normal Forms

All relations are not equal. Some are easy to process, and others are problematic. Relations are categorized into **normal forms** based on the kinds of problems that they have. Knowledge of these normal forms will help you create appropriate database designs. To understand normal forms, we need first to define modification anomalies.

Modification Anomalies

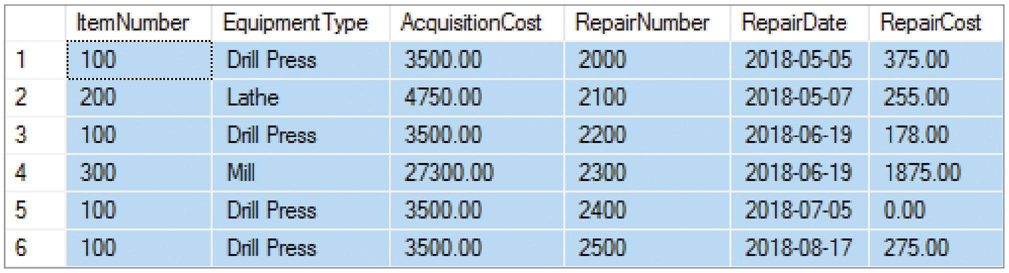
Consider the EQUIPMENT_REPAIR relation in Figure 3-10, which stores data about manufacturing equipment and equipment repairs. Suppose we delete the data for repair number 2100. When we delete this row (the second one in Figure 3-10), we remove not only data about the repair, but also data about the machine itself. We will no longer know, for example, that the machine was a Lathe and that its AcquisitionCost was 4750.00. When we delete one row, the structure of this table forces us to lose facts about two different things: a machine and a repair. This condition is called a **deletion anomaly**.

Now suppose we want to enter the first repair for a piece of equipment. To enter repair data, we need to know not just RepairNumber, RepairDate, and RepairCost, but also ItemNumber, EquipmentType, and AcquisitionCost. If we work in the repair department, this is a problem because we are unlikely to know the value of AcquisitionCost. The structure of this table forces us to enter facts about two entities when we just want to enter facts about one. This condition is called an **insertion anomaly**.

Finally, suppose we want to change existing data. If we alter a value of RepairNumber, RepairDate, or RepairCost, there is no problem. But if we alter a value of ItemNumber, EquipmentType, or AcquisitionCost, we may create a data inconsistency. To see why, suppose we update the last row of the table in Figure 3-10 using the data (100, 'Drill Press', 5500, 2500, '08/17/18', 275).

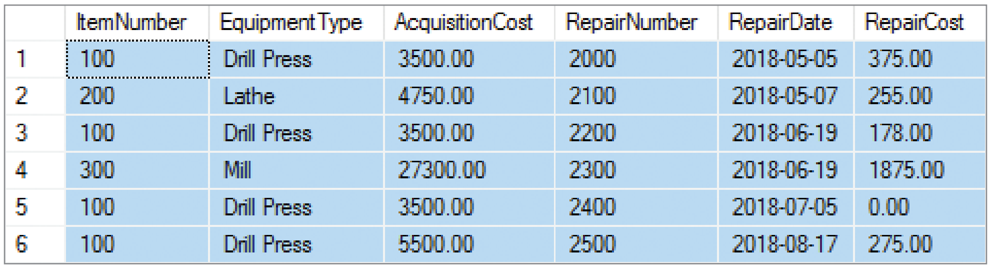
Figure 3-11 shows the table after this erroneous update. The drill press has two different AcquisitionCosts. Clearly, this is an error. Equipment cannot be acquired at two different

FIGURE 3-10 The EQUIPMENT_REPAIR Relation



	ItemNumber	EquipmentType	AcquisitionCost	RepairNumber	RepairDate	RepairCost
1	100	Drill Press	3500.00	2000	2018-05-05	375.00
2	200	Lathe	4750.00	2100	2018-05-07	255.00
3	100	Drill Press	3500.00	2200	2018-06-19	178.00
4	300	Mill	27300.00	2300	2018-06-19	1875.00
5	100	Drill Press	3500.00	2400	2018-07-05	0.00
6	100	Drill Press	3500.00	2500	2018-08-17	275.00

[3] For more information and discussion, see the Wikipedia article on database integrity at *http://en.wikipedia.org/wiki/Database_integrity* and the articles linked to that article.



	ItemNumber	EquipmentType	AcquisitionCost	RepairNumber	RepairDate	RepairCost
1	100	Drill Press	3500.00	2000	2018-05-05	375.00
2	200	Lathe	4750.00	2100	2018-05-07	255.00
3	100	Drill Press	3500.00	2200	2018-06-19	178.00
4	300	Mill	27300.00	2300	2018-06-19	1875.00
5	100	Drill Press	3500.00	2400	2018-07-05	0.00
6	100	Drill Press	5500.00	2500	2018-08-17	275.00

FIGURE 3-11

The EQUIPMENT_REPAIR Relation After an Incorrect Update

costs. If there were, say, 10,000 rows in the table, however, it might be very difficult to detect this error. This condition is called an **update anomaly**.

BY THE WAY Notice that the EQUIPMENT_REPAIR table in Figures 3-10 and 3-11 duplicates data. For example, the AcquisitionCost of the same item of equipment appears several times. Any table that duplicates data is susceptible to update anomalies like the one in Figure 3-11. A table that has such inconsistencies is said to have **data integrity problems**.

As we will discuss further in Chapter 4, to improve query speed, we sometimes design a table to have duplicated data. Be aware, however, that any time we design a table this way, we open the door to data integrity problems.

A Short History of Normal Forms

When Codd defined the relational model, he noticed that some tables had modification anomalies. In his second paper,[4] he defined first normal form, second normal form, and third normal form. He defined **first normal form (1NF)** as the *set of conditions for a relation*, shown in Figure 3-4. Any table meeting the conditions in Figure 3-4 is therefore a relation in 1NF.

This definition, however, brings us back to the "To Key or Not to Key" discussion. Codd's set of conditions for a relation does not require a primary key, but one is clearly implied by the condition that all rows must be unique. Thus, there are various opinions on whether or not a relation has to have a defined primary key to be in 1NF.[5]

For practical purposes, we will define 1NF as it is used in this book as a table that:

1. Meets the set of conditions for a relation, and
2. Has a defined primary key.[6]

Codd also noted that some tables (or, interchangeably in this book, relations) in 1NF had modification anomalies. He found that he could remove some of those anomalies by applying certain conditions. A relation that met those conditions, which we will discuss later in this chapter, was said to be in **second normal form (2NF)**. He also observed, however, that relations in 2NF could also have anomalies, and so he defined **third normal form (3NF)**, which is a set of conditions that removes even more anomalies and which we will also discuss later in this chapter. As time went by, other researchers found

[4] E. F. Codd and A. L. Dean, "Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description," *Access and Control*, San Diego, California, November 11–12, 1971 ACM 1971.

[5] For a review of some of the discussion, see the Wikipedia article at *http://en.wikipedia.org/wiki/First_normal_form*.

[6] Some definitions of 1NF also state that there can be "no repeating groups." This refers to the *multivalue, multicolumn problem* we discuss in Chapter 4 and also deal with in our discussion of *multivalued dependencies* later in this chapter.

still other ways that anomalies can occur, and the conditions for **Boyce-Codd Normal Form (BCNF)** were defined.

These normal forms are defined so that a relation in BCNF is in 3NF, a relation in 3NF is in 2NF, and a relation in 2NF is in 1NF. Thus, if you put a relation into BCNF, it is automatically in the lesser normal forms.

Normal forms 2NF through BCNF concern anomalies that arise from functional dependencies. Other sources of anomalies were found later. They led to the definition of **fourth normal form (4NF)** and **fifth normal form (5NF)**, both of which we will discuss later in this chapter. So it went, with researchers chipping away at modification anomalies, each one improving on the prior normal form.

In 1982, Ronald Fagin published a paper that took a different tack.[7] Instead of looking for just another normal form, Fagin asked, "What conditions need to exist for a relation to have no anomalies?" In that paper, he defined **domain/key normal form (DK/NF)** (and, no. that is not a typo—the slash appears between domain and key in the complete name, but between DK and NF in the acronym.) Fagin ended the search for normal forms by showing that a relation in DK/NF has no modification anomalies and, further, that a relation that has no modification anomalies is in DK/NF. DK/NF is discussed in more detail later in this chapter.

Normalization Categories

As shown in Figure 3-12, normalization theory can be divided into three major categories. Some anomalies arise from functional dependencies, some arise from multivalued dependencies, and some arise from data constraints and odd conditions.

2NF, 3NF, and BCNF are all concerned with anomalies that are caused by functional dependencies. A relation that is in BCNF has no modification anomalies from functional dependencies. It is also automatically in 2NF and 3NF, and, therefore, we will focus on transforming relations into BCNF. However, it is instructive to work through the progression of normal forms from 1NF to BCNF in order to understand how each normal form deals with anomalies, and we will do this later in this chapter.[8]

As shown in the second row of Figure 3-12, some anomalies arise because of another kind of dependency called a multivalued dependency. Those anomalies can be eliminated by placing each multivalued dependency in a relation of its own, a condition known as 4NF. You will see how to do that in the last section of this chapter.

The third source of anomalies is esoteric. These problems involve specific, rare, and even strange data constraints. Accordingly, we will not discuss them in this text.

FIGURE 3-12

Summary of Normalization Theory

Source of Anomaly	Normal Forms	Design Principles
Functional dependencies	1NF, 2NF, 3NF, BCNF	BCNF: Design tables so that every determinant is a candidate key.
Multivalued dependencies	4NF	4NF: Move each multivalued dependency to a table of its own.
Data constraints and oddities	5NF, DK/NF	DK/NF: Make every constraint a logical consequence of candidate keys and domains.

[7] Ronald Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys," *ACM Transactions on Database Systems*, September 1981, pp. 387–415.
[8] See Christopher J. Date, *An Introduction to Database Systems*, 8th ed. (New York: Addison-Wesley, 2003) for a complete discussion of normal forms.

From First Normal Form to Boyce-Codd Normal Form Step by Step

As we discussed earlier in this chapter, a table is in 1NF if and only if (1) it meets the *definition of a relation* in Figure 3-4 and (2) it has a *defined primary key*. From Figure 3-4 this means that the following must hold: the cells of a table must be a single value, and neither repeating groups nor arrays are allowed as values; all entries in a column must be of the same data type; each column must have a unique name, but the order of the columns in the table is not significant; and no two rows in a table may be identical, but the order of the rows is not significant. To this, we add the requirement of having a primary key defined for the table.

Second Normal Form

When Codd discovered anomalies in 1NF tables, he defined 2NF to eliminate some of these anomalies. A relation is in 2NF if and only if *it is in 1NF* and *all non-key attributes are determined by the entire primary key*. This means that if the primary key is a composite primary key, then no non-key attribute can be determined by an attribute or set of attributes that make up only part of the key. Thus, if you have a relation **R (A, B, N, O, P)** with the composite key **(A, B)**, then none of the non-key attributes **N, O**, or **P** can be determined by just **A** or just **B**.

Note that the only way a non-key attribute can be dependent on part of the primary key is if there is a *composite primary key*. This means that relations with *single-attribute primary keys* are automatically in 2NF.

For example, consider the STUDENT_ACTIVITY relation:

STUDENT_ACTIVITY (StudentID, Activity, ActivityFee)

The STUDENT_ACTIVITY relation is in 1NF and is shown with sample data in Figure 3-13. Note that STUDENT_ACTIVITY has the composite primary key (StudentID, Activity), which allows us to determine the fee a particular student will have to pay for a particular activity. However, because fees are determined by activities, ActivityFee is also functionally dependent on just Activity itself, and we can say that ActivityFee is **partially dependent** on the key of the table. The set of functional dependencies is therefore:

(StudentID, Activity) → ActivityFee
Activity → ActivityFee

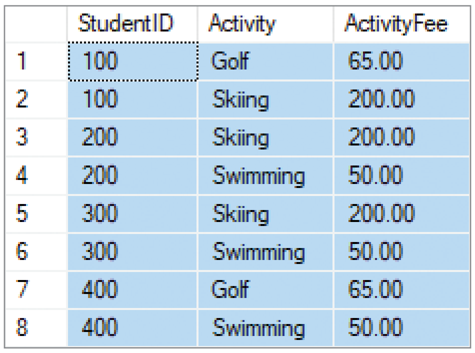
Thus, there is a non-key attribute determined by part of the composite primary key, and the STUDENT_ACTIVITY relation is *not* in 2NF. What do we do in this case? We will have to move the columns of the functional dependency based on the partial primary key attribute into a separate relation while leaving the determinant in the original relation as a foreign key. We will end up with two relations:

STUDENT_ACTIVITY (StudentID, *Activity*)
ACTIVITY_FEE (Activity, ActivityFee)

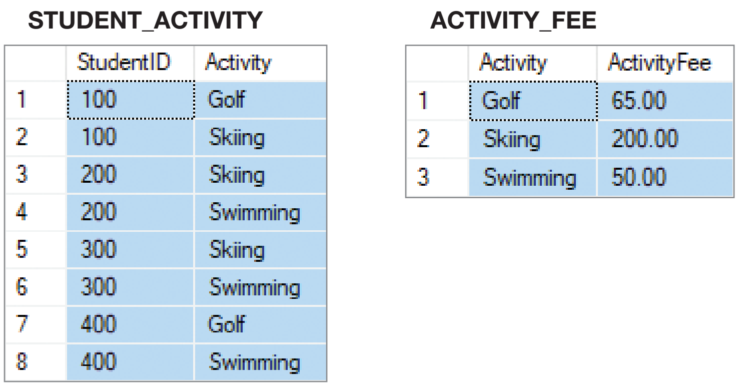
FIGURE 3-13

The 1NF STUDENT_ACTIVITY Relation

STUDENT_ACTIVITY



	StudentID	Activity	ActivityFee
1	100	Golf	65.00
2	100	Skiing	200.00
3	200	Skiing	200.00
4	200	Swimming	50.00
5	300	Skiing	200.00
6	300	Swimming	50.00
7	400	Golf	65.00
8	400	Swimming	50.00



STUDENT_ACTIVITY

	StudentID	Activity
1	100	Golf
2	100	Skiing
3	200	Skiing
4	200	Swimming
5	300	Skiing
6	300	Swimming
7	400	Golf
8	400	Swimming

ACTIVITY_FEE

	Activity	ActivityFee
1	Golf	65.00
2	Skiing	200.00
3	Swimming	50.00

FIGURE 3-14

The 2NF STUDENT_ACTIVITY and ACTIVITY_FEE Relations

The Activity column in STUDENT_ACTIVITY becomes a foreign key. The new relations are shown in Figure 3-14. Now, are the two new relations in 2NF? Yes. STUDENT_ACTIVITY still has a composite primary key, but now has no attributes that are dependent on only a part of this composite key. ACTIVITY_FEE has a set of attributes (just one each in this case) that are dependent on the entire primary key.

Third Normal Form

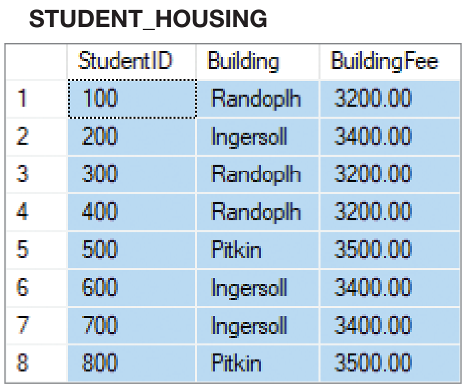
However, the conditions necessary for 2NF do not eliminate all anomalies. To deal with additional anomalies, Codd defined 3NF. A relation is in 3NF if and only if *it is in 2NF* and *there are no non-key attributes determined by another non-key attribute*. The technical name for a non-key attribute determined by another non-key attribute is **transitive dependency**.[9] We can therefore restate the definition of 3NF: a relation is in 3NF if and only if *it is in 2NF* and *it has no transitive dependencies*. Thus, in order for our relation **R (<u>A</u>, <u>B</u>, N, O, P)** to be in 3NF, none of the non-key attributes **N, O**, or **P** can be determined by **N, O**, or **P**.

For example, consider the relation STUDENT_HOUSING shown in Figure 3-15. The STUDENT_HOUSING relation is in 2NF, and the table schema is:

STUDENT_HOUSING (<u>StudentID</u>, Building, BuildingFee)

Here we have a single-attribute primary key, StudentID, so the relation is in 2NF because there is no possibility of a non-key attribute being dependent on only part of the primary key. Furthermore, if we know the student, we can determine the building where he or she is residing, so:

StudentID → Building



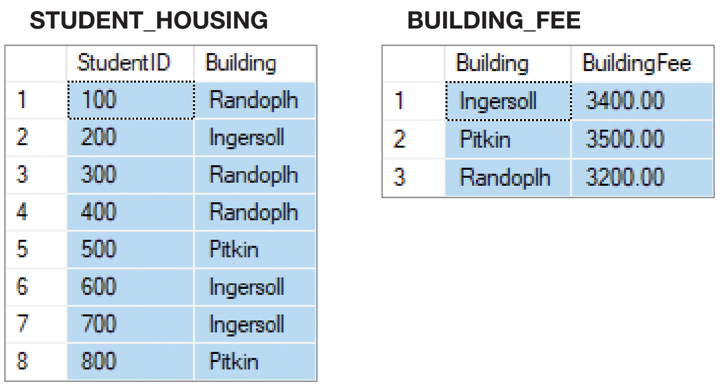
STUDENT_HOUSING

	StudentID	Building	BuildingFee
1	100	Randoplh	3200.00
2	200	Ingersoll	3400.00
3	300	Randoplh	3200.00
4	400	Randoplh	3200.00
5	500	Pitkin	3500.00
6	600	Ingersoll	3400.00
7	700	Ingersoll	3400.00
8	800	Pitkin	3500.00

FIGURE 3-15

The 2NF STUDENT_HOUSING Relation

[9] In terms of functional dependencies, a transitive dependency is defined as: IF A → B and B → C, THEN A → C.



STUDENT_HOUSING

	StudentID	Building
1	100	Randoplh
2	200	Ingersoll
3	300	Randoplh
4	400	Randoplh
5	500	Pitkin
6	600	Ingersoll
7	700	Ingersoll
8	800	Pitkin

BUILDING_FEE

	Building	BuildingFee
1	Ingersoll	3400.00
2	Pitkin	3500.00
3	Randoplh	3200.00

FIGURE 3-16
The 3NF STUDENT_HOUSING and BUILDING_FEE Relations

However, the building fee is independent of which student is housed in the building, and, in fact, the same fee is charged for every room in a building. Therefore, Building determines BuildingFee:

Building → BuildingFee

Thus, a non-key attribute (BuildingFee) is functionally determined by another non-key attribute (Building), and the relation is *not* in 3NF.

To put the relation into 3NF, we will have to move the columns of the functional dependency into a separate relation while leaving the determinant in the original relation as a foreign key. We will end up with two relations:

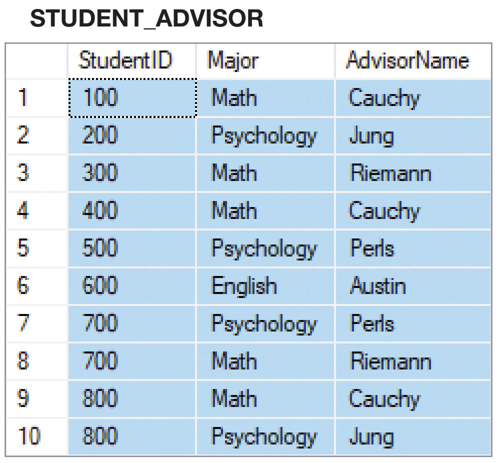
STUDENT_HOUSING (StudentID, *Building*)
BUILDING_FEE (Building, BuildingFee)

The Building column in STUDENT_HOUSING becomes a foreign key. The two relations are now in 3NF (work through the logic yourself to make sure you understand 3NF) and are shown in Figure 3-16.

Boyce-Codd Normal Form

Some database designers normalize their relations to 3NF. Unfortunately, there are still anomalies due to functional dependences in 3NF. Together with Raymond Boyce, Codd defined BCNF to fix this situation. A relation is in BCNF if and only if *it is in 3NF* and *every determinant is a candidate key*.

For example, consider the relation STUDENT_ADVISOR shown in Figure 3-17, where a student (StudentID) can have one or more majors (Major), a major can have one or more



STUDENT_ADVISOR

	StudentID	Major	AdvisorName
1	100	Math	Cauchy
2	200	Psychology	Jung
3	300	Math	Riemann
4	400	Math	Cauchy
5	500	Psychology	Perls
6	600	English	Austin
7	700	Psychology	Perls
8	700	Math	Riemann
9	800	Math	Cauchy
10	800	Psychology	Jung

FIGURE 3-17
The 3NF STUDENT_ADVISOR Relation