CHAPTER 4

The Relational Model and Normalization

The relational model is the industry-standard way to store and process database data. In this chapter, we will discuss the basic terms and concepts of this model. We will also describe normalization, which is a technique for transforming relations that have undesirable characteristics into those that do not have those characteristics. In Chapters 6 through 8, we will describe SQL, a language for defining and processing relations.

The relational model was first proposed by E.F. Codd in a landmark 1970 paper.[1] Initially, the relational model was considered by the industry to be "too theoretical" and too slow and cumbersome for high-speed transaction processing. This objection was gradually overcome, and by the 1980s, thanks to products such as DB2 and Oracle, large relational databases were in use for high-volume transaction processing.

We will begin this chapter with definitions of some key terms.

[1] E.F. Codd, "A Relational Model of Data for Large Shared Databanks," Communications of the ACM, June, 1970, pp. 377–387.

RELATIONS

Chapter 1 stated that relational DBMS products store data in the form of tables. Actually, this is not quite correct. They store data in the form of relations, which are a special type of table. Specifically, a **relation** is a two-dimensional table that has the characteristics listed in Figure 4-1. First, each row in the table holds data that pertain to some entity or a portion of some entity. Second, each column of the table contains data that represent an attribute of the entity. Thus, in an EMPLOYEE relation, each row contains data about a particular employee, and each column contains data that represent an attribute of that employee—such as Name, Phone, or EmailAddress.

In addition, to be a relation, the cells of the table must hold a single value; no repeating elements are allowed in a cell. Also, all of the entries in any column must be of the same kind. For example, if the third column in the first row of a table contains EmployeeNumber, the third column in all other rows must contain EmployeeNumber as well. Further, each column has a unique name, and the order of the columns in the table is unimportant. Similarly, the order of the rows is unimportant. Finally, no two rows in a table may be identical.

Sample Relation and Two Non-relations

Figure 4-2 shows a sample EMPLOYEE table. Consider this table in light of the characteristics shown in Figure 4-1. First, each row is about an EMPLOYEE entity and each column represents an attribute of employees, so those two conditions are met. There is only one value per cell, and all entries in a column are of the same kind. Column names are unique, and we could change the order of either the columns or the rows and not lose any information. Finally, no two rows are identical. Because this table has all the characteristics listed in Figure 4-1, we can classify it as a relation.

Figures 4-3(a) and 4-3(b) show two tables that are not relations. The EMPLOYEE table in Figure 4-3(a) is not a relation because the Phone column has cells with multiple entries. Tom Caruthers has three values for phone, and Richard Bandalone has two. Multiple entries per cell are not allowed in a relation.

FIGURE 4-1

Characteristics of a Relation

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

FIGURE 4-2

Sample Relation

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	236-9987
200	Mary	Abernathy	Finance	MA@somewhere.com	444-8898
300	Liz	Smathers	Finance	LS@somewhere.com	777-0098
400	Tom	Caruthers	Accounting	TC@somewhere.com	236-9987
500	Tom	Jackson	Production	TJ@somewhere.com	444-9980
600	Eleanore	Caldera	Legal	EC@somewhere.com	767-0900
700	Richard	Bandalone	Legal	RB@somewhere.com	767-0900

FIGURE 4-3

Tables but Not Relations (a) Order of Rows is Important and (b) Multiple Entries per Cell

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	236-9987
200	Mary	Abernathy	Finance	MA@somewhere.com	444-8898
300	Liz	Smathers	Finance	LS@somewhere.com	777-0098
400	Tom	Caruthers	Accounting	TC@somewhere.com	236-9987, 266-9987, 555-7171
500	Tom	Jackson	Production	TJ@somewhere.com	444-9980
600	Eleanore	Caldera	Legal	EC@somewhere.com	767-0900
700	Richard	Bandalone	Legal	RB@somewhere.com	767-0900, 767-0011

(a)

EmployeeNumber	FirstName	LastName	Department	Email	Phone
100	Jerry	Johnson	Accounting	JJ@somewhere.com	236-9987
200	Mary	Abernathy	Finance	MA@somewhere.com	444-8898
300	Liz	Smathers	Finance	LS@somewhere.com	777-0098
400	Tom	Caruthers	Accounting	TC@somewhere.com	236-9987
				Fax:	266-9987
				Home:	555-7171
500	Tom	Jackson	Production	TJ@somewhere.com	444-9980
600	Eleanore	Caldera	Legal	EC@somewhere.com	767-0900
				Fax:	236-9987
				Home:	555-7171
700	Richard	Bandalone	Legal	RB@somewhere.com	767-0900

(b)

The table in Figure 4-3(b) is not a relation for two reasons. First, the order of the rows is *not* unimportant. The row under Tom Caruthers contains his fax phone number. If we rearrange the rows, we may lose track of the correspondence between his name and data and his fax phone number. The second reason is that not all values in the Email column are of the same kind. Some of the values are email addresses and others are types of phone numbers.

Note, by the way, that although we can have at most one value in a cell, that value can vary in length. Figure 4-4 shows the table shown in Figure 4-2 with a variable length

FIGURE 4-4

Relation with Variable Length Attribute

EmployeeNumber	FirstName	LastName	Department	Email	Phone	Comment
100	Jerry	Johnson	Accounting	JJ@somewhere.com	236-9987	Joined the Accounting Department in March after completing his MBA at night. Will sit for CPA exam this fall.
200	Mary	Abernathy	Finance	MA@somewhere.com	444-8898	
300	Liz	Smathers	Finance	LS@somewhere.com	777-0098	
400	Tom	Caruthers	Accounting	TC@somewhere.com	236-9987	
500	Tom	Jackson	Production	TJ@somewhere.com	444-9980	
600	Eleanore	Caldera	Legal	EC@somewhere.com	767-0900	
700	Richard	Bandalone	Legal	RB@somewhere.com	767-0900	Is a full-time consultant to legal on a retainer basis.

Comment attribute. Even though this Comment is lengthy and varies in length from row to row, there is still only one Comment per cell. Thus, the table in Figure 4-4 is a relation.

A Note on Terminology

In the database world, people generally use the terms *table* and *relation* interchangeably. Accordingly, from now on we will do the same in this book. Thus, any time we use the term *table,* we mean a table that meets the characteristics listed in Figure 4-1. Keep in mind, however, that strictly speaking there are tables that are not relations.

Sometimes, especially in traditional data processing, people will use the term *file* or *datafile* instead of relation. When they do so, they will use the term *record* for row and the term *field* for column. To further complicate the issue, database theoreticians sometimes use yet another set of terms: they call a relation a *relation*, a column an *attribute*, and a row a *tuple* (rhymes with "couple"). To make things even more confusing, people often mix up these sets of terms. It is typical to hear someone refer to a relation that has rows and fields. As long as you know what is intended, this mixing is not important.

Before we move on, there is one other source of confusion to discuss. According to Figure 4-1, a table that has duplicate rows is not a relation. In practice, however, this condition is often ignored. Particularly when we manipulate relations with the SQL, we may end up with a table that has duplicate rows. To make that table into a relation, we should eliminate the duplicates. On a large table, however, checking for duplication can be very time-consuming. Therefore, the default behavior for DBMS products is not to check for duplicate rows. Hence, in practice, there may be tables with duplicate rows that are still called relations. You will see examples of this situation in Chapter 6.

▶ *TYPES* OF KEYS

A **key** is one or more columns of a relation that identifies a row. A key can be unique or non-unique. For example, for the relation in Figure 4-2, EmployeeNumber is a **unique key** because a value of EmployeeNumber identifies a unique row. Thus, a query to display all EMPLOYEEs having an EmployeeNumber of 200 produce a single row. On the other hand, Department is a **non-unique** key. It is a key because it is used to identify a row, but it is non-unique because a value of Department identifies potentially more than one row. Thus, a query to display all rows having a Department value of 'Accounting' produces several rows.

From the data in Figure 4-2, it appears that EmployeeNumber, LastName, and Email are all unique identifiers. In deciding if this is true, however, it is not sufficient simply to examine sample data. Rather, the developers must ask the users or other subject matter experts whether or not a certain column is unique. The column LastName is a good example. It may turn out that our sample data just happens to have unique values for LastName. In general, the users may say that LastName is not always unique in the EMPLOYEE relation.

Composite Keys

Suppose the users say that LastName is not unique in general, but that the combination of LastName and Department is unique. Somehow, the users know that there will never be two Johnsons, for example, in the Accounting department. If that were the case, then we could say that the combination (LastName, Department) is a unique key. A key that contains two or more attributes is called a **composite key**.

It might turn out that users say that the combination (LastName, Department) is not unique, but that the combination (FirstName, LastName, Department) is unique. Such a key is a composite key with three attributes.

Primary and Candidate Keys

Now, suppose the users tell us that EmployeeNumber is a unique key, that Email is a unique key, and that the combination (FirstName, LastName, Department) is a unique key. As you will learn in the next chapter, when designing a database, we choose one of the unique identifiers to be the **primary key**. The other unique keys are referred to as **candidate keys** because they are candidates to be the primary key.

The primary key is important not only because it can be used to identify unique rows, but also because it is used to represent rows in relationships. (You saw an example of the use of ID keys for Lakeview Equipment Rentals in Chapter 1.) Additionally, many DBMS products use values of the primary key to organize storage for the relation. They also build indexes and other special structures to make it easy (and fast) to use a primary key value to locate a row in physical storage.

Sometimes, relations are denoted by showing the name of the relation followed by the columns of the relation in parentheses. The primary key of the relation is underlined. Thus, the following expression denotes a relation named CUSTOMER, having CustomerID, Name, Email, Phone, and Balance columns:

CUSTOMER (CustomerID, Name, Email, Phone, Balance)

The primary key of the relation is CustomerID. There may also be candidate keys, but they are not shown with this notation.

Functional Dependencies

To understand functional dependency, let's make a short excursion into the world of algebra. Suppose you are buying boxes of cookies and someone tells you that each box costs $4. With this fact, you can compute the cost of several boxes with the formula:

CookieCost = NumberOfBoxes x $5

A more general way to express the relationship between CookieCost and NumberOfBoxes is to say that CookieCost depends upon NumberOfBoxes. Such a statement tells us the character of the relationship of 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 follows:

NumberOfBoxes → CookieCost

This expression can also be read as "NumberOfBoxes determines CookieCost." The variable on the left NumberOfBoxes in this example is called the **determinant.**

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

ExtendedPrice = Quantity * UnitPrice

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

Or the following:

(Quantity, UnitPrice) → ExtendedPrice

In this case, the composite (Quantity, UnitPrice) is the determinant of ExtendedPrice.

Now, let's expand these ideas a bit. Suppose you know that a sack contains red, blue, or yellow objects. Further, suppose you know that the red objects weigh five pounds, the blue objects weigh three pounds, and the yellow objects weigh seven 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 the same way we did previously:

ObjectColor → Weight

Thus, "Weight is functionally dependent on ObjectColor," or "ObjectColor determines Weight." The relationship here does not involve an equation, but this functional dependency is still true. 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 the following:

ObjectColor → Shape

Thus, ObjectColor determines shape. We can put these two together to state the following:

ObjectColor → (Weight, Shape)

Thus, ObjectColor determines Weight and Shape.

A way to represent these facts is to put them into a table as follows:

ObjectColor	Weight	Shape
Red	5	Ball
Blue	3	Cube
Yellow	7	Cube

This table meets all of the conditions shown in Figure 4-1, so we can refer to it as a relation. It has a primary key of ObjectColor. We can express this relation as follows:

OBJECT (ObjectColor, Weight, Shape)

Now, you may be thinking that we have just performed some trick or sleight-of-hand to arrive at a relation, but in truth, one can make the argument that the only reason for having relations is to store instances of functional dependencies. When we have a relation like the following:

PLANT (ItemNumber, VarietyName, Cost, Price)

We are simply storing facts that express the functional dependency:

ItemNumber → (VarietyName, Cost, Price)

Functional Dependencies with Composite Groups Composite groups, such as (Quantity, Price), can occur on either side of a functional dependency. The meaning of the group, however, is different, depending on which side it appears. For example,

OrderNumber → (CustomerNumber, ItemNumber, Quantity)

means that, given a value of OrderNumber, we can determine the values of CustomerNumber, ItemNumber, and Quantity. We can also write this expression as three separate expressions:

OrderNumber → CustomerNumber

OrderNumber→ ItemNumber

OrderNumber → Quantity

On the other hand, the following expression:

(CustomerNumber, ItemNumber, Quantity) → Price

means that given values for CustomerNumber, ItemNumber, Quantity, we can determine Price. Notice that we cannot split this determinant into pieces. For example, it is *not* true that if

(CustomerNumber, ItemNumber, Quantity) → Price

then

CustomerNumber → Price

We need all three parts of the determinant to obtain the value of Price.

Difference between Primary Key and Determinant Before we move on, it is important for you to understand that although a primary key is always a determinant, a determinant is not necessarily a primary key. Consider the following relation

HOUSING (SID, DormName, Fee)

where SID is the identifier of a student, DormName is the name of a dormitory and Fee is the rental cost of living in that dormitory. Assume all students in a particular dorm pay the same Fee.

Because SID is a primary key, we know that SID → (DormName, Fee). Thus, SID is both a primary key and a determinant. Also, because all students in a particular dorm pay the same fee, we know that DormName→ Fee. Thus, DormName is a determinant, but it is not a primary key.

NORMALIZATION

Unfortunately, not all relations are equally desirable. A table that meets the minimum definition of a relation may not have an effective or appropriate structure. For some relations, changing the data can have undesirable consequences, called **modification anomalies**. Anomalies can be eliminated by redefining the relation into two or more relations. In most circumstances, the redefined, or **normalized**, relations are preferred.

Modification Anomalies

Consider the Activity relation in Figure 4-5. If we delete the tuple for Student 100, we will lose not only the fact that Student 100 is a skier but also the fact that skiing costs $200. This is called a **deletion anomaly**; that is, by deleting the facts about one entity (that Student 100 is a skier), we inadvertently delete facts about another entity (that skiing costs $200). With one deletion, we lose facts about two entities.

FIGURE 4-5

Activity Relation

ACTIVITY (SID, Activity, Fee)
Sample Data

SID	Activity	Fee
100	Skiing	200
150	Swimming	50
175	Squash	50
200	Swimming	50

The same relation can be used to illustrate an **insertion anomaly**. Suppose we want to store the fact that scuba diving costs $175, but we cannot enter this data into the ACTIVITY relation until a student takes up scuba diving. This restriction seems silly. Why should we have to wait until someone takes the activity before we can record its price? This restriction is called an insertion anomaly. We cannot insert a fact about one entity until we have an additional fact about another entity.

The relation in Figure 4-5 can be used for some applications, but it obviously has problems. We can eliminate both the deletion and the insertion anomalies by dividing the ACTIVITY relation into two relations, each dealing with a different theme. For example, we can put the SID and Activity attributes into one relation (we will call the new relation STU-ACT for student activity), and we can put the Activity and Fee attributes into a relation called ACT-COST (for activity cost). Figure 4-6 shows the same sample data stored in these two new relations.

Now, if we delete Student 100 from STU-ACT, we do not lose the fact that skiing costs $200. Furthermore, we can add scuba diving and its fee to the ACT-COST relation, even before anyone enrolls. Thus, the deletion and the insertion anomalies have been eliminated.

Separating one relation into two relations has a disadvantage, however. Suppose a student tries to sign up for a nonexistent activity. For instance, Student 250 wants to enroll in racquetball. We can insert this new tuple in STU-ACT (the row would contain 250, Racquetball), but should we? Should a student be allowed to enroll in an activity that is not in the relation ACT-COST? Put another way, should the system somehow prevent student rows from being added if the value of the ACTIVITY is not in the ACT-COST table? The answer to this question lies with the users' requirements. If the action should be prohibited, this constraint (a type of business rule) must be documented as part of the schema design. Later in implementation, the constraint will be defined to the DBMS if the product in use provides such constraint checking. If not, the constraint must be enforced by application programs.

Suppose the user specifies that activities can exist before any student enrolls in them, but no student may enroll in an activity that does not have a fee assigned to it (that is, no activities that are not in the ACT-COST table). We can document this constraint in any of several ways in the database design: Activity in STU-ACT is a subset of Activity in ACT-COST, or STU-ACT [Activity] is a subset of ACT-COST [Activity], or STU-ACT [Activity] $\subseteq$ ACT-COST [Activity].

According to this notation, the brackets [ ] denote a column of data that is extracted from a relation. These expressions simply mean that the values in the Activity attribute

FIGURE 4-6

The Division of ACTIVITY into Two Relations

STU-ACT (SID, Activity)

SID	Activity
100	Skiing
150	Swimming
175	Squash
200	Swimming

ACT-COST (Activity, Fee)

Activity	Fee
Skiing	200
Swimming	50
Squash	50

of STU-ACT must exist in the Activity attribute of ACT-COST. It also means that before we allow an Activity to be entered into STU-ACT, we must check to make sure that it is already present in ACT-COST. Constraints like this are called **referential integrity constraints**.

Essence of Normalization

The anomalies in the ACTIVITY relation shown in Figure 4-5 can be stated in the following intuitive way: Problems occur because ACTIVITY contains facts about two different themes:

- Students who participate in each activity
- How much each activity costs

When we add a new row, we must add data about two themes at once; when we delete a row, we are forced to delete data about two themes at once.

The essence of the normalization process is the following: Every normalized relation should have a single theme. Any relation having two or more themes should be broken up into two or more relations, each of which has a single theme. When we find a relation with modification anomalies, we eliminate them by splitting the relation into two or more separate ones, each containing a single theme.

Every time we break up a relation, however, we may create referential integrity constraints. Hence, remember to check for such constraints every time you break a relation into two or more.

In the remainder of this chapter, you will learn many rules about normalization. All of these rules are special cases of the process just described.

Classes of Relations

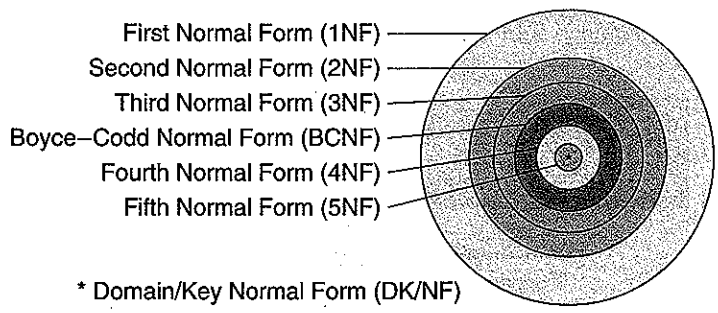
Relations can be classified by the types of modification anomalies to which they are vulnerable. In the 1970s, relational theorists chipped away at these types. Someone would find an anomaly, classify it, and think of a way to prevent it. Each time this happened, the criteria for designing relations improved. These classes of relations and the techniques for preventing anomalies are called **normal forms**. Depending on its structure, a relation may be in first normal form, second normal form, or some other normal form.

As shown in Figure 4-7, these normal forms are nested. That is, a relation in second normal form is also in first normal form, and a relation in 5NF (fifth normal form) is also in 4NF, BCNF, 3NF, 2NF, and 1NF.

These normal forms were helpful, but they had a serious limitation. No theory guaranteed that any of them would eliminate all anomalies; each form could eliminate just certain ones. This changed in 1981, however, when R. Fagin defined a new normal form called **domain/key normal form (DK/NF)**. In an important paper, Fagin showed that a

FIGURE 4-7

Relationship of Normal Forms



relation in DK/NF is free of all modification anomalies, regardless of their type.[2] He also showed that any relation that is free of modification anomalies must be in DK/NF.

Until DK/NF was defined, it was necessary for relational database theorists to continue looking for more and more anomalies and more and more normal forms. Fagin's proof, however, simplified the situation. If we can put a relation in DK/NF, we can be sure that it will have no anomalies. The trick is knowing how to put relations in DK/NF.

FIRST THROUGH FIFTH NORMAL FORMS

Any table of data that meets the definition of a relation is said to be in **first normal form**. Remember that for a table to be a relation, it must have the characteristics listed in Figure 4-1.

The relation shown in Figure 4-5 is in first normal form. As we have seen, however, relations in first normal form may have modification anomalies. To eliminate those anomalies, we split the relation into two or more relations. When we do this, the new relations are in some other normal form—just which one depends on the anomalies we have eliminated, as well as the ones to which the new relations are vulnerable.

Second Normal Form

To understand second normal form, consider the ACTIVITIES relation illustrated in Figure 4-8. This relation has modification anomalies similar to the ones we examined earlier. If we delete the tuple for Student 175, we will lose the fact that squash costs $50. Also, we cannot enter an activity until a student signs up for it. Thus, the relation suffers from both deletion and insertion anomalies.

The problem with this relation is that it has a dependency involving only part of the key. The key is the composite (SID, Activity), but the relation contains a dependency, Activity → Fee. The determinant of this dependency (Activity) is only part of the key (SID, Activity). In this case, we say that Fee is *partially dependent* on the key of the table. There would be no modification anomalies if Fee were dependent on all of the key. To eliminate the anomalies, we must separate the relation into two relations.

This example leads to the definition of second normal form: *A relation is in second normal form if all its non-key attributes are dependent on all of the key.* According to this definition, if a relation has a single attribute as its key, it is automatically in second normal form. Because the key is only one attribute, by default every non-key attribute is dependent on all of the key; there can be no partial dependencies. Thus, second normal is of concern only in relations that have composite keys.

FIGURE 4-8

ACTIVITIES Relation

ACTIVITIES (SID, Activity, Fee)

SID	Activity	Fee
100	Skiing	200
100	Golf	65
150	Swimming	50
175	Squash	50
175	Swimming	50
200	Swimming	50
200	Golf	65

[2] R. Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys," *ACM Transactions on Database Systems*, September 1981, pp. 387–414.

ACTIVITIES can be decomposed to form two relations in second normal form. The relations are the same as those shown in Figure 4-6: namely, STU-ACT and ACT-COST. We know the new relations are in second normal form because they both have single-attribute keys.

Third Normal Form

Relations in second normal form can also have anomalies. Consider the HOUSING relation in Figure 4-9(a). The key is SID, and the functional dependencies are SID → Dorm and Dorm → Fee. These dependencies arise because each student lives in only one dormitory and each dormitory charges only one fee. Everyone living in Randolph Hall, for example, pays $3,200 per quarter.

Since SID determines Dorm and Dorm determines Fee, then indirectly SID → Fee. An arrangement of functional dependencies like this is called a **transitive dependency** because SID determines Fee through the attribute Dorm.

The key of HOUSING is SID (both Dorm and Fee are determined by SID), and hence the relation is in second normal form. Despite this, however, HOUSING has anomalies because of the transitive dependency.

What happens if we delete the second tuple shown in Figure 4-10(a)? We lose not only the fact that Student 150 lives in Ingersoll Hall, but also the fact that it costs $3,100 to live there. This is a deletion anomaly. And how can we record the fact that the Fee for Carrigg Hall is $3,500? We cannot record it until a student decides to move in. This is an insertion anomaly.

To eliminate the anomalies from a relation in second normal form, the transitive dependency must be removed, which leads to a definition of third normal form: *A relation is in third normal form if it is in second normal form and has no transitive dependencies.*

The HOUSING relation can be divided into two relations in third normal form. This has been done for the relations STU-HOUSING (SID, Dorm) and BLDG-FEE (Dorm, Fee) in Figure 4-9(b).

FIGURE 4-9

Elimination of Transitive Dependency (a) Relation with Transitive Dependency and (b) Relations Eliminating the Transitive Dependency

HOUSING (SID, Dorm, Fee)
Key: SID
Functional dependencies: Dorm → Fee
SID → Dorm → Fee

SID	Dorm	Fee
100	Randolph	3200
150	Ingersoll	3100
200	Randolph	3200
250	Pitkin	3100
300	Randolph	3200

(a)

STU-HOUSING (SID, Dorm)

SID	Dorm
100	Randolph
150	Ingersoll
200	Randolph
250	Pitkin
300	Randolph

BLDG-FEE (Dorm, Fee)

Dorm	Fee
Randolph	3200
Ingersoll	3100
Pitkin	3100

(b)

FIGURE 4-10

Boyce–Codd Normal Form (a) Relation in Third Normal Form, but Not in Boyce–Codd Normal Form and (b) Relations in Boyce–Codd Normal Form

ADVISER (SID, Major, Fname)

Key (candidate): (SID, Fname)

Functional dependencies: Fname→Major

SID	Major	Fname
100	Math	Cauchy
150	Psychology	Jung
200	Math	Riemann
250	Math	Cauchy
300	Psychology	Perls
300	Math	Riemann

(a)

STU-ADV (SID, Fname)

SID	Fname
100	Cauchy
150	Jung
200	Riemann
250	Cauchy
300	Perls
300	Riemann

ADV-SUBJ (Fname, Subject)

Fname	Subject
Cauchy	Math
Jung	Psychology
Riemann	Math
Perls	Psychology

(b)

The ACTIVITY relation shown in Figure 4-5 also has a transitive dependency. In ACTIVITY, SID determines Activity and Activity determines Fee. Therefore, ACTIVITY is not in third normal form. Decomposing ACTIVITY into the relations STU-ACT (SID, Activity) and ACT-COST (Activity, Fee) eliminates the anomalies.

Boyce–Codd Normal Form

Unfortunately, even relations in third normal form can have anomalies. Consider the ADVISER relation in Figure 4-10(a). Suppose the requirements underlying this relation are that a student (SID) can have one or more majors (Major), a major can have several faculty members (Fname) as advisers, and a faculty member (Fname) advises in only one major area. Also assume no two faculty members have the same name.

Because students can have several majors, SID does not determine Major. Moreover, because students can have several advisers, SID also does not determine Fname. Thus, SID by itself cannot be a key.

The combination (SID, Major) determines Fname, and the combination (SID, Fname) determines Major. Hence, either of the combinations can be a key. As stated, two or more attributes or attribute collections that can be a key are called candidate keys. Whichever of the candidates is selected to be *the* key is called the primary key.

In addition to the candidate keys, there is another functional dependency to consider: Fname determines Major (any faculty member advises in only one major. Therefore, given the Fname, we can determine the Major). Thus, Fname is a determinant.

By definition, ADVISER is in first normal form. It is also in second normal form because it has no non-key attribute (all attributes are part of at least one key). And it also is in third normal form because it has no transitive dependencies. Despite all this, however, it has modification anomalies.

Suppose Student 300 drops out of school. If we delete Student 300's tuple, we lose the fact that Perls advises in psychology. This is a deletion anomaly. Similarly, how can we store the fact that Keynes advises in economics? We cannot store it until a student majors in economics. This is an insertion anomaly.

Situations like this lead to the definition of Boyce–Codd normal form (BCNF): *A relation is in BCNF if every determinant is a candidate key.* ADVISER is not in BCNF because the determinant, Fname, is not a candidate key.

As with the other examples, ADVISER can be decomposed into two relations having no anomalies. For example, the relations STU-ADV (SID, Fname) and ADV-SUBJ (Fname, Subject) have no anomalies.

Relations in BCNF have no anomalies in regard to functional dependencies and this seemed to put the issue of modification anomalies to rest. It was soon discovered, however, that anomalies can arise from situations other than functional dependencies.

Fourth Normal Form

Consider the STUDENT relation in Figure 4-11, showing the relationship among students, majors, and activities. Suppose that students can enroll in several different majors and participate in several different activities. Because this is so, the only key is the combination of attributes (SID, Major, Activity). Student 100 majors in music and accounting, and she also participates in swimming and tennis. Student 150 majors only in math and participates in jogging.

What is the relationship between SID and Major? It is not a functional dependency because students can have several majors. A single value of SID can have many values of Major. Also, a single value of SID can have many values of Activity.

This attribute dependency is called a **multi-value dependency**. Multi-value dependencies lead to modification anomalies. To begin, note the data redundancy in Figure 4-11. Student 100 has four records, each of which shows one of her majors paired with one of her activities. If the data were stored with fewer rows—say, there were only two rows, one for music and swimming and one for accounting and tennis—the implications would be misleading. It would *appear* that Student 100 swam only when she was a music major and played tennis only when she was an accounting major. But this interpretation is illogical. Her majors and her activities are completely independent of each other. So to prevent such a misleading conclusion, we store all the combinations of majors and activities.

Suppose that because Student 100 decides to sign up for skiing, we add the tuple [100, MUSIC, SKIING], as shown in Figure 4-12(a). The relation at this point implies that Student 100 skis as a music major, but not as an accounting major. In order to keep the data consistent, we must add one row for each of her majors paired with skiing. Thus, we must also add the row [100, ACCOUNTING, SKIING], as illustrated in Figure 4-12(b). This is an update anomaly—too much updating needs to be done to make a simple change in the data.

FIGURE 4-11

Relation with Multi-Value Dependencies

STUDENT (SID, Major, Activity)

Multi-value dependencies: SID →→ Major
SID →→ Activity

SID	Major	Activity
100	Music	Swimming
100	Accounting	Swimming
100	Music	Tennis
100	Accounting	Tennis
150	Math	Jogging

FIGURE 4-12

STUDENT Relations with Insertion Anomalies (a) Insertion of a Single Tuple and (b) Insertion of Two Tuples

STUDENT (SID, Major, Activity)

SID	Major	Activity
100	Music	Skiing
100	Music	Swimming
100	Accounting	Swimming
100	Music	Tennis
100	Accounting	Tennis
150	Math	Jogging

(a)

SID	Major	Activity
100	Music	Skiing
100	Accounting	Skiing
100	Music	Swimming
100	Accounting	Swimming
100	Music	Tennis
100	Accounting	Tennis
150	Math	Jogging

(b)

In general, a multi-value dependency exists when a relation has at least three attributes, two of them are multi-value, and their values depend on only the third attribute. In other words, in a relation R (A, B, C), a multi-value dependency exists if A determines multiple values of B, A determines multiple values of C, and B and C are independent of each other. As we saw in the previous example, SID determines multiple values of Major and SID determines multiple values of Activity, but Major and Activity are independent of each other.

Refer to Figure 4-11. Notice how multi-value dependencies are written: SID $\rightarrow\rightarrow$ Major, and SID $\rightarrow\rightarrow$ Activity. This is read as follows: "SID multi-determines Major, and SID multi-determines Activity." This relation is in BCNF (2-NF because it is all key; 3NF because it has no transitive dependencies; and BCNF because it has no non-key determinants). However, as we have seen, it has anomalies: If a student adds a major, we must enter a row for the new major, paired with each of the student's activities. The same holds true if a student enrolls in a new activity. If a student drops a major, we must delete each of his rows containing that major. If he participates in four activities, there are four rows containing the major he has dropped, and each of them must be deleted.

To eliminate these anomalies, we must eliminate the multi-value dependency. We do this by creating two relations, each one storing data for only one of the multi-value attributes. The resulting relations do not have anomalies. They are STU-MAJOR (SID, Major) and STU-ACT (SID, Activity), as seen in Figure 4-13.

From these observations, we define fourth normal form in the following way: *A relation is in fourth normal form if it is in BCNF and has no multi-value dependencies.* After

FIGURE 4-13

Elimination of Multi-Value Dependency

STU-MAJOR (SID, Major)

SID	Major
100	Music
100	Accounting
150	Math

STU-ACT (SID, Activity)

SID	Activity
100	Skiing
100	Swimming
100	Tennis
150	Jogging

we have discussed domain/key normal form later in this chapter, we will return to describe multi-value dependencies in another more intuitive way.

Fifth Normal Form

Fifth normal form concerns dependencies that are rather obscure. It has to do with relations that can be divided into subrelations, as we have been doing, but then cannot be reconstructed. The condition under which this situation arises has no clear intuitive meaning. We do not know what the consequences of such dependencies are or even if they have any practical consequences. For more information about fifth normal form, refer to *SQL for Smarties,* pp. 34, 35.[3]

DOMAIN/KEY NORMAL FORM

Each of the normal forms we have discussed was identified by researchers who found anomalies with some relations that were in a lower normal form: Noticing modification anomalies with relations in second normal form led to the definition of third normal form, and so on. Although each normal form solved some of the problems that had been identified with the previous one, no one could know what problems had not yet been identified. With each step, progress was made toward a well-structured database design, but no one could guarantee that no more anomalies would be found. In this section, we study a normal form that guarantees that there will be no anomalies of any type. When we put relations into that form, we know that even the obscure anomalies associated with fifth normal form cannot occur.

In 1981, Fagin published an important paper in which he defined domain/key normal form (DK/NF).[4] He showed that a relation in DK/NF has no modification anomalies and that a relation having no modification anomalies must be in DK/NF. This finding establishes a bound on the definition of normal forms, so no higher normal form is needed—at least in order to eliminate modification anomalies.

Equally important, DK/NF involves only the concepts of key and domain—concepts that are fundamental and close to the heart of database practitioners. They are readily supported by DBMS products (or could be, at least). In a sense, Fagin's work formalized and justified what many practitioners believed intuitively, but were unable to express precisely.

Definition

In concept, DK/NF is quite simple: A relation is in DK/NF if every constraint on the relation is a logical consequence of the definition of keys and domains. Consider the important terms in this definition: constraint, key, and domain.

Constraint in this definition is intended to be very broad. Fagin defines a constraint as any rule governing static values of attributes that is precise enough so that we can ascertain whether or not it is true. Edit rules, intrarelation and interrelation constraints, functional dependencies, and multi-value dependencies are examples of such constraints. Fagin expressly excludes constraints pertaining to changes in data values or time-dependent constraints. For example, the rule "Salesperson salary in the current period can never be less than salary in the prior period" is excluded from Fagin's definition of constraint. Except for time-dependent constraints, Fagin's definition is both broad and inclusive.

[3] Celko, Joe. SQL for Smarties, Second Edition. *San Francisco: Morgan Kaufman, 2000.*
[4] Fagin, pp. 387–414.

A **key** is a unique identifier of a row, as we have already defined. The third significant term in the definition of DK/NF is **domain**. In Chapter 2, we defined domain as a named set of possible attribute values. Fagin's proof says a domain constraint has been satisfied if attribute values meet the restrictions of the domain definition.

Informally, a relation is in DK/NF if enforcing key and domain restrictions causes all of the constraints to be met. Moreover, because relations in DK/NF cannot have modification anomalies, the DBMS can prohibit them by enforcing key and domain restrictions.

Unfortunately, there is no known algorithm for converting a relation to DK/NF, nor is it even known which relations can be converted to DK/NF. Finding or designing DK/NF relations is more art than a science.

In spite of this, DK/NF is an exceedingly useful design objective in the practical world of database design. If we can define relations so that constraints on them are logical consequences of domains and keys, there will be no modification anomalies. For many designs, this objective can be accomplished. When it cannot, the constraints must be built into the logic of application programs that process the database.

The following three examples illustrate DK/NF.

Example 1 of Domain/Key Normal Form

Consider the STUDENT relation in Figure 4-14, which contains attributes SID, GradeLevel, Dorm, and Fee. Dorm is the dorm in which the student lives, and Fee is the amount the student pays to live in that dorm.

SID functionally determines the other three attributes, so SID is a key. Assume we also know from the requirements definition that Dorm → Fee and that SIDs must not begin with 1. If we can express these constraints as logical consequences of domain and key definitions, we can be certain, according to Fagin's theorem, that there will be no modification anomalies. For this example, it will be easy.

To enforce the constraint that student numbers not begin with 1, we simply define the domain for student numbers to incorporate this constraint (see Figure 4-15). Enforcing the domain restriction guarantees that this constraint will be met.

Next, we need to make the functional dependency Dorm → Fee a logical consequence of keys. If Dorm were a key attribute, Dorm → Fee would be a logical consequence of a key. Therefore, the question becomes how to make Dorm a key. It cannot be

FIGURE 4-14

Example 1 of DK/NF

STUDENT (SID, GradeLevel, Dorm, Fee)

Key: SID

Constraints: Dorm → Fee
SID must not begin with digit 1

FIGURE 4-15

Domain Key Definition for Example 1

Domain Definitions:

Attribute	Domain Name	Values
SID	StudentID	4 decimal digits, first digit not 1
GradeLevel	StudentYear	('FR', 'SO', 'JR', 'SN', 'GR')
Dorm	BuildingNames	Char(4)
Fee	StudentFees	Any currency value

Relation and Key Definitions:

STUDENT (SID, GradeLevel, Dorm)

BLDG-FEE (Dorm, Fee)

a key in STUDENT because more than one student lives in the same dorm, but it can be a key of its own relation. Thus, we define the relation BLDG-FEE with Dorm and Fee as its attributes. Dorm is the key of this relation. Having defined this new relation, we can remove Fee from STUDENT. The final domain and relation definitions for this example appear in Figure 4-15.

This is the same result we obtained when converting a relation from 2NF to 3NF to remove transitive dependencies. In this case, however, the process was simpler and the result more robust. It was simpler because we did not need to know that we were eliminating a transitive dependency. We simply needed to find creative ways to make all the constraints logical consequences of domain and key definitions. The result was more robust because when converting the relation to 3NF, we knew only that it had fewer anomalies than when it was in 2NF. By converting the relation to DK/NF, we know that the relations have no modification anomalies whatsoever.

Example 2 of Domain/Key Normal Form

The next more complicated example involves the relation shown in Figure 4-16. The PROFESSOR relation contains data about professors, the classes they teach, and the students they advise. FID (for Faculty ID) and Fname uniquely identify a professor. SID uniquely identifies a student, but Sname does not necessarily identify a SID. Professors can teach several classes and advise several students, but a student is advised by only one professor. FIDs start with a 1, but SIDs must not start with a 1.

These statements can be expressed more precisely by the functional and multi-value dependencies shown in Figure 4-16. FID and Fname functionally determine each other (in essence, they are equivalent). FID and Fname multi-determine Class and SID. SID functionally determines FID and Fname. SID determines Sname.

In more complex examples such as this one, it is helpful to consider DK/NF from a more intuitive light. Remember that the essence of normalization is that every relation should have a single theme. Considered from this perspective, there are three themes in PROFESSOR. One is the correspondence between FIDs and Fnames; the second concerns the classes that a professor teaches; and the third concerns the identification number, name, and adviser of a given student.

Figure 4-17 shows three relations that reflect these themes. The FACULTY relation represents the equivalence of FID and Fname. FID is the key and Fname is an alternative key, which means that both attributes are unique to the relation. Because both are keys, the functional dependencies FID → Fname and Fname → FID are logical consequences of keys.

The PREPARATION relation contains the correspondence of faculty and classes; it shows the classes that a professor is prepared to teach. The key is the combination (Fname, Class). Both attributes are required in the key because a professor may teach several classes, and a class may be taught by several professors. Finally, STUDENT represents the student and adviser names for a particular SID. Observe that each of these relations has a

FIGURE 4-16

Example 2 of DK/NF

PROFESSOR (FID, Fname, Class, SID, Sname)

Constraints:
- FID → Fname
- Fname → FID
- FID →→ Class | SID
- Fname →→ Class | SID
- SID → FID
- SID → Fname
- SID → Sname
- FID must start with 1; SID must not start with 1

FIGURE 4-17

Domain Key Definition for Example 2

Domain Definitions:

Attribute	Domain Name	Values
FID	FacultyID	4 decimal digits, first digit is 1
Fname	PersonNames	Char(50)
Class	ClassNames	Char(10); values {list of valid course names}
SID	StudentID	4 decimal digits, first digit is not 1
Sname	PersonNames	Char(50)

Relation and Key Definitions:

FACULTY (FID, Fname)

Candidate key: Fname

PREPARATION (Fname, Class)

STUDENT (SID, Sname, Fname)

single theme. These relations express all of the constraints of Figure 4-16 as a logical consequence of domains and key definitions. These relations are, therefore, in DK/NF.

Note that separating the PREPARATION theme from the STUDENT theme has eliminated the multi-value dependencies. When we examined fourth normal form, we found that in order to eliminate multi-value dependencies, we had to separate the multi-value attributes into different relations. Our approach here is to break a relation with several themes into several relations, each with one theme. In doing that, we eliminated a multi-value dependency. In fact, we arrived at the same solution using both approaches.

Example 3 of Domain/Key Normal Form

The next example concerns a situation that was not addressed by any of the other normal forms, but occurs frequently in practice. This relation has a constraint among data values within a tuple that is neither a functional dependency nor a multi-value dependency.

Consider the constraints in the relation STU-ADVISER illustrated in Figure 4-18. This relation contains information about a student and his or her adviser. SID determines Sname, FID, Fname, and GradFacultyStatus, so it is therefore the key. FID and Fname identify a unique faculty member and are equivalent to each other, as in Example 2. Both FID and Fname determine GradFacultyStatus. Finally, the new type of constraint is that only members of the graduate faculty are allowed to advise graduate students.

FIGURE 4-18

Example 3 of DK/NF

STU-ADVISER (SID, Sname, FID, Fname, GradFacultyStatus)

Key: SID

Constraints:
- FID → Fname
- Fname → FID
- FID and Fname → GradFacultyStatus
- Only graduate faculty can advise graduate students
- FID begins with 1
- SID must not begin with 1
- SID of graduate student begins with 9
- GradFacultyStatus = { 0 for undergraduate faculty; 1 for graduate faculty }

The domain restrictions are that SID must not begin with a 1, SID must begin with a 9 for graduate students, FID must begin with a 1, and GradFacultyStatus is 0 for undergraduate faculty and 1 for graduate faculty. With these domain definitions, the constraint that graduate students must be advised by graduate faculty can be expressed as a constraint on row values. Specifically, if the SID starts with 9, the value of GradFacultyStatus must be 1.

To put this relation in DK/NF, we proceed as in Example 2. What are the basic themes of this relation? There is one regarding faculty personnel that relates FID, Fname, and GradFacultyStatus. Because FID and Fname determine GradFacultyStatus, either of these attributes can be the key, and this relation is in DK/NF (see Figure 4-19).

Now, consider the data regarding students and advisers. Although it may first appear that there is only one theme, that of advising, the constraint that only graduate faculty can advise graduate students implies otherwise. Actually, there are two themes: graduate advising and undergraduate advising. Thus, Figure 4-19 contains a G_ADV relation for graduate students and a UG_ADV relation for undergraduates. Look at the domain definitions: GSID starts with a 9, Gfname is the Fname of a FACULTY tuple with GradFacultyStatus equal to 1, and UGSID must not begin with 1 or 9. All the constraints described in Figure 4-18 are implied by the key and domain definitions shown in Figure 4-19. These relations are therefore in DK/NF and have no modification anomalies.

To summarize the discussion of normalization, Figure 4-20 lists the normal forms and presents the defining characteristic of each.

FIGURE 4-19

Doman Key Definition for Example 3

Domain Definitions:

Attribute	Domain Name	Values
FID	FacultyID	4 decimal digits, first digit is 1
Fname	PersonNames	Char(50)
GradFacultyStatus	FacultyStatus	Values {0, 1}
GSID	GradStudentID	4 decimal digits, first digit is 9
UGSID	UnderGradStudentID	4 decimal digits, first digit is not 1 and not 9
Sname	PersonNames	Char(50)
Gfname	PersonNames	Values {FACULTY.Fname where GradFacultyStatus = 1}

Relation and Key Definitions:

FACULTY (<u>FID</u>, Fname, GradFacultyStatus)

Candidate key: Fname

G_ADV (<u>GSID</u>, Sname, Gfname)

UG_ADV (<u>UGSID</u>, Sname, Fname)

FIGURE 4-20

Summary of Normal Forms

Form	*Defining Characteristic*
1NF	Any relation
2NF	All nonkey attributes are dependent on all of each key.
3NF	There are no transitive dependencies.
BCNF	Every determinant is a candidate key.
4NF	There are no multi-valued dependencies.
5NF	Not described in this discussion
DK/NF	All constraints on relations are logical consequences of domains and keys.

THE SYNTHESIS OF RELATIONS

In the previous section, we approached relational design from an analytical perspective. The questions we asked were these: "Given a relation, is it in good form? Does it have modification anomalies?" In this section, we look at relational design from a different perspective—a synthetic one. From this perspective, we ask, "Given a set of attributes with certain functional dependencies, what relations should we form?"

First, observe that two attributes (A and B, for example) can be related in three ways:

1. They determine each other:
   A → B and B → A
   Hence, A and B have a one-to-one attribute relationship.
2. One determines the other.
   A → B, but B not → A
   Hence, A and B have a many-to-one relationship.
3. They are functionally unrelated.
   A not → B and B not → A
   Hence, A and B have a many-to-many attribute relationship.

One-to-One Attribute Relationships

If A determines B and B determines A, the values of the attributes have a one-to-one relationship. This must be because if A determines B, the relationship between A and B is many-to-one. It is also true, however, that if B determines A, the relationship between B and A must be many-to-one. For both statements to be true at the same time, the relationship between A and B must actually be one-to-one (which is a special case of many-to-one), and the relationship between B and A is also actually one-to-one. Therefore, the relationship is one-to-one.

This case is illustrated by FID and Fname in Examples 2 and 3 in the previous section on domain/key normal form. Each of these attributes uniquely identifies a faculty person. Consequently, one value of FID corresponds to exactly one value of Fname, and vice versa.

Three equivalent statements can be drawn from the example of FID and Fname:

- If two attributes functionally determine each other, the relationship of their data values is one-to-one.
- If two attributes uniquely identify the same entity, the relationship of their data values is one-to-one.
- If two attributes have a one-to-one relationship, they functionally determine each other.

When creating a database with attributes that have a one-to-one relationship, the two attributes must occur together in at least one relation. Other attributes that are functionally determined by these (an attribute that is functionally determined by one of them is functionally determined by the other as well) may also reside in this same relation.

Consider FACULTY (FID, Fname, GradFacultyStatus) in Example 3 in the previous section. FID and Fname determine each other. GradFacultyStatus can also occur in this relation because it is determined by FID and Fname. Attributes that are not functionally determined by these attributes may not occur in a relation with them. Consider the relations FACULTY and PREPARATION in Example 2, in which both FID and Fname occur in FACULTY, but Class (from PREPARATION) may not. Class can have multiple values for a faculty member, so Class is not dependent on FID or Fname. If we added Class to the FACULTY relation, the key of FACULTY would need to be either (FID, Class) or (Fname, Class). In this case, however, FACULTY would not be in

DK/NF because the dependencies between FID and Fname would not be logically implied by either of the possible keys.

These statements are summarized in the first column of Figure 4-21, and the record definition rules are listed in Figure 4-22. If A and B have a one-to-one relationship, they can reside in the same relation, say R. A determines B and B determines A. The key of the relation can be either A or B. A new attribute, C, can be added to R if either A or B functionally determines C.

Attributes having a one-to-one relationship must exist together in at least one relation in order to establish their equivalence (FID of 198, for example, refers to Professor Heart). It is generally undesirable to have them occur together in more than one relation, however, because this causes needless data duplication. Often, one or both of the two attributes occur in other relations. In Example 2, Fname occurs in both PREPARATION and STUDENT. Although it is possible to place Fname in PREPARATION and

FIGURE 4-21

Summary of Three Types of Attribute Relationships

	Type of Attribute Relationship		
	One to One	Many to One	Many to Many
Relation Definition*	R(A, B)	S(C, D)	T(E, F)
Dependencies	A -> B B -> A	C -> D	E -> F F -> E
Key	Either A or B	C	(E, F)
Rule for Adding Another Attribute	Either A or B -> C	C -> F	(E, F) -> G

* The letters used in these relation definitions match those used in Figure 4-22.

FIGURE 4-22

Summary of Rules for Constructing Relations

Concerning One-to-One Attribute Relationships

- Attributes that have a one-to-one relationship must occur together in at least one relation. Call the relation *R* and the attributes *A* and *B*.
- Either *A* or *B* must be the key of *R*.
- An attribute can be added to *R* if it is functionally determined by *A* or *B*.
- An attribute that is not functionally determined by *A* or *B* cannot be added to *R*.
- *A* and *B* must occur together in *R*, but should not occur together in other relations.
- Either *A* or *B* should be consistently used to represent the pair in relations other than *R*.

Concerning Many-to-One Attribute Relationships

- Attributes that have a many-to-one relationship can exist in a relation together. Assume *C* determines *D* in relation *S*.
- *C* must be the key of *S*.
- An attribute can be added to *S* if it is determined by *C*.
- An attribute that is not determined by *C* cannot be added to *S*.

Concerning Many-to-Many Attribute Relationships

- Attributes that have a many-to-many relationship can exist in a relation together. Assume two such attributes, *E* and *F*, reside together in relation *T*.
- The key of *T* must be (*E*, *F*).
- An attribute can be added to *T* if it is determined by the combination (*E*, *F*).
- An attribute may not be added to *T* if it is not determined by the combination (*E*, *F*).
- If adding a new attribute, *G*, expands the key to (*E*, *F*, *G*), then the theme of the relation has been changed. Either *G* does not belong in *T* or the name of *T* must be changed to reflect the new theme.

FID in STUDENT, this generally is bad practice because when attributes are paired in this way, one of them should be selected to represent the pair in all other relations. Fname was selected in Example 2.

Many-to-One Attribute Relationships

If attribute A determines B, but B does not determine A, the relationship among their data values is many-to-one. In the adviser relationship in Example 2, SID determines FID. Many students (SID) are advised by a faculty member (FID), but each student is advised by only one faculty member. This then is a many-to-one relationship.

For a relation to be in DK/NF, all constraints must be implied by keys, so every determinant must be a key. If A, B, and C are in the same relation, and if A determines B, then A must be the key (meaning it also determines C). If instead (A, B) determines C, (A, B) must be the key. In this latter case, no other functional dependency, such as A determines B, is allowed.

You can apply these statements to database design in the following way: If A determines B when constructing a relation, the only other attributes you can add to the relation must also be determined by A. For example, suppose you put SID and Dorm together in a relation called STUDENT. You may add any other attribute determined by SID, such as Sname, to this relation. But if the attribute Fee is determined by Dorm, you may not add it to this relation. Fee can be added only if SID → Fee.

These statements are summarized in the center column of Figure 4-21. If C and D have an N:1 relationship, they may reside together in a relation (S, for example). C will determine D, but D will not determine C. The key of S will be C. Another attribute, E, can be added to S only if C determines E.

Many-to-Many Attribute Relationships

If A does not determine B and B does not determine A, the relationship among their data values is many-to-many. In Example 2, Fname and Class have a many-to-many relationship. A professor teaches many classes and a class is taught by many professors. In a many-to-many relationship, both attributes must be a key of the relation. For instance, the key of PREPARATION in Example 2 is the combination (Fname, Class).

When constructing relations that have multiple attributes as keys, you can add new attributes that are functionally dependent on all of the key. NumberOfTimesTaught is functionally dependent on both (Fname, Class) and can be added to the relation. FacultyOffice, however, cannot be added because it would be dependent only on Fname, not on Class. If FacultyOffice needs to be stored in the database, it must be added to the relation regarding faculty, not to the relation regarding preparations.

These statements are summarized in the right column of Figure 4-21. If E and F have an M:N relationship, E does not determine F and F does not determine E. Both E and F can be put into a relation T; if this is done, the key of T will be the composite (E, F). A new attribute, G, can be added to T if it is determined by all of (E, F). It cannot be added to T if it is determined by only one of E or F.

Consider a similar example. Suppose we add Classroom Number to PREPARATION. Is ClassroomNumber functionally determined by the key of PREPARATION, (Fname, Class)? Most likely it is not because a professor could teach a particular class in many different rooms.

The composite (Fname, Class) and ClassroomNumber have an M:N relationship. Because this is so, the rules in Figure 4-21 can be applied, but with E representing (Fname, Class) and F representing ClassroomNumber. Now we can compose a new relation, T, with attributes Fname, Class, and ClassroomNumber. The key becomes (Fname, Class, ClassroomNumber). In this situation, we have created a new relation with a new theme. Consider relation T, which contains faculty names, classes, and classroom num-

bers. The theme of this relation is therefore no longer PREPARATION, but instead is WHO-WHAT-WHERE-TAUGHT.

Changing the theme may or may not be appropriate. If ClassroomNumber is important, the theme does need to be changed. In that case, PREPARATION is the wrong relation, and WHO-WHAT-WHERE-TAUGHT is a more suitable theme.

On the other hand, depending on user requirements, PREPARATION may be completely suitable as it is. If so, then if ClassroomNumber belongs in the database at all, it should be located in a different relation—perhaps SECTION-NUMBER, CLASS-SECTION, or some similar relation.

MULTI-VALUE DEPENDENCIES, ITERATION 2

The discussion about many-to-many attribute value relationships may make the concept of multi-value dependencies easier to understand. The problem with the relation STUDENT (SID, Major, Activity) in Figure 4-11 is that it has *two* different many-to-many relationships—one between SID and Major and the other between SID and Activity. Clearly, a student's various majors have nothing to do with his or her various activities. Putting both of these many-to-many relationships in the same relation, however, makes it appear as if there is some association.

Major and Activity are independent, and there is no problem if a student has only one of each. SID functionally determines Major and Activity, and the relation is in DK/NF. In this case, both the relationships between Major and SID and Activity and SID are many-to-one.

Another way of perceiving the difficulty is to examine the key (SID, Major, Activity). Because STUDENT has many-to-many relationships, all of the attributes have to be in the key. Now what theme does this key represent? We might say the combination of a student's studies and activities. But this is not one thing; it is plural. One row of this relation describes only part of the combination, and we need all of the rows about a particular student in order to get the whole picture. *In general, a row should have all of the data about one instance of the relation's theme.* A row of Customer, for example, should have all the data we want about a particular customer.

Consider PREPARATION in Example 2 in the section on domain/key normal form. The key is (Fname, Class). The theme this represents is that a particular professor is prepared to teach a particular class. We need only one row of the relation to get all of the information we have about the combination of that professor and that class. Looking at more rows does not generate any more information about it.

As you know, the solution to the multi-value dependency constraint problem is to split the relation into two relations, each with a single theme. STU-MAJOR shows the combination of a student and a major. Everything we know about the combination is in a single row, and we will not gain more information about that combination by examining more rows.

DE-NORMALIZED DESIGNS

The techniques of normalization are designed to eliminate anomalies by reducing data duplication to key values. Although this goal is usually appropriate, there are situations in which de-normalized designs are preferred. In particular, when a normalized design is unnatural, awkward, or results in unacceptable performance, a de-normalized design is better.

Normalization Unnatural

Consider the relation:

CUSTOMER (CustNumber, CustName, City, State, Zip)

This relation is not in domain/key normal form because it contains the functional dependency Zip → (City, State) that is not implied by CustNumber, the key. Thus, there is a constraint that is not implied by domains and keys.

Following the usual normalization procedure, we can break this relation into two relations that are in domain/key normal form:

CUSTOMER (CustNumber, CustName, Zip)

CODES (Zip, City, State)

with the referential integrity constraint that values of CUSTOMER.Zip must already exist in CODES.Zip.

These two tables are in domain/key normal form, but they most likely do not represent a better design. Every time a user wants to know a customer's city and state, a lookup in the CODES table (meaning to read the row that has the appropriate city and state data) is required. If the design were not normalized, the city and state data would be stored with the rest of the customer data and no lookup would be necessary. Further, the disadvantages of duplicating the city and state data are probably not very important.

Normalization Awkward

For another example of de-normalization, consider the following relation:

COLLEGE (CollegeName, Dean, AssistantDean)

and suppose that a college has one dean and from one to three assistant deans. In this case, the key of the table is (CollegeName, AssistantDean). This table is not in domain/key normal form because the constraint CollegeName → Dean is not a logical consequence of the table's key.

COLLEGE can be normalized into the following relation:

DEAN (CollegeName, Dean)

and the following relation:

ASSISTANT-DEAN(CollegeName, AssistantDean)

But now, whenever a database application needs to obtain data about the college, it must read at least two rows and possibly as many as four rows of data. An alternative to this design is to place all three AssistantDeans into the COLLEGE table, each in a separate attribute. The table is then the following:

COLLEGE1 (CollegeName, Dean, AssistantDean1, AssistantDean2, AssistantDean3)

COLLEGE1 is in domain/key normal form because all of its attributes are functionally dependent on the key CollegeName. But something has been lost. To see what is lost, suppose that you want to determine the names of the COLLEGEs that have an assistant dean named 'Mary Abernathy.' To do this, you have to look for this value in each of the three AssistantDean columns. Your query would appear something like this:[5]

[5] These statements are examples of SQL, a relational language that we will discuss in detail in Chapters 6 and 7. For now, just think of them intuitively; you will learn the format of them in that chapter.

```
SELECT      CollegeName
FROM        COLLEGE1
WHERE       AssistantDean1 = 'Mary Abernathy' OR
            AssistantDean2 = 'Mary Abernathy' OR
            AssistantDean3 = 'Mary Abernathy'
```

Using the normalized design with ASSISTANT-DEAN, you would need only to state the following:

```
SELECT      CollegeName
FROM        ASSISTANT-DEAN
WHERE       AssistantDean = 'Mary Abernathy'
```

Thus, there are three possible designs. The first design is un-normalized and duplicates dean data. The second one is normalized, but it requires processing two to four rows to obtain all of the college data; the third is normalized, but it is awkward to process.

Which design is best? It depends on the requirements, the workload, the size of the database, and so forth. If few colleges have more than one assistant dean, the first design is probably acceptable. If any college can have more than three assistant deans, the last design is infeasible. This last design also feels wrong on an aesthetic basis. There are three columns that represent the same attribute, and usually only one column is used for an attribute.

The point of this example is to show that sometimes a design team must consider criteria other than normalization.

Normalization Can Cause Poor Performance

Consider a mail order company that uses a database with the following table, which is in domain/key normal form:

ITEM (ItemNumber, Name, Color, Description, Picture, QuantityOnHand, QuantityOnOrder, Price)

Assume this table is used to prepare the monthly catalog and to support order processing. To support catalog production, Description is a long memo field that explains the features and benefits of the product. It may be 1000 bytes or longer. Even more problematic, Picture is a jpg-style image that can be as large as 256K bytes. For purposes of creating the catalog, such large attributes are not a problem because the table is processed just once in sequence.

The order processing application, however, accesses the ITEM table thousands of times and in random order, and fast performance is very important. Assume also that the order processing application does not need Description and Price. Unfortunately, depending on the characteristics of the DBMS in use, the presence of the two large attributes Description and Picture may significantly slow retrieval and update performance.

If this is the case, the designers might decide to create a second table that contains a duplicate copy of data needed by the order processing application. They may create, for example, a table like the following with duplicate data used only by order processing:

ORDERITEM (ItemNumber, Name, Color, QuantityOnHand, QuantityOnOrder, Price)

Now, both of these tables are normalized, so you might say this design is not really a normalization problem. But the purpose of normalization is to reduce data duplication

to prevent anomalies and integrity problems. By duplicating the data, the designers are forcing multiple updates to record a single fact. This is akin to an insertion anomaly, even if the insertion occurs on two different tables.

Such a duplicate table design creates a potential for serious data integrity problems. The designers will need to develop both automated and manual controls to ensure that both copies of the data are updated in a consistent manner. Before implementing this design, the developers need to ensure that the increased performance will be worth the cost of the controls and the risk of integrity problems.

Another reason for duplicating data is to support query and reporting applications. You will see examples of such duplication by using what is called a **star schema** when we discuss OLAP processing in Chapter 15.

The relational model is the industry standard for database processing today. It was first published by E. F. Codd in 1970. At first, it was deemed too theoretical, but it was used for high-volume transaction processing in organizations by the 1980s, thanks to DBMS products such as DB2 and Oracle.

A relation is a two-dimensional table that has the characteristics listed in Figure 4-1. In this book and in the database world in general, the term *table* is used synonymously with relation. Three sets of terminology are used for relational structures. *Table*, *row*, and *column* are most commonly used, but *file* (or *datafile*), *record*, and *field* are sometimes used in traditional data processing. Theorists also use *relation*, *tuple*, and *attribute* for the same three constructs. Sometimes, these terms are mixed and matched. Strictly speaking, a relation may not have duplicate rows; sometimes this condition is relaxed, however, because eliminating duplicates can be a time-consuming process.

A key is one or more columns of a relation that is used to identify a row. A unique key identifies a single row; a non-unique key identifies several rows. A composite key is a key having two or more attributes. A relation has one primary key, which must be a unique key. A relation may also have additional unique keys, which are called candidate keys. A primary key is used to represent the table in relationships, and many DBMS products use values of the primary key to organize table storage. Also, an index is normally constructed to give fast access by primary key values.

A functional dependency occurs when the value of one attribute (or set of attributes) determines the value of a second attribute (or set of attributes). The attribute on the left side of the functional dependency is called the determinant. One way to view the purpose of a relation is to say that the relation exists to store instances of functional dependencies. Another way to define a primary (and candidate key) is to say that such a key is an attribute that functionally determines all of the other attributes in a relation.

When updated, some relations suffer from undesirable consequences called modification anomalies. A deletion anomaly occurs when the deletion of a row loses information about two or more entities. An insertion anomaly occurs when the relational structure forces the addition of facts about two entities at the same time. Anomalies can be removed by splitting the relation into two or more relations.

There are many types of modification anomalies. Relations can be classified by the types of anomaly that they eliminate. Such classifications are called normal forms.

By definition, every relation is in first normal form. A relation is in second normal form if all non-key attributes are dependent on all of the key. A relation is in third normal form if it is in second normal form and has no transitive dependencies. A relation is in Boyce–Codd normal form if every determinant is a candidate key. A relation is in fourth normal form if it is in Boyce–Codd normal form and has no multi-value dependencies. The definition of fifth normal form is intuitively obscure, so we did not define it.

A relation is in domain/key normal form if every constraint on the relation is a logical consequence of the definition of domains and keys. A constraint is any constraint on

the static values of attributes whose truth can be evaluated. A domain is a named set of values that an attribute can have.

In addition to normalization, which is a process of analyzing relations, it is also possible to construct well-formed relations by synthesis, according to the relationship between attribute values. If two attributes functionally determine each other, they have a one-to-one relationship. If one attribute functionally determines the other, but not the reverse, the attributes have a many-to-one relationship. If neither attribute determines the other, they have a many-to-many relationship. These facts can be used when constructing relations, as summarized in Figure 4-22.

In some cases, normalization is not desirable. Whenever a table is split into two or more tables, extra processing is required when the tables are later rejoined. Also, referential integrity constraints need to be enforced. If the cost of the extra processing of the two tables and their integrity constraint is greater than the benefit of avoiding modification anomalies, normalization is not recommended. In some cases, creating repeating columns is preferred to the standard normalization techniques; and in other cases, controlled redundancy is used to improve performance.

ESTIO

4.1 What restrictions must be placed on a table for it to be considered a relation?

4.2 Define the following terms: *relation, tuple, attribute, file, record, field, table, row, column.*

4.3 Define *functional dependency.* Give an example of two attributes that have a functional dependency and give an example of two attributes that do not have a functional dependency.

4.4 If SID functionally determines Activity, does this mean that only one value of SID can exist in the relation? Why or why not?

4.5 Define *determinant.*

4.6 Give an example of a relation having a functional dependency in which the determinant has two or more attributes.

4.7 Define *key.*

4.8 If SID is a key of a relation, is it a determinant? Can a given value of SID occur more than once in the relation?

4.9 What is a deletion anomaly? Give an example other than one in this text.

4.10 What is an insertion anomaly? Give an example other than one in this text.

4.11 Explain the relationship of first, second, third, Boyce–Codd, fourth, fifth, and domain/key normal forms.

4.12 Define *second normal form.* Give an example of a relation in 1NF, but not in 2NF. Transform the relation into relations in 2NF.

4.13 Define *third normal form.* Give an example of a relation in 2NF, but not in 3NF. Transform the relation into relations in 3NF.

4.14 Define *BCNF.* Give an example of a relation in 3NF, but not in BCNF. Transform the relation into relations in BCNF.

4.15 Define *multi-value dependency.* Give an example.

4.16 Why are multi-value dependencies not a problem in relations with only two attributes?

4.17 Define *fourth normal form.* Give an example of a relation in BCNF, but not in 4NF. Transform the relation into relations in 4NF.

4.18 Define *domain/key normal form.* Why is it important?

4.19 Transform the following relation into DK/NF. Make and state the appropriate assumptions about functional dependencies and domains.

EQUIPMENT (Manufacturer, Model, AcquisitionDate, BuyerName, BuyerPhone, PlantLocation, City, State, ZIP)

4.20 Transform the following relation into DK/NF. Make and state the appropriate assumptions about functional dependencies and domains.

INVOICE (Number, CustomerName, CustomerNumber, CustomerAddress, ItemNumber, ItemPrice, ItemQuantity, SalespersonNumber, SalespersonName, Subtotal, Tax, TotalDue)

4.21 Answer question 4.20 again, but this time add attribute CustomerTaxStatus (0 if nonexempt, 1 if exempt). Also, add the constraint that there will be no tax if CustomerTaxStatus = 1.

4.22 Give an example, other than one in this text, in which you would judge normalization to be not worthwhile. Show the relations and justify your design.

4.23 Explain two situations in which database designers might intentionally choose to create data duplication. What is the risk of such designs?

[illegible] TEST

4.24 Consider the following relation definition and sample data:

PROJECT Relation

ProjectID	EmployeeName	EmployeeSalary
100A	Jones	64K
100A	Smith	51K
100B	Smith	51K
200A	Jones	64K
200B	Jones	64K
200C	Parks	28K
200C	Smith	51K
200D	Parks	28K

PROJECT (ProjectID, EmployeeName, EmployeeSalary)

Where ProjectID is the name of a work project

EmployeeName is the name of an employee who works on that project

EmployeeSalary is the salary of the employee whose name is EmployeeName

Assuming that all of the functional dependencies and constraints are apparent in this data, which of the following statements is true?

A. ProjectID → EmployeeName

B. ProjectID → EmployeeSalary

C. (ProjectID, EmployeeName) → EmployeeSalary

D. EmployeeName → EmployeeSalary

E. EmployeeSalary → ProjectID

F. EmployeeSalary → (ProjectID, EmployeeName)

Answer these questions:

G. What is the key of PROJECT?

H. Are all non-key attributes (if any) dependent on all of the key?

I. In what normal form is PROJECT?

J. Describe two modification anomalies from which PROJECT suffers.

K. Is ProjectID a determinant?

L. Is EmployeeName a determinant?

M. Is (ProjectID, EmployeeName) a determinant?

N. Is EmployeeSalary a determinant?

O. Does this relation contain a partial-key problem? If so, what is it?

P. Redesign this relation to eliminate the modification anomalies.

4.25 Consider the following relation definition and sample data:

PROJECT-HOURS Relation

EmployeeName	ProjectID	TaskID	Phone	TotalHours
Don	100A	B-1	12345	12
Don	100A	P-1	12345	12
Don	200B	B-1	12345	12
Don	200B	P-1	12345	12
Pam	100A	C-1	67890	26
Pam	200A	C-1	67890	26
Pam	200D	C-1	67890	26

PROJECT-HOURS (EmployeeName, ProjectID, TaskID, Phone, TotalHours)

Where EmployeeName is the name of an employee

ProjectID is the name of a project

TaskID is the name standard work task

Phone is the employee's telephone number

TotalHours is the hours worked by the employee on this project

Assuming that all of the functional dependencies and constraints are apparent in this data, which of the following statements is true?

A. EmployeeName → ProjectID

B. EmployeeName →→ ProjectID

C. EmployeeName → TaskID

D. EmployeeName →→ TaskID

E. EmployeeName → Phone

F. EmployeeName → TotalHours

G. (EmployeeName, ProjectID) → TotalHours

H. (EmployeeName, Phone) → TaskID

I. ProjectID → TaskID

J. TaskID → ProjectID

Answer these questions:

K. What are all of the determinants?

L. Does this relation contain a partial-key problem? If so, what is it?

M. Does this relation contain a multi-value dependency? If so, what are the unrelated attributes?

N. What is the deletion anomaly that this relation contains?

O. How many themes does this relation have?

P. Redesign this relation to eliminate the modification anomalies. How many relations did you use? How many themes does each of your new relations contain?

4.26 Consider the following domain, relation, and key definitions:

Domain Definitions

EmployeeName	in	Names values CHAR(20)
PhoneNumber	in	Phones values DEC(5)
EquipmentName	in	ENames values CHAR(10)
Location	in	Places values CHAR(7)
Cost	in	Money values CURRENCY
Date	in	Dates values YYMMDD
Time	in	Times values HHMM where HH between 00 and 23 and MM between 00 and 59

Definitions of Relation, Key, and Constraint

EMPLOYEE (EmployeeName, PhoneNumber)
- Key: EmployeeName
- Constraints: EmployeeName → PhoneNumber

EQUIPMENT (EquipmentName, Location, Cost)
- Key: EquipmentName
- Constraints: EquipmentName → Location
  EquipmentName → Cost

APPOINTMENT (Date, Time, EquipmentName, EmployeeName)
- Key: (Date, Time, EquipmentName)
- Constraints: (Date, Time, EquipmentName) → EmployeeName

A. Modify the definitions to enforce this constraint: An employee may not sign up for more than one equipment appointment.

B. Define nighttime to refer to the hours between 2100 and 0500. Add an attribute Employee Type whose value is 1 if the employee works during nighttime. Change this design to enforce the constraint that only employees who work at night can schedule nighttime appointments.

FIREDUP PROJECT QUESTIONS

FiredUp hired a team of database designers (who should have been fired!) to create the following relations for a database to keep track of their stove, repair, and customer data. See the projects at the end of Chapters 1 through 3 to review their needs. For each of the following relations, specify candidate keys, functional dependencies, and multi-valued dependencies (if any). Justify these specifications unless they are obvious. Given your specifications about keys and so on, what normal form does each relation have? Transform each relation into two or more relations that are in domain/key normal form. Indicate the primary key of each table, candidate keys, foreign keys; and specify any referential integrity constraints.

In answering these questions, assume the following:

- Stove type and version determine tank capacity.
- A stove can be repaired many times, but never more than once on a given day.
- Each stove repair has its own repair invoice.
- A stove can be registered to different users, but never at the same time.
- A stove has many component parts and each component part can be used on many stoves. Thus, FiredUp maintains records about part types, such as *burner valve,* and not about particular parts such as burner valve number 41734 manufactured on 12 December 2003.

A. PRODUCT1 (SerialNumber, Type, VersionNumber, TankCapacity, DateOfManufacture, InspectorInitials)

B. PRODUCT2 (SerialNumber, Type, TankCapacity, RepairDate, RepairInvoiceNumber, RepairCost)

C. REPAIR1 (RepairInvoiceNumber, RepairDate, RepairCost, RepairEmployeeName, RepairEmployeePhone)

D. REPAIR2 (RepairInvoiceNumber, RepairDate, RepairCost, RepairEmployeeName, RepairEmployeePhone, SerialNumber, Type, TankCapacity)

E. REPAIR3 (RepairDate, RepairCost, SerialNumber, DateOfManufacture)

F. STOVE1 (SerialNumber, RepairInvoiceNumber, ComponentPartNumber)

G. STOVE2 (SerialNumber, RepairInvoiceNumber, RegisteredOwnerID)

Assume there is a need to record the owner of a stove, even if it has never been repaired.

H. Given the assumptions of this case, the relations and attributes in items A–G and your knowledge of small business, construct a set of domain/key relations for FiredUp. Indicate primary keys, foreign keys, and referential integrity constraints.