CHAPTER 7

Normalization

Abstract

This chapter covers the process of transforming an ER diagram into a well-designed relational database. For each normal norm, the chapter presents the rules that relations must meet to reach that normal form, and the design problems that may remain. It also covers the process for transforming relations into higher normal forms, including relational algebra operations, where appropriate.

Keywords

normalization
normal forms
database normalization
first normal form
second normal form
third normal form
fourth normal form
fifth normal form
sixth normal form
domain-key normal form
Boyce–Codd normal form
relational database design

Given any pool of entities and attributes, there is a large number of ways you can group them into relations. In this chapter, you will be introduced to the process of *normalization*, through which you create relations that avoid most of the problems that arise from bad relational design.

There are at least two ways to approach normalization. The first is to work from an ER diagram. If the diagram is drawn correctly, then there are some simple rules you can use to translate it into relations that will avoid most relational design problems. The drawback to this approach is that it can be difficult to determine whether your design is correct. The second approach is to use the theoretical concepts behind good design to create your relations. This is a bit more difficult than working from an ER diagram, but often results in a better design.

Normalization theory relies heavily on the relational algebra operation project. When moving a relation from one design stage to another, we *decompose* a relation by taking projections to create more than one relation. Normalization also occasionally uses the join operation.

In practice, you may find it useful to use a combination of both the ER diagram and theoretical approaches. First, create an ER diagram, and use it to design your relations. Then, check those relations against the theoretical rules for good design, and make any changes necessary to meet the rules.

Translating an ER Diagram into Relations

An ER diagram in which all many-to-many relationships have been transformed into one-to-many relationships, through the introduction of composite entities, can be translated directly into a set of relations. To do so:

- Create one table for each entity.
- For each entity that is only at the “one” end of one or more relationships, and not at the “many” end of any relationship, create a single-column primary key, using an arbitrary unique identifier if no natural primary key is available.
- For each entity that is at the “many” end of one or more relationships, include the primary key of each parent entity (those at the “one” end of the relationships) in the table as foreign keys.
- If an entity at the “many” end of one or more relationships has a natural primary key (for example, an order number or an invoice number), use that single column as the primary key. Otherwise, concatenate the primary key of its parent with any other column or columns needed for uniqueness to form the table’s primary key.

Following these guidelines, we end up with the following tables for the *Antique Opticals* database:

```
customer (customer_numb, customer_first_name,
     customer_last_name, customer_street, customer_city,
     customer_state, customer_zip, customer_phone)

distributor (distributor_numb, distributor_name,
```

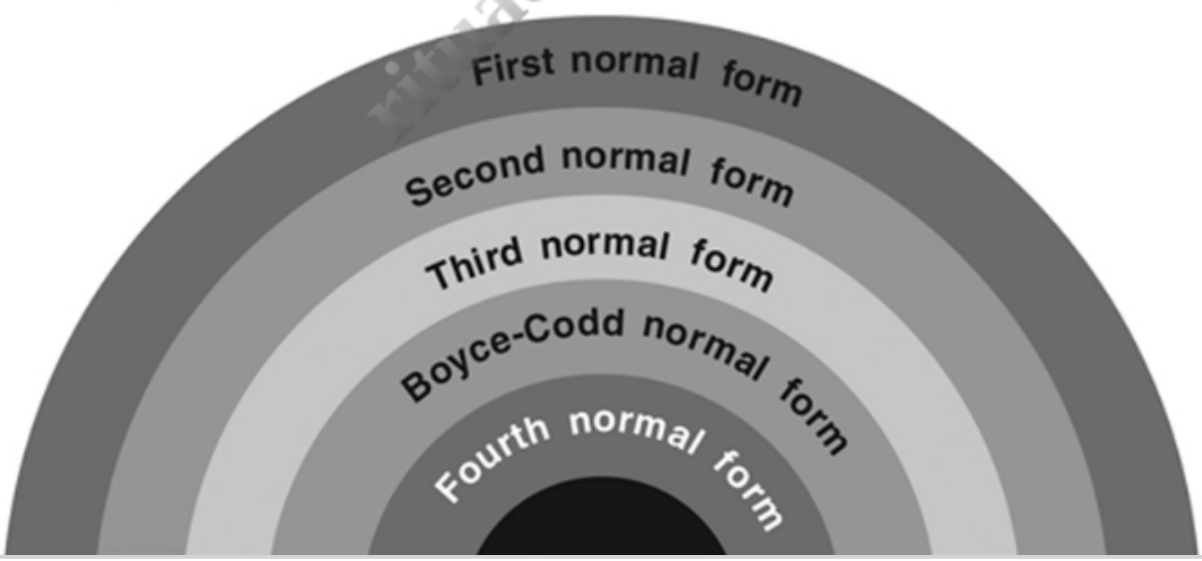
```
distributor (distributor_numb, distrbutor_name,
     distrbutor_street, distrbutor_city, distrbutor_state,
     distrbutor_zip, distrbutor_phone,
     distrbutor_contact_person, contact_person_ext)

item (item_numb, item_type, title, distributor_numb,
     retail_price, release_date, genre, quant_in_stock)

order (order_numb, customer_numb, order_date, credit_card_numb,
     credit_card_exp_date, order_complete?, pickup_or_ship?)

order item (order_numb, item_numb, quantity, discount_percent,
     selling_price, line_cost, shipped?, shipping_date)

purchase (purchase_date, customer_numb, items_received?,
     customer_paid?)

purchase item (purchase_date, customer_numb, item_numb,
     condition, price_paid)

actor (actor_numb, actor_name)

performance (actor_numb, item_numb, role)

producer (producer_name, studio)

production (producer_name,item_numb)
```

Note: You will see these relations reworked a bit throughout the remainder of the first part of this book to help illustrate various aspects of database design. However, the preceding is the design that results from a direct translation of the ER diagram.

Normal Forms

The theoretical rules that the design of a relation must meet are known as *normal forms*. Each normal form represents an increasingly stringent set of rules. Theoretically, the higher the normal form, the better the design of the relation.

As you can see in Figure 7.1, there are six nested normal forms, indicating that if a relation is in one of the higher, outer normal forms, it is also in all of the normal forms inside it.



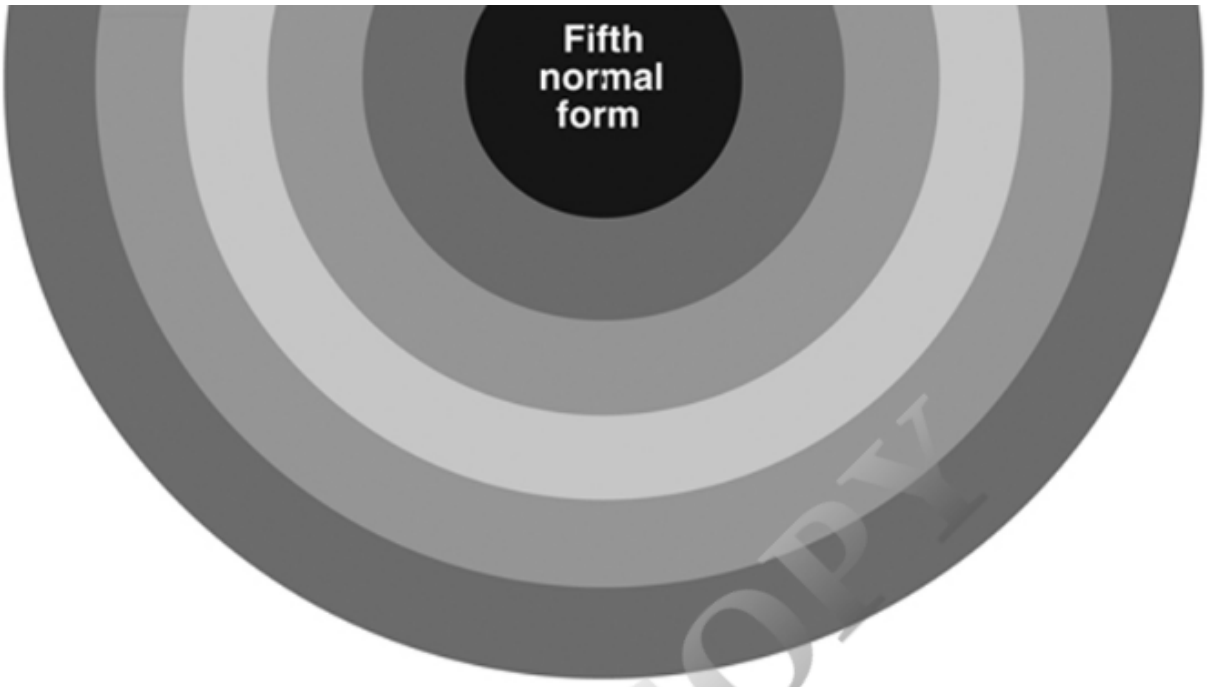


FIGURE 7.1 **Nested normal forms.**

In most cases, if you can place your relations in third normal form (3NF), then you will have avoided most of the problems common to bad relational designs. The three higher normal forms—Boyce–Codd, fourth normal form (4NF), and fifth normal form (5NF)—handle special situations that arise only occasionally. However, the situations that these normal forms handle are conceptually easy to understand, and can be used in practice, if the need arises.

In recent years, sixth normal form has been added to relational database design theory. It is not precisely a more rigorous normal form than fifth normal form, although it uses the same principles to transform relations from one form to another. You will be introduced to it briefly at the end of this chapter.

Note: In addition to the six normal forms in Figure 7.1 and sixth normal form, there is another normal form—domain/key normal form—that is of purely theoretical importance and to this date, has not been used as a practical design objective.

First Normal Form

A table is in first normal form (1NF) if it meets the following criteria:

- The data are stored in a two-dimensional table.
- There are no repeating groups.

The key to understanding 1NF therefore is understanding the nature of a repeating group of data.

Understanding Repeating Groups

A *repeating group* is an attribute that has more than one value in each row of a table. For example, assume that you were working with an employee relation, and needed to store the names and birthdates of the employees' children. Because each employee can have more than one child, the names of children and the children's birthdates each form a repeating group.

Note: A repeating group is directly analogous to a multivalued attribute in an ER diagram.

There is actually a very good reason why repeating groups are disallowed. To see what might happen if they were present, take a look at Figure 7.2, an instance of an *employee* table containing repeating groups.

emp#	first	last	children's names	children's birthdates
1001	Jane	Doe	Mary, Sam	1/9/02, 5/15/04
1002	John	Doe	Lisa, David	1/9/00, 5/15/01
1003	Jane	Smith	John, Pat, Lee, Mary	10/5/04, 10/12/00, 6/6/2006, 8/21/04
1004	John	Smith	Michael	7/4/06
1005	Jane	Jones	Edward, Martha	10/21/05, 10/15/99

FIGURE 7.2 **A table with repeating groups.**

Note: The table in Figure 7.2 is not a legal relation, because it contains those repeating groups. Therefore, we should not call it a relation.
Notice that there are multiple values in a single row in both the children's names and children's birthdates columns. This presents two major

Notice that there are multiple values in a single row in both the children's names and children's birthdates columns. This presents two major problems:

- There is no way to know exactly which birthdate belongs to which child. It is tempting to say that we can associate the birthdates with the children by their positions in the list, but there is nothing to ensure that the relative positions will always be maintained.
- Searching the table is very difficult. If, for example, we want to know which employees have children born before 2005, the DBMS will need to perform data manipulation to extract the individual dates themselves. Given that there is no way to know how many birthdates there are in the column for any specific row, the processing overhead for searching becomes even greater.

The solution to these problems, of course, is to get rid of the repeating groups altogether.

Handling Repeating Groups

There are two ways to get rid of repeating groups to bring a table into conformance with the rules for first normal form—a right way and a wrong way. We will look first at the wrong way so you will know what *not* to do.

In Figure 7.3 you can see a relation that handles repeating groups by creating multiple columns for the multiple values. This particular example includes three pairs of columns for a child's name and birthdate.

emp#	first	last	child name 1	child bdate 1	child name 2	child bdate 2	child name 3	child bdate 3
1001	Jane	Doe	Mary	1/1/02	Sam	5/15/04		
1002	John	Doe	Lisa	1/1/00	David	5/15/01		
1003	Jane	Smith	John	10/5/04	Pat	10/12/00	Lee	6/6/06
1004	John	Smith	Michael	7/4/06				
1005	Joe	Jones	Edward	10/21/05	Martha	10/15/99		

FIGURE 7.3 **A relation handling repeating groups in the wrong way.**

The relation is Figure 7.3 does meet the criteria for first normal form. The repeating groups are gone and there is no problem identifying which birthdate belongs to which child. However, the design has introduced several problems of its own:

- The relation is limited to three children for any given employee. This means that there is no room to store Jane Smith's fourth child. Should you put another row for Jane Smith into the table? If so, then the primary key of this relation can no longer be just the employee number. The primary key must include one child's name as well.
- The relation wastes space for people who have less than three children. Given that disk space is one of the least expensive elements of a database system, this is probably the least of the problems with this relation.
- Searching for a specific child becomes very clumsy. To answer the question "Does anyone have a child named Lee?" the DBMS must construct a query that includes a search of all three child name columns because there is no way to know in which column the name might be found.

The right way to handle repeating groups is to create another table (another entity) to handle multiple instances of the repeating group. In the example we have been using, we would create a second table for the children, producing something like Figure 7.4.

employee

emp#	first	last
1001	Jane	Doe
1002	John	Doe
1003	Jane	Smith
1004	John	Smith
1005	Joe	Jones

children

children

emp#	c_first	c_birthdate
1001	Mary	1/1/02
1001	Sam	5/15/04
1002	Lisa	1/1/00
1002	David	5/15/01
1003	John	10/5/04
1003	Pat	10/12/00
1003	Lee	6/6/06
1003	Mary	8/21/04
1004	Michael	7/4/06
1005	Edward	10/21/05
1005	Martha	1015/99

FIGURE 7.4 **The correct way to handle a repeating group.**

Neither of the two new tables contains any repeating groups and this form of the design avoids all the problems of the preceding solution:

- There is no limit to the number of children that can be stored for a given employee. To add another child, you simply add another row to the *children* table.
- There is no wasted space. The children table uses space only for data that are present.
- Searching for a specific child is much easier because children's names are found in only one column.

Problems with First Normal Form

Although first normal form relations have no repeating groups, they are full of other problems. To see what these problems are, we will look at the table underlying the data entry form in Chapter 3. (This table comes from *Antique Opticals'* original data management system, rather than the new and improved design you saw earlier in this chapter.) Expressed in the notation for relations that we have been using, the relation is:

```
orders (customer_numb, first_name, last_name, street,
    city, state, zip, phone, order_numb, order_date,
    item_numb, title, price, has_shipped?)
```

The first thing we need to do is determine the primary key for this table. The customer number alone will not be sufficient, because the customer number repeats for every item ordered by the customer. The item number will also not suffice, because it is repeated for every order on which it appears. We cannot use the order number, because it is repeated for every item on the order. The only solution is a concatenated key, in this example, the combination of the order number and the item number.

Given that the primary key is made up of the order number and the item number, there are two important things we cannot do with this relation:

- We cannot add data about a customer until the customer places at least one order because, without an order and an item on that order, we do not have a complete primary key.
- We cannot add data about a merchandise item we are carrying without that item being ordered. There must be an order number to complete the primary key.

The preceding are *insertion anomalies*, a situation that arises when you are prevented from inserting data into a relation because a complete primary key is not available. (Remember that *no* part of a primary key can be null.)

primary key is not available. (Remember that *no* part of a primary key can be null.)

Note: To be strictly correct, there is a third insertion anomaly in the orders relation. You cannot insert an order until you know one item on the order. In a practical sense, however, no one would enter an order without there being an item ordered.

Insertion anomalies are common in first normal form relations that are not also in any of the higher normal forms. In practical terms, they occur because there are data about more than one entity in the relation. The anomaly forces you to insert data about an unrelated entity (for example, a merchandise item) when you want to insert data about another entity (such as a customer).

First normal form relations can also give us problems when we delete data. Consider, for example, what happens if a customer cancels the order of a single item and the business rules state that cancelled items and orders are to be deleted:

- In cases where the deleted item was the only item on the order, you lose all data about the order.
- In cases where the order was the only order on which the item appeared, you lose data about the item.
- In cases where the deleted item was the item ordered by a customer you lose all data about the customer.

These *deletion anomalies* occur because part of the primary key of a row becomes null when the merchandise item data are deleted, forcing you to remove the entire row. The result of a deletion anomaly is the loss of data that you would like to keep. In practical terms, you are forced to remove data about an unrelated entity when you delete data about another entity in the same table.

Note: Moral to the story: More than one entity in a table is a bad thing.

One of the most important things to keep in mind about insertion and deletion anomalies is that they affect entire entities. If you want to insert customer data but can't because the customer has not placed an order, for example, you can't insert *any* customer data. The anomaly does not just prevent you from entering a primary key value, but also anything else you want to store about an instance of the customer entity.

There is a final type of anomaly in the *orders* relation that is not related to the primary key: a *modification*, or *update*, *anomaly*. The *orders* relation has a great deal of unnecessary duplicated data, in particular, information about customers. When a customer moves, then the customer's data must be changed in every row for every item on every order ever placed by the customer. If every row is not changed, then data that should be the same are no longer the same. The potential for these inconsistent data is the modification anomaly.

Second Normal Form

The solution to anomalies in a first normal form relation is to break the relation down so that there is one relation for each entity in the 1NF relation. The *orders* relation, for example, will break down into four relations (*customers*, *items*, *orders*, and *line items*). Such relations are in at least second normal form (2NF).

In theoretical terms, second formal form relations are defined as follows:

- The relation is in first normal form.
- All nonkey attributes are functionally dependent on the entire primary key.

The new term in the preceding is *functionally dependent*, a special relationship between attributes.

Understanding Functional Dependencies

A functional dependency is a one-way relationship between two attributes, such that at any given time, for each unique value of attribute A, only one value of attribute B is associated with it throughout the relation. For example, assume that A is the customer number from the *orders* relation. Each customer number is associated with one customer first name, one last name, one street address, one city, one state, one zip code, and one phone number. Although the values for those attributes may change at any moment, there is only one.

We therefore can say that first name, last name, street, city, state, zip, and phone are functionally dependent upon the customer number. This relationship is often written:

```
customer_numb ->> first_name, last_name, street, city, state,
    zip, phone
```

and read "customer number determines first name, last name, street, city, state, zip, and phone." In this relationship, customer number is known as the *determinant* (an attribute that determines the value of other attributes).

Notice that the functional dependency does not necessarily hold in the reverse direction. For example, any given first or last name may be associated with more than one customer number. (It would be unusual to have a customer table of any size without some duplication of names.)

The functional dependencies in the orders table are:

```
customer_numb ->> first_name, last_name, street, city, state,
     zip, phone

item_numb ->> title, price

order_numb ->> customer_numb, order_date

item_numb + order_numb ->> has_shipped?
```

Notice that there is one determinant for each entity in the relation and the determinant is what we have chosen as the entity identifier. Notice also that, when an entity has a concatenated identifier, the determinant is also concatenated. In this example, whether an item has shipped depends on the combination of the item and the order.

on the combination of the item and the order.

Using Functional Dependencies to Reach 2NF

If you have correctly identified the functional dependencies among the attributes in a database environment, then you can use them to create second normal form relations. Each determinant becomes the primary key of a relation. All the attributes that are functionally dependent upon it become non-key attributes in the relation.

The four relations into which the original *orders* relation should be broken are:

```
customer (customer_numb, first_name, last_name, street, city,
     state, zip, phone)

item (item_numb, title, price)

order (order_numb, customer_numb, order_date)

order_items (order_numb, item_numb, has_shipped?)
```

Each of these should, in turn, correspond to a single entity in your ER diagram.

Note: When it comes to deciding what is driving database design—functional dependencies or entities—it is really a "chicken and egg" situation. What is most important is that there is consistency between the ER diagram and the functional dependencies you identify in your relations. It makes no difference whether you design by looking for functional dependencies or for entities. In most cases, database design is an iterative process in which you create an initial design, check it, modify it, and check it again. You can look at either functional dependencies and/or entities at any stage in the process, checking one against the other for consistency.

The relations we have created from the original orders relation have eliminated the anomalies present in the original:

- It is now possible to insert data about a customer before the customer places an order.
- It is now possible to insert data about an order before we know an item on the order.

- It is now possible to store data about merchandise items before they are ordered.
- Line items can be deleted from an order without affecting data describing that item, the order itself, or the merchandise item.
- Data describing the customer are stored only once and therefore any change to those data needs to be made only once. A modification anomaly cannot occur.

Problems with 2NF Relations

Although second normal form eliminates problems from many relations, you will occasionally run into relations that are in second normal form, yet still exhibit anomalies. Assume, for example, that each new DVD title that *Antique Opticals* carries comes from one distributor, and that each distributor has only one warehouse, which has only one phone number. The following relation is therefore in second normal form:

```
item (item_numb, title, distrib_numb, warehouse_phone_number)
```

For each item number, there is only one value for the item's title, distributor, and warehouse phone number. However, there is one insertion anomaly—you cannot insert data about a distributor until you have an item from that distributor—and a deletion anomaly—if you delete the only item from a distributor, you lose data about the distributor. There is also a modification anomaly: the distributor's warehouse phone number is duplicated for every item the company gets from that distributor. The relation is in second normal form, but not third.

Third Normal Form

Third normal form is designed to handle situations like the one you just read about in the preceding section. In terms of entities, the item relation does contain two entities: the merchandise item and the distributor. That alone should convince you that the relation needs to be broken down into two smaller relations, both of which are now in third normal form:

```
item (item_numb, distrib_numb)

distributor (distrib_numb, warehouse_phone_number)
```

The theoretical definition of third normal form says:

- The relation is in second normal form.
- There are no transitive dependencies.

The functional dependencies found in the original relation are an example of a *transitive dependency*.

Transitive Dependencies

A transitive dependency exists when you have the following functional dependency pattern:

```
A ->> B and B ->> C therefore A ->> C
```

This is precisely the case with the original items relation. The only reason that the warehouse phone number is functionally dependent on the item number is because the distributor is functionally dependent on the item number, and the phone number is functionally dependent on the distributor. The functional dependencies are really:

```
item_numb ->> distrib_numb
distrib_numb ->> warehouse_phone_number
```

Note: Transitive dependencies take their name from the transitive property in mathematics, which states that if a > b and b > c, then a > c.

There are two determinants in the original items relation, each of which should be the primary key of its own relation. However, it is not merely the presence of the second determinant that creates the transitive dependency. What really matters is that the second determinant is not a candidate key (could be used as a primary key) for the relation.

Consider, for example, this relation:

```
Item (item_numb, upc, distrib_numb, price)
```

item (item_numb, upc, distrib_numb, price)

The item number is an arbitrary number that *Antique Opticals* assigns to each merchandise item. The UPC is an industry-wide code that is unique to each item as well. The functional dependencies in this relation are:

```
item_numb ->> upc, distrib_numb, price

upc ->> item_numb, distrib_numb, price
```

Is there a transitive dependency here? No, because the second determinant is a candidate key. (*Antique Opticals* could just as easily have used the UPC as the primary key.) There are no insertion, deletion, or modification anomalies in this relation; it describes only one entity—the merchandise item.

A transitive dependency therefore exists only when the determinant that is not the primary key is not a candidate key for the relation. For example, in the *items* table we have been using as an example, the distributor is a determinant, but not a candidate key for the table. (There can be more than one item coming from a single distributor.)

When you have a transitive dependency in a 2NF relation, you should break the relation into two smaller relations, each of which has one of the determinants in the transitive dependency as its primary key. The attributes determined by the determinant become nonkey attributes in each relation. This removes the transitive dependency—and its associated anomalies—and places the relation in third normal form.

Note: A second normal form relation that has no transitive dependencies is, of course, automatically in third normal form.

Boyce–Codd Normal Form

For most relations, third normal form is a good design objective. Relations in that state are free of most anomalies. However, occasionally you run into relations that exhibit special characteristics where anomalies still occur. Boyce–Codd normal form (BCNF), fourth normal form (4NF), and fifth normal form (5NF) were created to handle such special situations.

Note: If your relations are in third normal form and do not exhibit the special characteristics that BCNF, 4NF, and 5NF were designed to handle, then they are automatically in 5NF.

The easiest way to understand BCNF is to start with an example. Assume that *Antique Opticals* decides to add a relation to its database to handle employee work scheduling. Each employee works one or two 4-hour shifts a day at the store. During each shift, an employee is assigned to one station (a place in the store, such as the front desk or the stockroom). Only one employee works a station during the given shift.

A relation to handle the schedule might be designed as follows:

```
schedule (employee_id, date, shift, station, worked_shift?)
```

Given the rules for the scheduling (one person per station per shift), there are two possible primary keys for this relation: *employee_ID* + *date* + *shift* or *date* + *shift* + *station*. The functional dependencies in the relation are:

```
employee_id + date + shift ->> station, worked_shift?

date + shift + station ->> employee_id, worked_shift?
```

Keep in mind that this holds true only because there is only one person working each station during each shift.

Note: There is very little difference between the two candidate keys as far as the choice of a primary key is concerned. In cases like this, you can choose either one.

This schedule relation exhibits *overlapping candidate keys*. (Both candidate keys have date and shift in common.) BCNF was designed to deal with relations that exhibit this characteristic.

To be in BCNF, a relation must meet the following rules:

- The relation is in third normal form.
- All determinants are candidate keys.

BCNF is considered to be a more general way of looking at 3NF because it includes those relations with the overlapping candidate keys. The sample *schedule* relation we have been considering does meet the criteria for BCNF because the two determinants are indeed candidate keys.

Fourth Normal Form

Like BCNF, fourth normal form was designed to handle relations that exhibit a special characteristic that does not arise too often. In this case, the special characteristic is something known as a *multivalued dependency*.

As an example consider the following relation:

movie info (title, star, producer)

A given movie can have more than one star; it can also have more than one producer. The same star can appear in more than one movie; a producer can also work on more than one movie (for example, see the instance in Figure 7.5). The relation must therefore include all columns in its key.

title	star	producer
Great Film	Lovely Lady	Money Bags
Great Film	Handsome Man	Money Bags
Great Film	Lovely Lady	Helen Pursestrings
Great Film	Handsome Man	Helen Pursestrings
Boring Movie	Lovely Lady	Helen Pursestrings
Boring Movie	Precocious Child	Helen Pursestrings

FIGURE 7.5 **A relation with a multivalued dependency.**

Because there are no nonkey attributes, this relation is in BCNF. Nonetheless, the relation exhibits anomalies:

- You cannot insert the stars of a movie without knowing at least one producer.
- You cannot insert the producer of a movie without knowing at least one star.
- If you delete the only producer from a movie, you lose information about the stars.
- If you delete the only star from a movie, you lose information about its producers.
- Each producer's name is duplicated for every star in the movie. By the same token, each star's name is duplicated for each producer of the movie. This unnecessary duplicated data forms the basis of a modification anomaly.

There are at least two unrelated entities in this relation: one that handles the relationship between a movie and its stars and another that handles the relationship between a movie and its producers. In a practical sense, that is the cause of the anomalies. (Arguably, there are also movie, star, and producer entities involved.)

However, in a theoretical sense, the anomalies are caused by the presence of a multivalued dependency in the same relation, which must be eliminated to get to fourth normal form. The rules for fourth normal form are:

- The relation is in BCNF.
- There are no multivalued dependencies.

Multivalued Dependencies

A multivalued dependency exists when for each value of attribute A, there exists a finite set of values of attribute B that are associated with it, and a finite set of values of attribute C that are also associated with it. Attributes B and C are independent of each other.

In the example we have been using, there is just such a dependency. First, for each movie title, there is a group of actors (the stars) who are associated with the movie. For each title, there is also a group of producers who are associated with it. However, the actors and the producers are independent of one another. As you can see in the ER diagram in Figure 7.6, the producers and stars have no direct relationship (despite the relationships being M:M).

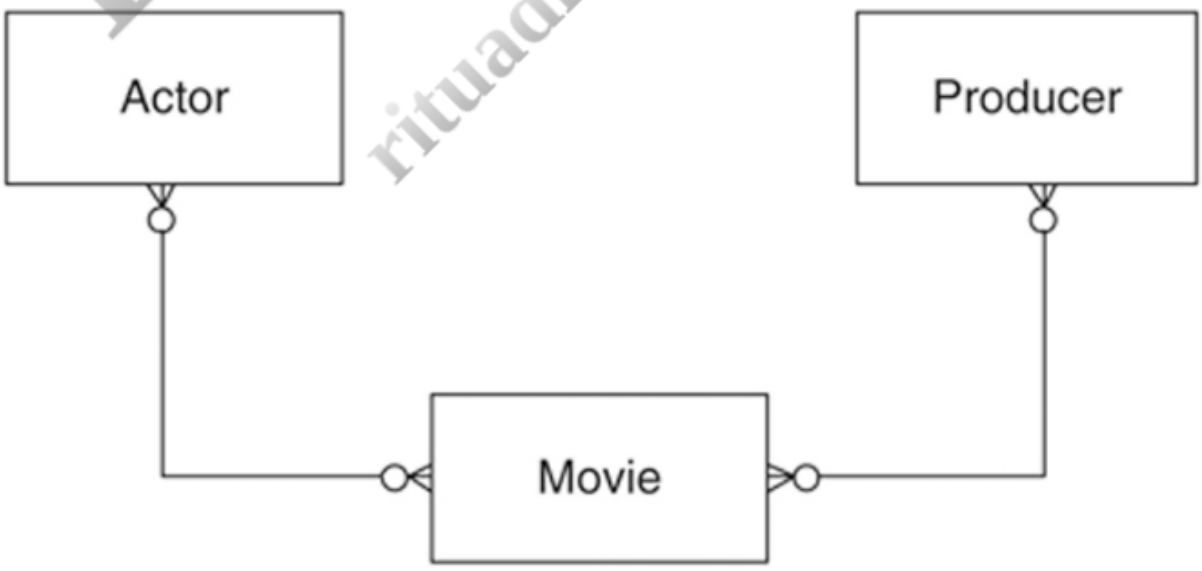


FIGURE 7.6 **An ER diagram of the multivalued dependency.**

FIGURE 7.6 An ER diagram of the multivalued dependency.

Note: At this point, do not let semantics get in the way of database theory. Yes, it is true that producers fund the movies in which the actors are starring, but in terms of database relationships, there is no direct connection between the two.

The multivalued dependency can be written:

```
title ->> star

title ->> producer
```

and read "title multidetermines star and title multidetermines producer."

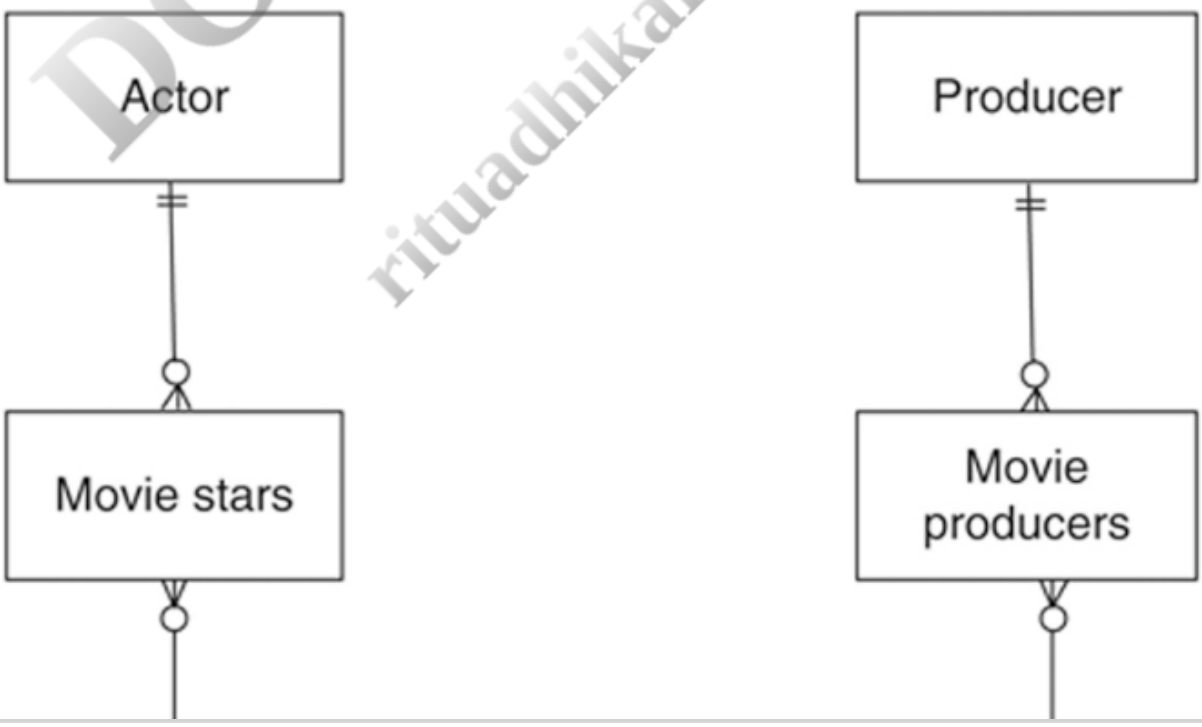
Note: To be strictly accurate, a functional dependency is a special case of a multivalued dependency where what is being determined is one value rather than a group of values.

To eliminate the multivalued dependency and bring this relation into fourth normal form, you split the relation placing each part of the dependency in its own relation:

```
movie_stars (title, star)

movie_producers (title, producer)
```

With this design, you can independently insert and remove stars and producers without affecting the other. Star and producer names also appear only once for each movie with which they are involved.

Another way to look at this is to notice that *movie_stars* and *movie_producers* are actually composite entities. When we add them to the ER diagram, they resolve the M:M relationships (Figure 7.7).



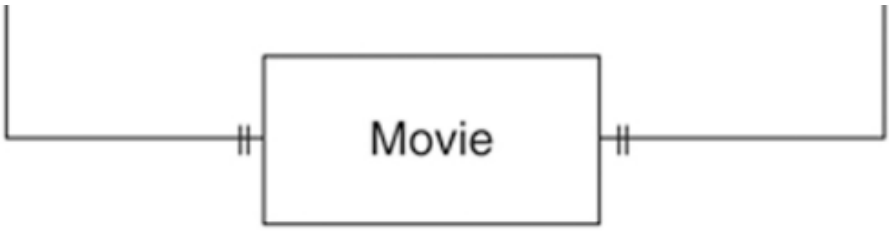


FIGURE 7.7 **Creating fourth normal form with composite entities.**

Fifth Normal Form

Fifth normal form—also known as projection-join normal form—is designed to handle a general case of a multivalued dependency known as a *join dependency*.

Consider the following relation that is in fourth normal form, but not fifth:

```
selections (customer_numb, series, item_numb)
```

This relation represents various series of discs, such as Star Trek or Rambo. Customers place orders for a series; when a customer orders a series, he or she must take all items in that series. Determining fifth normal form becomes relevant only when this type of rule is in place. If customers could request selected titles from a series, then the relation would be fine. Because it would be all-key and would have no multivalued dependencies, it would automatically fall through the normal form rules to 5NF.

To make the problems with this table under the preceding rule clearer, consider the instance of the relation in Figure 7.8.

customer number	series	item number
1005	Star Wars	2090
1005	Star Wars	2091
1005	Star Wars	2092
1005	Star Wars	4689
1005	Star Wars	4690
1005	Star Wars	4691
1010	Harry Potter	3200
1010	Harry Potter	3201
1010	Harry Potter	3202
1010	Harry Potter	3203

1010	Harry Potter	3204
2180	Star Wars	2090
2180	Star Wars	2091
2180	Star Wars	2092
2180	Star Wars	4689
2180	Star Wars	4690
2180	Star Wars	4691

FIGURE 7.8 **A relation in 4NF but not 5NF.**

Because this table has no multivalued dependencies, it is automatically in fourth normal form. However, there is a great deal of unnecessary duplicated data in this relation—the item numbers are repeated for every customer that orders a given series. The series name is also repeated for every item in the series and for every customer ordering that series. This relation will therefore be prone to modification anomalies.

There is also a more subtle issue: under the rules of this relation, if customer 2180 orders the first Harry Potter movie and indicates that he or she would like more movies in the series, then the only way to put that choice in the table is to add rows for all five Harry Potter movies. You may be forced to add rows that you don't want to add and introduce data that are not accurate.

Note: There is no official term for the preceding anomaly. It is precisely the opposite of the insertion anomalies described earlier in this chapter, although it does involve a problem with inserting data.

By the same token, if a customer doesn't want one item in a series, then you must remove all the rows for that customer for that series from the table. If the customer still wants the remaining items in the series, then you have a deletion anomaly.

As you might guess, you can solve the problem by breaking the table into two smaller tables, eliminating the unnecessary duplicated data, and the insertion and deletion anomalies:

```
series_subscription (customer_numb, series)

series_content (series, item_numb)
```

The official definition for 5NF is as follows:

- The relation is in fourth normal form.
- All join dependencies are implied by the candidate keys.

A *join dependency* occurs when a table can be put together correctly by joining two or more tables, all of which contain only attributes from the original table. The original *selections* relation does have a join dependency, because it can be created by joining the *series subscription* and *series content relations*. The join is valid only because of the rule that requires a customer to order all items in a series.

A join dependency is implied by candidate keys when all possible projections from the original relation that form a join dependency each contain a candidate key for the original relation. For example, the following projections can be made from the *selections* relation:

```
A: (customer_numb, series)

B: (customer_numb, item_numb)

C: (series, item_numb)
```

C. (series, item_numb)

We can regenerate the *selections* relation by combining any two of the preceding relations. Therefore, the join dependencies are A + B, A + C, B + C, and A + B + C. Like other relational algebra operations, the join theoretically removes duplicate rows, so although the raw result of the join contains extra rows, they will be removed from the result, producing the original table.

Note: One of the problems with 5NF is that as the number of columns in a table increases, the number of possible projections increases exponentially. It can therefore be very difficult to determine 5NF for a large relation.

However, each of the projections does not contain a candidate key for the *selections* relation. All three columns from the original relation are required for a candidate key. Therefore, the relation is not in 5NF. When we break down the selections relation into *series_selections* and *series_content*, we eliminate the join dependencies, ensuring that the relations are in 5NF.

Sixth Normal Form

Normalization theory has been very stable for more than 45 years. However, in the late 1990s, C. J. Date, one of the foremost experts in database theory, proposed a sixth normal form, particularly to handle situations in which there is temporal data. This is technically not a project-join normal form, as were all of those discussed earlier in this chapter.

Consider the following relation:

```
customer (id, valid_interval, street, city, state, zip, phone)
```

The intent of this relation is to maintain a history of a customer's locations and when they were valid (starting date to ending date). Depending on the circumstances, there may be a great deal of duplicated data in this relation (for example, if only the phone number changed) or very little (for example, if there is a move to a new state with a new phone number). Nonetheless, there is only one functional dependency in the relation:

```
id + valid_interval ->> street, city, state, zip, phone
```

There are no transitive dependencies, no overlapping candidate keys, no multivalued dependencies, and all join dependencies are implied by the candidate key(s). The relation is therefore in fifth normal form.

Sixth normal form was created to handle the situation where temporal data vary independently to avoid unnecessary duplication. The result is tables that cannot be decomposed any further; in most cases, the tables include the primary key and a single non-key attribute. The sixth normal form tables for the sample *customer* relation would be as follows:

```
street_addresses (id, valid_interval, street)

cities (id, valid_interval, city)

states (id, valid_interval, state)

zip_codes (id, valid_interval, zip)

phone_numbers (id, valid_interval, phone)
```

The resulting tables eliminate the possibility of redundant data, but introduce some time consuming joins to find a customer's current address or to assemble a history for a customer. For this reason alone, you may decide that it makes sense to leave the relation in 5NF and not decompose it further.

Going to sixth normal form may also introduce the need for a *circular inclusion constraint*. There is little point in including a street address for a customer unless a city, state, and zip code exist for the same date interval. The circular inclusion constraint would therefore require that if a row for any given interval and any given customer ID exists in any of *street_addresses*, *cities*, *states*, or *zip_codes*, matching rows must exist in all of those tables. Today's relational DBMSs do not support circular inclusion constraints nor are they included in the current SQL standard. If such a constraint is necessary, it will need to be enforced through application code.

For Further Reading

There are many books available that deal with the theory of relational databases. You can find useful supplementary information in the

Earp R. *Database Design Using Entity-Relationship Diagrams*. Taylor & Francis; 2007.
Halpin T, Morgan T. *Information Modeling and Relational Databases*. third ed. Morgan Kaufman; 2008.
Hillyer, M. An Introduction to Database Normalization. http://dev.mysql.com/tech-resources/articles/intro-to-normalization.html
Olivé A. *Conceptual Modeling of Information Systems*. Springer; 2007.
Pratt PJ, Adamski JJ. *Concepts of Database Management*. sixth ed. Course Technology; 2007.
Ritchie C. *Database Principles and Design*. third ed. Cengage Learning Business Press; 2008.