CHAPTER 7

1. Objectives of Normalization

The basic objective of logical modeling is to develop a "good" description of the data, its relationships, and its constraints. For the relational model, this means that we must identify a suitable set of relations. However, the task of choosing the relations is a difficult one, because there are many options for the designer to consider. This chapter is devoted to explaining some methods of improving logical design. The techniques presented here are based on a large body of research into the logical design process generally called normalization.

The purpose of normalization is to produce a stable set of relations that is a faithful model of the operations of the enterprise. By following the principles of normalization we can achieve a design that is highly flexible, allowing the model to be extended when needed to account for new attributes, entity sets, and relationships. We can also reduce redundancy in the database and ensure that the design is free of certain update, insertion, and deletion anomalies. An anomaly is an inconsistent, incomplete, or contradictory state of the database. If these anomalies were present, we would be unable to represent some information, we might lose information when certain updates were performed, and we would run the risk of having data become inconsistent over time.

Research into these anomalies was first done by Codd, who identified the causes and defined the first three "normal forms". A relation is in a specific normal form if it satisfies the set of requirements or constraints for that form. Note that the constraints we discuss are schema constraints, permanent properties of the relation, not merely of some instance of the relation. Later research by Boyce and Codd led to a refinement of the third of these forms. Additional research by Fagin and independently by Zaniolo resulted in the definition of three new normal forms. All of the normal forms are nested, in that each satisfies the constraints of the previous one but is a "better" form because each eliminates flaws found in the previous form. Figure 7.1 shows how the seven normal forms are related. The largest circle represents all relations. Among the set of all relations those that are in first normal form are represented by the next-largest circle. Of those in first normal form, there are some that qualify as second normal form as well; these fall into the next circle, and so on. Our design objective should be to put the schema in the highest normal form that is practical and appropriate for the application. "Normalization" means putting a relation into a higher normal form. Although there is still a great deal of research being done on this process, the last normal form, domain-key normal form, is the "final" one in a sense which we will discuss later. Normalization requires that we have a clear grasp of the semantics of the application. Merely examining an instance or extension is not sufficient, because an instance does not provide enough information about all the possible values or combinations of values for relation attributes.

In attempting to pinpoint the causes of update, insertion, and deletion anomalies, researchers have identified three important kinds of dependencies: functional dependencies, multivalued dependencies and join dependencies. Additional dependencies appear in the research literature.

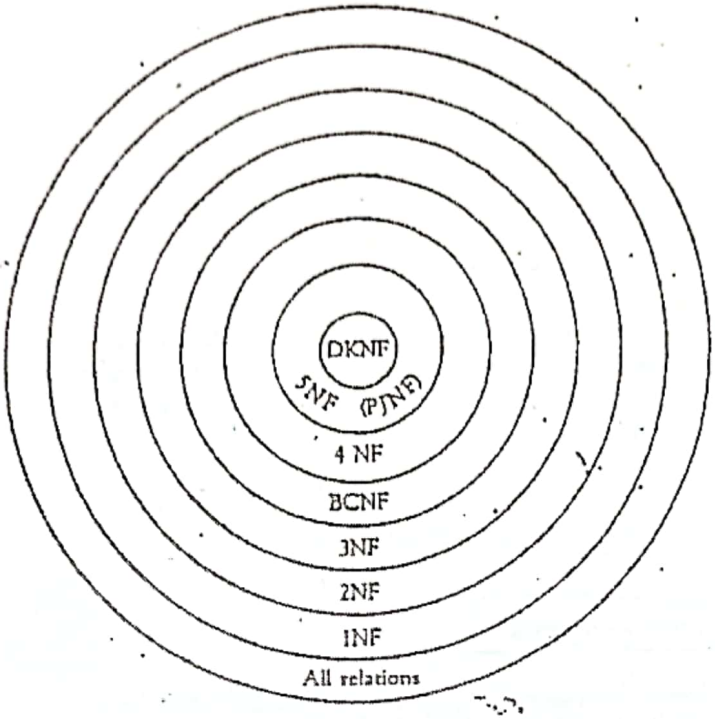


Figure 7.1

Functional Dependency

A functional dependency (FD) is a type of relationship between attributes as described in the following definition:

Definition of Functional Dependency

DEFINITION: If A and B are attributes or sets of attributes of relation R, we say that B is functionally dependent on A if each value of A in R has associated with it exactly one value of B in R.

$A \rightarrow B$

We write this as $A \rightarrow B$, read as "A functionally determines B" or "A determines B". (Note: If we want to be very explicit about which relation we are discussing, particularly in the case where A or B appears in another relation, we write the qualified names, as in $R.A \rightarrow R.B$.) This does not mean that A causes B or that the value of B can be calculated from the value of A by a formula, although sometimes that is the case. It simply means that if we know the value of A and we examine the table of relation R, we will find only one value of B in all the rows that have the given value of A at any one time. Thus when two rows have the same A value, they must also have the same B value. However, for a given B value, there may be several different A values. When a functional dependency exists, the attribute or set of attributes on the left side of the arrow is called a determinant.

To illustrate functional dependency, let us consider the following relation:

```
STUDENT(STUID, STUNAME, MAJOR, CREDITS, STATUS, SOCSECNO)
```

This relation stores information about students in a college. Here, we will assume that every student has a unique ID and social security number and that each student has at most one major. We will assume that names are not unique, so that two different students may have the same name. CREDITS means the number of credits completed, and STATUS refers to the year the student is in—freshman, sophomore, junior, or senior.

Figure 7.2 shows an instance of the table. Although we will use this instance to help us detect functional dependencies, we are trying to determine permanent characteristics of the relation, not simply of the instance. Examining the table, we see that if we are given a specific value of STUID, there is only one value of STUNAME associated with that particular STUID. For example, for STUID S1006, the associated STUNAME is Lee,Pamela. We know from our assumptions that this is a characteristic of the relation, not just of this instance, so STUNAME is functionally dependent on STUID, and we can write

```
STUID → STUNAME
```

However, for a given value of STUNAME there may be more than one STUID. We note that for the STUNAME Jones,Mary there are two STUID values, S1003 and S1060. Therefore we cannot turn the functional dependency around and write STUNAME → STUID. For each of the other attributes, there is only one value associated with a particular value of STUID, so all the attributes are functionally dependent on STUID. We write

```
STUID → STUNAME, MAJOR, CREDITS, STATUS, SOCSECNO
```

to indicate that each of the attributes on the right is functionally dependent on STUID. Similarly, we have

```
SOCSECNO → STUID, STUNAME, MAJOR, CREDITS, STATUS
```

Also note that STATUS is functionally dependent on CREDITS. For example, if the value of CREDITS is known to be 15, the STATUS is automatically freshman. We write

```
CREDITS → STATUS
```

However, we cannot write STATUS → CREDITS, since two freshmen may have a different number of credits, as we see in the records of S1006 and S1060. For this case of functional dependency, the determinant CREDITS does not functionally determine all the other attributes of the relation. Note also that the value of CREDITS is not necessarily unique. Several students could have the same number of credits, so uniqueness is not a necessary characteristic of determinants. The instance in Figure 7.2 did not really

STUID	STUNAME	MAJOR	CREDITS	STATUS	SOCSECNO
S1001	Smith,Tom	History	90	Sen	100429500
S1003	Jones,Mary	Math	95	Sen	010124567
S1006	Lee,Pamela	CSC	15	Fresh	088520876
S1010	Burns,Edward	Art	63	Jun	099320985
S1060	Jones,Mary	CSC	25	Fresh	054624738

Figure 7.2 STUDENT Table (Each student has at most one major)

demonstrate the lack of uniqueness of CREDITS, which shows that we must be careful in making judgments from instances only. We need to concentrate on the meanings of the attributes and their constraints in identifying functional dependencies.

Superkeys, Candidate Keys, and Primary Keys

Recall from Chapter 5 that a superkey is an attribute or a set of attributes that identifies an entity uniquely. In a table, a superkey is any column or set of columns whose values can be used to distinguish one row from another. Since a superkey identifies each entity uniquely, it functionally determines all the attributes of a relation. For the STUDENT table in Figure 7.2, STUID is a superkey. So is the combination of {STUID,STUNAME}. In fact, {STUID,any other attribute} is a superkey for this relation. The same is true for {SOCSECNO,any other attribute}.

A candidate key is a superkey such that no proper subset of its attributes is itself a superkey. Therefore, a candidate key must be a minimal identifier. In our example, the superkey {STUID,STUNAME} is not a candidate key because it contains a proper subset, STUID, that is a superkey. However, STUID by itself is a candidate key. Similarly, SOCSECNO is a candidate key. If no two students were permitted to have the same combination of name and major values, the combination {STUNAME,MAJOR} would also be a candidate key. A relation may have several candidate keys. We will use the term "composite key" to refer to a key that consists of more than one attribute. We will also drop the set brackets in showing sets of attributes.

A primary key is a candidate key that is used to identify tuples in a relation. In our example, STUID might be the primary key and SOCSECNO an alternate key. These two attributes functionally determine each other, and all of the attributes are functionally dependent on each of them. Although not explicitly stated in the definition, an important characteristic of a primary key is that none of its attributes may have null values. If we permitted null values in keys, we would be unable to tell records apart, since two records with null values in the same key field might be indistinguishable. It is also desirable to enforce the "no nulls" rule for candidate keys.

Inference Rules

Rules of inference for functional dependencies, called inference axioms or Armstrong's axioms, after their developer, can be used to find all the FDs logically implied by a set of FDs. These rules are sound, meaning that they are an immediate consequence of the definition of functional dependency and that any functional dependency that can be derived from a given set of FDs using them is true. They are also complete, meaning they can be used to derive every valid inference about dependencies, so that if a particular FD cannot be derived from a given set of FDs, the given set of FDs does not imply that particular FD.

Let A, B, C, and D be subsets of attributes of a relation R. The following axioms hold: (Note: AC means A and C.)

F1. Reflexivity

If B is a subset of A, then $A \rightarrow B$. This also implies that $A \rightarrow A$ always holds. Functional dependencies of this type are called trivial functional dependencies.

F2. Augmentation

If $A \rightarrow B$, then $AC \rightarrow BC$

F3. Transitivity

If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

The following rules can be derived from the previous three:

F4. Additivity or Union

If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.

F5. Projectivity or Decomposition

If $A \rightarrow BC$ then $A \rightarrow B$ and $A \rightarrow C$.

F6. Pseudotransitivity

If $A \rightarrow B$ and $CB \rightarrow D$, then $AC \rightarrow D$.

These rules can be used to develop a formal theory of functional dependencies, but we will concentrate instead on their practical applications.

(If F is a set of functional dependencies for a relation R, the set of all functional dependencies that can be derived from F, $F+$, is called the closure of F.) Armstrong's axioms are sufficient to compute all of $F+$; that is, if we were to apply these rules repeatedly, we would find all the functional dependencies in $F+$. However, the task would obviously be very complex and take a lot of time.

Given a set of functional dependencies F of a relation R, we are often interested in finding all the attributes in R that are functionally dependent on a certain attribute or

set of attributes, A, in R. We call this set of attributes the closure of A or $A+$.) Clearly, if $A+$ is all of R, then A is a superkey for R. We could find $A+$ by computing all of $F+$ and then choosing only those functional dependencies where A is the determinant, but there is a shortcut. The following algorithm can be used to find $A+$, given a set F of functional dependencies:

```
result := A;
while (result changes) do
     for each functional dependency B → C in F
     if B is contained in result then result := result ∪ C;
end;
A+ := result;
```

For example, let R be a relation with attributes W, X, Y, Z and functional dependencies

$$W \rightarrow Z$$

$$YZ \rightarrow X$$

$$WZ \rightarrow Y$$

Let us compute $WZ+$. We assign WZ to the result and enter the while for the first time. We look for an FD whose determinant is contained in result, which has only WZ at the moment. The FD $WZ \rightarrow Y$ satisfies this requirement, so we assign $WZ \cup Y$ to result. Since we had a change in result, we enter the while again. Now we look for an FD where W, Z, Y, or any combination of these three is a determinant. We can use $YZ \rightarrow X$, and now we assign $WZY \cup X$ to result. Since we have found that every attribute of R is in $WZ+$, WZ is a superkey.

To find $W+$, we assign W to result and enter the while for the first time. Using the FD $W \rightarrow Z$, we assign $W \cup Z$ to result. Since we had a change to result, we enter the while again. This time we use $WZ \rightarrow Y$ and we assign $WZ \cup Y$ to result. Now we look for an FD where any combination of WZY appears as the determinant. This time we can use $YZ \rightarrow X$, and we assign $WZY \cup X$ to result. Since we have found that every attribute of R is in $W+$, W is a superkey for this relation. Because it has no proper subset that is also a superkey, W is a candidate key as well. Note that we now know that WZ is not a candidate key.

It is easy to verify that YZ is not a superkey. We start with result := YZ. Using $YZ \rightarrow X$, result becomes YZX. Now we look for an FD where some combination of YZX is the determinant. Since there is none, we cannot add any new attributes to result. This means that $YZ+$ is only YZX, so W is not functionally dependent on YZ, which means that YZ is not a superkey.

Given a set of functional dependencies, we would like to be able to determine whether any of them is redundant, meaning that it can be derived from the others. To do so, we can use the following algorithm:

```
1. Choose a candidate FD, say X → Y, and remove it from the set of FDs.
2. result := X;
   while (result changes and Y is not contained in result) do
    for each FD, A → B, remaining in the reduced set of FDs,
```

```
        if A is a subset of result, then result := result∪B .
    end
3.  If Y is a subset of result, then the FD X → Y is redundant
```

We could then remove the FD $X \rightarrow Y$ from the set, since it can be derived from the other FDs. By testing every FD in the set and removing any that can be derived from the others, we can find a nonredundant set of FDs equivalent to the original set.

For example, suppose that we have the following set of FDs:

(1) $W \rightarrow Z$

(2) $W \rightarrow Y$

(3) $YZ \rightarrow X$

(4) $WZ \rightarrow Y$

We begin by testing (1), $W \rightarrow Z$. We assign W to result. Now we search for an FD [other than (1)] in which W is the determinant. We find one in (2), $W \rightarrow Y$. Now we assign $W \cup Y$ to result. Searching for an FD having a determinant that is contained in WY, we find none. Therefore, we are unable to show that Z is contained in result, and we conclude that (1) is not redundant.

Next we test (2), $W \rightarrow Y$. We assign W to result. Searching for an FD other than (2) whose determinant is W, we find one in (1), so we can assign WZ to result. Now we seek an FD whose determinant is contained in WZ. We find (4), $WZ \rightarrow Y$, and now we can assign WZY to result. Since Y is now contained in result, we can exit the while and conclude that (2) is redundant. We now eliminate (2) from the set of FDs.

Testing (3), we assign YZ to result. We look for an FD other than (3) or (2) whose determinant is contained in YZ. Finding none, we conclude that (3) is not redundant.

Testing (4), we assign WZ to result. In (1), we see that the determinant, W, is contained in result, so we could add the right side of (1), Z, to result, but it is already there. There is no other FD whose determinant is contained in WZ, except for (2), which we have eliminated. Therefore, we conclude that (4) is not redundant in the reduced set of FDs since we are unable to get Y in result.

Sometimes we would like to be able to substitute one set of FDs for another. If F and G are two sets of FDs, then F is a cover or G if every FD in G is also in $F+$, which means that every FD in G can be derived from the FDs in F. In the special case where F is a cover for G and G is also a cover for F (i.e., $F+ = G+$), then F and G are said to be equivalent. A cover, F, for G, is said to be nonredundant if F is a cover for G but no proper subset of F is a cover for G.

G may have several nonredundant covers, so we would like to be able to find one that has the fewest number of FDs in it and is in a standard, or canonical, form in which each FD has a single attribute on the right side.

A set of FDs, F, is said to be minimal if

(1) F has no redundant FDs
(2) The right side of every FD in F has a single attribute
(3) No attribute on the left side of any FD in F is redundant.

This means that if $X \rightarrow Y$ is an FD in F, there is no proper subset S of X such that $S \rightarrow Y$ can be used in place of $X \rightarrow Y$ and the resulting set is equivalent to F. If F is minimal and equivalent to G, then F is a minimal cover for G. Intuitively, F is a minimal cover if the number of FDs in it is less than or equal to the number of FDs in any canonical set equivalent to it. A set of FDs may have several minimal covers.

7.5 First Normal Form

We will use a counterexample to describe first normal form. If we assume that a student is permitted to have more than one major, and we try to store multiple majors in the same field of the student record, our STUDENT table might appear as in Figure 7.3(a). This example violates the definition of first normal form, which is:

Definition of First Normal Form (1NF)

DEFINITION: A relation is in first normal form if and only if every attribute is single-valued for each tuple.

This means that each attribute in each row, or each "cell" of the table, contains only one value. No repeating fields or groups are allowed. An alternative way of describing first normal form is to say that the domains of the attributes of a relation are atomic, that is, they consist of single units that cannot be broken down further. In our table in Figure 7.3(a), we see this rule violated in the records of students S1006 and S1010, who now have two values listed for MAJOR. This is the first such example we have seen, because all the relations we considered until now were in first normal form. In fact, much of relational theory is based on relations in first normal form, although there is current research on non-first-normal-form relations. It is important to have first normal form so that the relational operators, as we have defined them, will work correctly. For example, if we perform the relational algebra operation SELECT STUDENT WHERE MAJOR ='Math' on the table in Figure 7.3(a), should the record of student S1006 be included? If we were to do a natural join with some other table using MAJOR as the joining column, would this record be paired with only those having both CSC and Math as MAJOR, or with those having either CSC or Math? What would happen on the join if another record had a double major listed as Math CSC? To avoid these ambiguities, we will insist that every relation be written in first normal form. If a relation is not already in 1NF, we can rewrite it, "flattening" it so that each of the multiple values appears in a different row. Therefore, we would rewrite the STUDENT table as in Figure 7.3(b). We also note that the key is no longer STUID, since any student with more than one major appears at least twice. Now we need STUID,MAJOR (or SOCSECNO,MAJOR) to identify a record uniquely.

STUID	STUNAME	MAJOR	CREDITS	STATUS	SOCSECNO
S1001	Smith,Tom	History	90	Sen	100429500
S1003	Jones,Mary	Math	95	Sen	010124567
S1006	Lee,Pamela	CSC	15	Fresh	088520876
		Math			
S1010	Burns,Edward	Art	63	Jun	099320985
		English			
S1060	Jones,Mary	CSC	25	Fresh	064624738

Figure 7.3(a) STUDENT Table (Students may have more than one major)

STUID	STUNAME	MAJOR	CREDITS	STATUS	SOCSECNO
S1001	Smith,Tom	History	90	Sen	100429500
S1003	Jones,Mary	Math	95	Sen	010124567
S1006	Lee,Pamela	CSC	15	Fresh	088520876
S1006	Lee,Pamela	Math	15	Fresh	088520876
S1010	Burns,Edward	Art	63	Jun	099320985
S1010	Burns,Edward	English	63	Jun	099320985
S1060	Jones,Mary	CSC	25	Fresh	064624738

Figure 7.3(b) STUDENT Table Rewritten in INF

Figure 7.3

Full Functional Dependency

Consider the relation

CLASS (COURSE#, STUID, STUNAME, FACID, SCHED, ROOM, GRADE)

In this example we will assume that COURSE# includes the department and section, and that there is only one faculty member for each class (i.e., no team teaching). The SCHED is a string that gives meeting days and time, and ROOM gives the building and room where the class always meets. An instance of this relation appears in Figure 7.4(a).

COURSE#	STUID	STUNAME	FACID	SCHED	ROOM	GRADE
ART103A	S1001	Smith,Tom	F101	MWF9	H221	A
ART103A	S1010	Burns,Edward	F101	MWF9	H221	
ART103A	S1006	Lee,Pamela	F101	MWF9	H221	B
CSC201A	S1003	Jones,Mary	F105	TUTHF10	M110	A
CSC201A	S1006	Lee,Pamela	F105	TUTHF10	M110	C
HST205A	S1001	Smith,Tom	F202	MWF11	H221	

Figure 7.4(a) CLASS Table

COURSE#	STUID	GRADE
ART103A	S1001	A
ART103A	S1010	
ART103A	S1006	B
CSC201A	S1003	A
CSC201A	S1006	C
HST205A	S1001	

CLASS2 Table

STUID	STUNAME
S1001	Smith,Tom
S1010	Burns,Edward
S1006	Lee,Pamela
S1003	Jones,Mary

STU Table

COURSE#	FACID	SCHED	ROOM
ART103A	F101	MWF9	H221
CSC201A	F105	TUTHF10	M110
HST205A	F202	MWF11	H221

COURSE Table

Figure 7.4(b) CLASS2, STU, and COURSE Tables

Figure 7.4

We have the following functional dependency

COURSE#, STUID → STUNAME, FACID, SCHED, ROOM, GRADE

Since there is no other candidate key, COURSE#,STUID is the (composite) key for the relation. Ignoring trivial functional dependencies, we also have the functional dependencies

COURSE# → FACID, SCHED, ROOM
STUID → STUNAME

In both of these cases, we find attributes that are functionally dependent on the combination COURSE#,STUID, but also functionally dependent on a subset of that combination. We say that such attributes are not fully functionally dependent on the combination.

Definition of Full Functional Dependence

DEFINITION: In a relation R, attribute B of R is fully functionally dependent on an attribute or set of attributes A of R if B is functionally dependent on A but not functionally dependent on any proper subset of A.

In our example, although STUNAME is functionally dependent on COURSE#,STUID, it is also functionally dependent on a proper subset of that combination, STUID. Similarly, FACID, SCHED, and ROOM are functionally dependent on the proper subset COURSE#. We note that GRADE is fully functionally dependent on the combination COURSE#,STUID.

7.2 Second Normal Form

Definition of Second Normal Form (2NF)

DEFINITION: A relation is in second normal form (2NF) if and only if it is in first normal form and all the nonkey attributes are fully functionally dependent on the key.

Clearly, if a relation is 1NF and the key consists of a single attribute, the relation is automatically 2NF. The only time we have to be concerned about 2NF is when the key is composite. From the definition, we see that the CLASS relation is not in second normal form. For example, STUNAME is not fully functionally dependent on the key COURSE#,STUID. Although there are other nonfull functional dependencies here, one is sufficient to show that the relation is not 2NF.

A relation that is not 2NF exhibits the update, insertion, and deletion anomalies that we referred to in Section 7.1. Suppose that we wish to change the schedule of ART103A to MWF12. It is possible we might update the first two records of the CLASS table but not the third, resulting in an inconsistent state in the database. It would then be impossible to tell the true schedule for that class. This is an update anomaly. An insertion anomaly occurs when we try to add information about a course for which no student has yet registered. Because the key is COURSE#,STUID, we are not permitted to insert a null value for STUID, so we are unable to record the course information, even though we may have the COURSE#, FACID, SCHED, and ROOM values available. Because we are not able to represent this information, we have an insertion anomaly. A similar problem occurs if we try to insert information about a student who has not yet registered for any course. A deletion anomaly occurs when we delete the record of the

only student taking a particular course. For example, if student S1001 drops HST205A, we lose all information about that course. It is desirable to keep the course information, but we cannot do so without a corresponding STUID. Similarly, if a student drops the only course that he or she is taking, we lose all information about that student.

A 1NF relation that is not 2NF can be transformed into an equivalent set of 2NF relations. We perform projections on the original relation in such a way that it is possible to get back the original by taking the join of the projections. Projections of this type are called lossless projections and are discussed in more detail in Section 7.14. Essentially, to make a relation 2NF, you identify each nonfull functional dependency and form projections by removing the attributes that depend on each of the determinants so identified. These determinants are placed in separate relations along with their dependent attributes. The original relation still contains the composite key and any attributes that are fully functionally dependent on it. Even if there are no attributes fully functionally dependent on the key of the original relation, it is important to keep the relation (with only the key attributes) in order to be able to reconstruct the original relation by a join. This "connecting relation" shows how the projections are related.

Applying this method to our example

```
CLASS (COURSE#, STUID, STUNAME, FACID, SCHED, ROOM, GRADE)
```

we first identify all the functional dependencies that we are concerned about. They are

```
COURSE# ⟶ FACID, SCHED, ROOM
STUID   ⟶ STUNAME
COURSE#, STUID ⟶ GRADE ( and, of course, FACID, SCHED, ROOM, STUNAME)
```

Using projection, we break up the CLASS relation into the following set of relations:

```
CLASS2 (COURSE#, STUID, GRADE)
COURSE (COURSE#, FACID, SCHED, ROOM)
STU (STUID, STUNAME)
```

The resulting relations are shown in Figure 7.4(b). Note that we could reconstruct the original relation by taking the natural join of these three relations. Even if there were no GRADE attribute, we would need the relation CLASS2(COURSE#,STUID) in order to show which students are enrolled in which courses. Without it, we could not join COURSE and STU. Using these new relations, we have eliminated the update, insertion, and deletion anomalies discussed earlier. We can change the course schedule for ART103A by updating the SCHED column in a single record in COURSE. We can add new course information to COURSE without having any student registered in the course, and we can insert new student information by adding a record to STU, without having to insert course information for that student. Similarly, we can drop a registration record from CLASS2 and still retain the information about the course in COURSE and about the student in STU.

7.3 Transitive Dependency

Although second-normal-form relations are better than those in first normal form, they may still have update, insertion, and deletion anomalies. Consider the following relation:

STUDENT (STUID, STUNAME, MAJOR, CREDITS, STATUS)

Figure 7.5(a) shows an instance of this relation. Here, the only candidate key is STUID, so we will use that as the primary key. Every other attribute of the relation is functionally dependent on the key, so we have the following functional dependency, among others:

STUID → CREDITS

However, since the number of credits determines STATUS, as discussed in Section 7.2, we also have

CREDITS → STATUS

Thus STUID functionally determines STATUS in two ways—directly, and transitively, through STATUS:

(STUID → CREDITS) AND (CREDITS → STATUS) ⟹ (STUID → STATUS)

Recall that transitivity is the third of Armstrong's axioms. Thus a transitive dependency is one that "carries over" another attribute. Put simply, a transitive dependency occurs when one nonkey attribute determines another nonkey attribute. For third normal form, we concentrate on relations with only one candidate key, and we eliminate transitive dependencies.

7.4 Third Normal Form

Transitive dependencies cause insertion, deletion, and update anomalies. For example, in the STUDENT table in Figure 7.5, we cannot insert the information that any student with 30 credits has Soph status until we have such a student, because that would require inserting a record without a STUID, which is not permitted. If we delete the record for the only student with a certain number of credits, we lose the information about the status associated with those credits. If we have several records with the same CREDITS value and we change the status associated with that value (e.g., making 24 credits now have status of Soph), we may accidentally fail to update all of the records, leaving the database in an inconsistent state. Because of these problems, it is desirable to remove

STUID	STUNAME	MAJOR	CREDITS	STATUS
S1001	Smith,Tom	History	90	Sen
S1003	Jones,Mary	Math	95	Sen
S1006	Lee,Pamela	CSC	15	Fresh
S1010	Burns,Edward	Art	63	Jun
S1060	Jones,Mary	CSC	25	Fresh

Figure 7.5(a) STUDENT Table

STUID	STUNAME	MAJOR	CREDITS
S1001	Smith,Tom	History	90
S1003	Jones,Mary	Math	95
S1006	Lee,Pamela	CSC	15
S1010	Burns,Edward	Art	63
S1060	Jones,Mary	CSC	25

STU2 Table

CREDITS	STATUS
15	Fresh
25	Fresh
63	Jun
90	Sen
95	Sen

STATS Table

Figure 7.5(b) STU2 and STATS Tables

Figure 7.5

transitive dependencies and create a set of relations that satisfy the following definition (applied to relations with only one candidate key):

Definition of Third Normal Form (3NF)

DEFINITION: A relation is in third normal form (3NF) if it is in second normal form and no nonkey attribute is transitively dependent on the key.

Paraphrasing Kent (see the Bibliography), you can remember the characteristics of third normal form by saying that each nonkey attribute must depend on the key, the whole key, and nothing but the key.)

In checking a second-normal-form relation for third normal form, we look to see if any nonkey attribute is functionally dependent on another nonkey attribute. If such a functional dependency exists, we remove the functionally dependent attribute from the relation, placing it in a new relation with its determinant. The determinant can remain in the original relation. For our STU example, since the undesirable dependency is CREDITS → STATUS, we form the set of relations

```
STU2 (STOID, STUNAME, MAJOR, CREDITS)
STATS (CREDITS, STATUS)
```

This decomposition is shown in Figure 7.5(b). In fact, we may decide not to store STATUS in the database at all, and calculate the STATUS for those views that need it. In that case, we simply drop the STATS relation.

This definition of third normal form is the original one developed by Codd. It is sufficient for relations that have a single candidate key, but was found to be deficient in cases where there are multiple candidate keys and where candidate keys are composite and overlapping. Therefore, an improved definition of third normal form, named for its developers, Boyce and Codd, was formulated to take care of all cases.

7.10 Boyce–Codd Normal Form

Definition of Boyce–Codd Normal Form (BCNF)

DEFINITION: A relation is in Boyce–Codd normal form (BCNF) if and only if every determinant is a candidate key.

For a relation with only one candidate key, third normal form and Boyce–Codd normal form are equivalent. In fact, some authors refer to this definition as the standard one for third normal form. Note that unlike previous forms that started with relations already in the lower normal form, this definition does not state that the relation must first be 2NF and then satisfy an additional condition. Therefore, to check for BCNF, we simply identify all the determinants and make sure that they are candidate keys. However, all relations that are BCNF are also 3NF.

Returning to our earlier examples to check directly for BCNF, we see that for our STUDENT relation of Section 7.8, shown in Figure 7.5(a), the determinants are STUID and CREDITS. Since CREDITS is not a candidate key, this relation is not BCNF. Performing the projections as we did in Section 7.9, the resulting relations are BCNF. For the relation CLASS in Section 7.7, shown in Figure 7.4(a), we found the determinant COURSE#, which is not (by itself) a candidate key, and STUID, also not a candidate key. Therefore, the CLASS relation is not BCNF. However, the relations resulting from the projections are BCNF.

Let us consider an example in which we have overlapping candidate keys:

```
FACULTY (FACNAME, DEPT, OFFICE, RANK, DATEHIRED)
```

For this example, we will assume that although faculty names are not unique, no two faculty members within a single department have the same name. We also assume that each faculty member has only one office, identified in OFFICE. A department may have several faculty offices, and faculty members from the same department may share offices. From these assumptions we see

```
OFFICE → DEPT
FACNAME,DEPT → OFFICE, RANK, DATEHIRED
FACNAME,OFFICE → DEPT, RANK, DATEHIRED
```

Our overlapping candidate keys are FACNAME,DEPT and FACNAME,OFFICE. If we choose FACNAME,DEPT as our primary key, we are left with a determinant, OFFICE, that is not a candidate key. This violates BCNF. (Note that the relation is 3NF. It is 1NF because each cell has a single value. It is 2NF because even though the key has two attributes, both are needed to determine the nonkey attributes, OFFICE, RANK, and DATEHIRED. It is 3NF because there is no transitive dependency. Even though OFFICE determines DEPT, the 3NF definition is not violated, since DEPT is part of the key.) To reach BCNF, we can decompose the FACULTY relation by projection into

```
FAC1 (DEPT, OFFICE)
FAC2 (FACNAME, OFFICE, RANK, DATEHIRED)
```

The key of the first relation is OFFICE, since a department may have several offices, but each office belongs to only one department. It is clearly BCNF, since the only determinant is the key. The key of the second is FACNAME,OFFICE. It is also BCNF, since its only determinant is the key. Note, however, that our final scheme does not show the functional dependency FACNAME,DEPT → OFFICE, RANK, DATEHIRED. (Note: If we had chosen FACNAME,OFFICE as the primary key of the original FACULTY relation, we would have OFFICE → DEPT. Since OFFICE is not a candidate key, the relation would not be BCNF. In fact, it would not be 2NF, since DEPT would not be fully functionally dependent on the key, FACNAME,OFFICE.)

A relation may have overlapping candidate keys and still be BCNF. For example, let us consider the relation

```
STU (STUNAME, STUADD, MAJOR, GPA)
```

Here, we will assume that a student can have only one major and that both of the combinations STUNAME,STUADD and STUNAME,MAJOR are unique, so we have two overlapping candidate keys. If we choose STUNAME,STUADD as the primary key, our only other determinant, STUNAME,MAJOR, is a candidate key. Therefore, the relation STU is already BCNF, and there is no need to decompose it.

Any relation that is not BCNF can be decomposed into BCNF relations by the method illustrated. However, it may not always be desirable to transform the relation into BCNF. In particular, if there is functional dependency that is not preserved when we perform the decomposition (i.e., the determinant and the attributes it determines end up in different relations), it is difficult to enforce the functional dependency in the database and an important constraint is lost. In that case, it is preferable to settle for 3NF, which always allows us to preserve dependencies. Our FACULTY relation provided an example in which we lost a functional dependency by normalizing to BCNF. In the resulting relations, the attributes FACNAME and DEPT appeared in different relations, and we had no way to express the fact that they determined all other attributes.

Comprehensive Example of Functional Dependencies

To summarize the various normal forms defined by functional dependencies, let us consider the following relation that stores information about projects in a large business:

```
WORK → (PROJNAME, PROJMGR, EMPID, HOURS, EMPNAME, BUDGET, STARTDATE,
SALARY, EMPMGR, EMPDEPT, RATING)
```

PROJNAME	PROJMGR	EMPID	HOURS	EMPNAME	BUDGET	STARTDATE	SALARY	EMPMGR	EMPDEPT	RATING
Jupiter	Smith	E101	25	Jones	100000	890115	40000	Levine	10	9
Jupiter	Smith	E105	10	Adams	100000	890115	55000	Jones	12	
Jupiter	Smith	E110	10	Rivera	100000	890115	43000	Levine	10	8
Maxima	Lee	E101	15	Jones	200000	900301	40000	Levine	10	
Maxima	Lee	E105	30	Adams	200000	900301	55000	Jones	12	
Maxima	Lee	E120	15	Tanaka	200000	900301	45000	Jones	15	

Figure 7.6 WORK Table

Figure 7.6 shows an instance of this relation. Assume that:

1. Each project has a unique name, but names of employees and managers are not unique.
2. Each project has one manager, whose name is stored in PROJMGR.

3. Many employees may be assigned to work on each project, and an employee may be assigned to more than one project. HOURS tells the number of hours per week that a particular employee is assigned to work on a particular project.
4. BUDGET stores the amount budgeted for a project, and STARTDATE gives the starting date for a project.
5. SALARY gives the annual salary of an employee.
6. EMPMGR gives the name of the employee's manager, who is not the same as the project manager.
7. EMPDEPT gives the employee's department. Department names are unique. The employee's manager is the manager of the employee's department.
8. RATING gives the employee's rating for a particular project. The project manager assigns the rating at the end of the employee's work on that project.

Using the assumptions, we find the following functional dependencies to begin with:

```
PROJNAME → PROJMGR, BUDGET, STARTDATE
EMPID → EMPNAME, SALARY, EMPMGR, EMPDEPT
PROJNAME, EMPID → HOURS, RATING
```

Since we assumed that people's names were not unique, EMPMGR does not functionally determine EMPDEPT. However, since department names are unique and each department has only one manager, we add

```
EMPDEPT → EMPMGR
```

Because people's names are not unique, PROJMGR does not determine PROJNAME. You may ask whether PROJMGR → BUDGET. Although it may be the case that the manager determines the budget in the English sense of the word, meaning that the manager comes up with the figures for the budget, you should recall that functional dependency does not mean "causes" or "figures out." Similarly, although the project manager assigns a rating to the employee, there is no functional dependency between PROJMGR and RATING.

Since we see that every attribute is functionally dependent on the combination PROJNAME,EMPID, we will choose that combination as our primary key, and see what normal form we have.

First Normal Form: With our composite key, each cell would be single-valued, so WORK is 1NF.

Second Normal Form: We found partial (nonfull) dependencies,

```
PROJNAME → PROJMGR, BUDGET, STARTDATE
EMPID → EMPNAME, SALARY, EMPMGR, EMPDEPT.
```

We transform the relation into an equivalent set of 2NF relations by projection, resulting in

```
PROJ (PROJNAME, PROJMGR, BUDGET, STARTDATE)
EMP (EMPID, EMPNAME, SALARY, EMPMGR, EMPDEPT)
WORK1 (PROJNAME, EMPID, HOURS, RATING)
```

Third Normal Form: Using the set of projections, {PROJ,EMP,WORK1}, we test each relation for 3NF. PROJ is 3NF because no nonkey attribute functionally determines another nonkey attribute. In EMP we have a transitive dependency, EMPDEPT → EMPMGR. Therefore, we need to rewrite EMP as

```
EMP1 (EMPID, EMPNAME, SALARY, EMPDEPT)
DEPT (EMPDEPT, EMPMGR)
```

In WORK1, there is no functional dependency between HOURS and RATING, so that relation is already 3NF. Our new set of 3NF relations is therefore

```
PROJ (PROJNAME, PROJMGR, BUDGET, STARTDATE)
EMP1 (EMPID, EMPNAME, SALARY, EMPDEPT)
DEPT (EMPDEPT, EMPMGR)
WORK1 (PROJNAME, EMPID, HOURS, RATING)
```

Boyce–Codd Normal Form: Our new 3NF set of relations is also BCNF, since, in each relation, the only determinant is the primary key.

We already know that our original relation could not have been BCNF, since it was not even 2NF. To verify the fact that WORK was not BCNF, all we needed to do was find a determinant that was not a candidate key. Any one of EMPID, EMPDEPT, or PROJNAME is sufficient to show that the original WORK relation was not BCNF.

7.12 Multivalued Dependencies

Although Boyce–Codd normal form removes any anomalies due to functional dependencies, further research by Fagin led to the identification of another type of dependency called a multivalued dependency that can cause similar design problems. To illustrate the concept of multivalued dependencies, consider the following relation:

```
FACULTY (FACID, DEPT, COMMITTEE)
```

Here we will assume that a faculty member can belong to more than one department. For example, a professor can be hired jointly by the CSC and Math departments. A faculty member can belong to several college-wide committees, each identified by the committee name. There is no relationship between department and committee. Figure 7.7(a) shows an unnormalized version of this relation. To make the relation 1NF, we must rewrite it as shown in Figure 7.7(b). Notice that we are forced to write all the combinations of DEPT values with COMMITTEE values for each faculty member, for else it would appear that there is some relationship between DEPT and COMMITTEE.

For example, without the second row, it would appear that F101 is on the Budget committee only as a member of the CSC Department, but not as a member of the Math Department. Also note that the key of the relation is now FACID,DEPT,COMMITTEE. The resulting relation is BCNF, since the only determinant is the key. Although we have taken care of all functional dependencies, there are still update, insertion, and deletion anomalies. If we want to update a committee that F101 belongs to from Budget to Advancement, we need to make two changes or have inconsistent data. If we want to insert the record of a faculty member who belongs to one or more departments but who is not on any committee, we are unable to do so, since COMMITTEE is part of the key, so null values are not permitted in that column. This is an insertion anomaly. Similarly, if F221 drops membership on the Library committee, we lose all the rest of the information stored for him or her, since we are not permitted to have a null value for an attribute of the key. Earlier, we found that similar problems were caused by functional dependencies, but there are none in this example, so we need to identify a new cause. Although a faculty member is not associated with only one department, he or she is certainly associated with a particular set of departments. Similarly, a faculty member is associated with a specific set of committees at any given time. The set of departments for a particular FACID is independent of the set of committees for that faculty member. This independence is the cause of the problems. To see how we can correct them, we need another definition.

Definition of Multivalued Dependency

DEFINITION Let R be a relation having attributes or sets of attributes A, B, and C. There is a multivalued dependence of attribute B on attribute A if and only if the set of B values associated with a given A value is independent of the C values.

We write this as $A \twoheadrightarrow B$ and read it as "A multidetermines B." If R has at least three attributes, A, B, and C, then in $R(A,B,C)$, if $A \twoheadrightarrow B$, then $A \twoheadrightarrow C$ as well.

Unlike rules for functional dependencies, which made certain tuples illegal in relations, multivalued dependencies make certain tuples essential in a relation. In the normalized FACULTY table shown in Figure 7.7 (b), we were forced to write certain tuples because we had included others. For example, when we wrote the combination of F101 with both the CSC and Math Department values, we had to write two tuples for each of the committee values, Budget and Curriculum, and place each department value in a tuple with each committee value. An alternative definition of multivalued dependency describes the tuples that must appear.

Alternative Definition of Multivalued Dependency

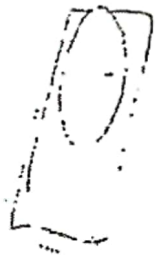
If R is a relation with multivalued dependency

$$A \twoheadrightarrow B$$

then in any table for R, if two tuples, $t1$ and $t2$, have the same A value, then there must exist two other tuples, $t3$ and $t4$, obeying these rules:

1. $t3$ and $t4$ have the same A value as $t1$ and $t2$.
2. $t3$ has the same B value as $t1$.
3. $t4$ has the same B value as $t2$.
4. If $R - B$ represents the attributes of R that are not in B, then $t2$ and $t3$ have the same values for $R - B$.
5. $t1$ and $t4$ have the same values for $R - B$.

The dependency $A \twoheadrightarrow B$ is called a trivial multivalued dependency if B is a subset of A or $A \cup B$ is all of R. Now we are ready to consider fourth normal form.



7.8 Fourth Normal Form

Definition of Fourth Normal Form (4NF)

DEFINITION: A relation is in fourth normal form (4NF) if and only if it is in Boyce–Codd normal form and there are no nontrivial multivalued dependencies.

Our FACULTY relation shown in Figure 7.7(b) is not in fourth normal form because of the two nontrivial multivalued dependencies

```
FACID ↠ DEPT
FACID ↠ COMMITTEE )
```

When a relation is BCNF but not 4NF, we can transform it into an equivalent set of 4NF relations by projection. We form two separate relations, placing in each the attribute that multidetermines the others, along with one of the multidetermined attributes or sets of attributes.

For our FACULTY relation, we form the two projections

```
FAC1 (FACID, DEPT)
FAC2 (FACID, COMMITTEE)
```

These relations, shown in Figure 7.7(c), are both in 4NF.)

7.9 Lossless Decomposition

In splitting relations by projection, we were very explicit about the method of decomposition to use. In particular, we were careful to use projections that could be "undone" by joining the resulting tables, so that the original table would result. By the term *original*

FACID	DEPT	COMMITTEE
F101	CSC	Budget
	Math	Curriculum
F221	Bio	Library
F330	Eng	Budget
		Admissions

Figure 7.7(a) Unnormalized FACULTY Table

FACID	DEPT	COMMITTEE
F101	CSC	Budget
F101	Math	Budget
F101	CSC	Curriculum
F101	Math	Curriculum
F221	Bio	Library
F330	Eng	Budget
F330	Eng	Admissions

Figure 7.7(b) Normalized FACULTY Table Showing Multivalued Dependencies

FACID	DEPT
F101	CSC
F101	Math
F221	Bio
F330	Eng

FAC1 Table

FACID	COMMITTEE
F101	Budget
F101	Curriculum
F221	Library
F330	Budget
F330	Admissions

FAC2 Table

Figure 7.7(c) FAC1 and FAC2 Tables in 4NF

Figure 7.7

table we do not mean merely the structure of the table (i.e., the column names) but the actual tuples as well. Such a decomposition is called a nonloss or lossless decomposition, because it preserves all the information in the original relation. Although we have used

the word *decomposition*, we have not written a formal definition, and we do so now.

Definition of Decomposition

DEFINITION: A decomposition of a relation R is a set of relations $\{R1, R2, \ldots, Rn\}$ such that each Ri is a subset of R and the union of all of the Ri is R.

Note that the Ri need not be disjoint. Now we are ready to define lossless decomposition.

Definition of Lossless Decomposition

DEFINITION: A decomposition $\{R1, R2, \ldots, Rn\}$ of a relation R is called a lossless decomposition for R if the natural join of $R1, R2, \ldots, Rn$ produces exactly the relation R.

Not every decomposition is lossless. It is possible to produce a decomposition that is lossy, one that loses information. As an example, consider the relation

```
MARKS (STUID, COURSE#, GRADE)
```

with the instance pictured in Figure 7.8 (a). Each tuple in this table shows that a certain student received the specified grade in the course listed. Now we form the two relations

```
MARK1 (STUID, COURSE#)
MARK2 (COURSE#, GRADE)
```

by projecting over the columns indicated. The results are pictured in Figure 7.8 (b).

We now perform the natural join of these two relations, using the common column COURSE#. We observe that the result, shown in Figure 7.8 (c), contains some tuples that were not in the original relation. These spurious tuples were introduced by the operations performed. Since, without the original table, we would have no way of identifying which tuples are genuine and which are spurious, we actually lose information (even though we have more tuples) if we substitute the projections for the original relation. We can guarantee that the decomposition is lossless by making sure that for each pair of relations that will be joined, the set of common attributes is a determinant of one of the relations. We can do this by placing functionally dependent attributes in a relation with their determinants and keeping the determinants themselves in the original relation. More formally, if R is decomposed into two relations $\{R1, R2\}$, the join is lossless if and only if either of the following holds in the closure of the set of FDs for R:

$$R1 \cap R2 \rightarrow R1 - R2$$

or

$$R1 \cap R2 \rightarrow R2 - R1$$

In the MARKS example, COURSE# was not a determinant for either GRADE or STUID, so the intersection of the two projections, COURSE#, did not functionally determine either projection. We saw several examples of lossless projection when we normalized relations. For example, when we projected CLASS into {STU,CLASS2, COURSE} the projection was lossless. Since CLASS2 ∩ COURSE is COURSE# and COURSE# → COURSE (i.e., COURSE# → COURSE#, FACID, SCHED, ROOM) we can form a lossless join of CLASS2 and COURSE. Then the join of that result with STU has STUID as its intersection. Since STUID → STUID, STUNAME, the second join is also lossless.

For a decomposition involving more than two relations, the previous test cannot be used, so we present an algorithm for the general case. Given a relation $R(A1, A2, \ldots, An)$, a set of functional dependencies, F, and a decomposition of R into relations $R1$, $R2, \ldots, Rm$, the following algorithm can be used to determine whether the decomposition has a lossless join:

1. Construct an m by n table, S, with a column for each of the n attributes in R and a row for each of the m relations in the decomposition.

STUID	COURSE#	GRADE
S1001	ART103A	A
S1001	HST205A	B
S1003	CSC201A	A
S1006	ART103A	B
S1010	ART103A	D

Figure 7.8(a) Original MARKS Table

STUID	COURSE#
S1001	ART103A
S1001	HST205A
S1003	CSC201A
S1006	ART103A
S1010	ART103A

MARK1 Table

COURSE#	GRADE
ART103A	A
HST205A	B
CSC201A	A
ART103A	B
ART103A	D

MARK2 Table

Figure 7.8(b) Projections of the MARKS Table

Figure 7.8

STUID	COURSE#	GRADE
S1001	ART103A	A
S1001	ART103A	B
S1001	ART103A	D
S1001	HST205A	B
S1003	CSC201A	A
S1006	ART103A	A
S1006	ART103A	B
S1006	ART103A	D
S1010	ART103A	A
S1010	ART103A	B
S1010	ART103A	D

Figure 7.8(c) Join of MARK1 and MARK2 Tables

Figure 7.8 (Continued)

2. For each cell $S(i,j)$ of S,
   if the attribute for the column, Aj, is in the relation for the row, Ri, then place the symbol $a(j)$ in the cell
   else place the symbol $b(i,j)$ there
3. Repeat the following process until no more changes can be made to S:
   for each FD $X \rightarrow Y$ in F
   for all rows in S that have the same symbols in the columns corresponding to the attributes of X, make the symbols for the columns that represent attributes of Y equal by the following rule:
   if any row has an a value, $a(j)$, set the value of that column in all the other rows equal to $a(j)$
   if no row has an a value, then pick any one of the b values, say $b(i,j)$, and set all the other rows equal to $b(i,j)$.
4. If, after all possible changes have been made to S, a row is made up entirely of a symbols, $a(1), a(2), \ldots, a(n)$, the join is lossless. If there is no such row, the join is lossy.

Example 1. Consider the relation $R(A, B, C, D, E)$ having decomposition consisting of $R1(A, C)$, $R2(A, B, D)$, and $R3(D, E)$ with FDs $A \rightarrow C$, $AB \rightarrow D$, and $D \rightarrow E$. Referring to Figure 7.9, we construct one row for each relation in the decomposition and one column for each of the five attributes of R. For each row, we place the value a with the

R(A,B,C,D,E)
Decomposition: R1(A,C), R2(A,B,D), R3(D,E)
FDs: A → C, AB → D, D → E

	A	B	C	D	E
R1(A,C)	a(1)	b(1,2)	a(3)	b(1,4)	b(1,5)
R2(A,B,D)	a(1)	a(2)	b(2,3)	a(4)	b(2,5)
R3(D,E)	b(3,1)	b(3,2)	b(3,3)	a(4)	a(5)

Figure 7.9(a) Initial placement of values

	A	B	C	D	E
R1(A,C)	a(1)	b(1,2)	a(3)	b(1,4)	b(1,5)
R2(A,B,D)	a(1)	a(2)	a(3)	a(4)	a(5)
R3(D,E)	b(3,1)	b(3,2)	b(3,3)	a(4)	a(5)

Figure 7.9(b) Table after considering all FDs

Figure 7.9 Testing for Lossless Join

column subscript in any column whose heading represents an attribute in that relation, and the value *b* with the usual row and column subscript in the column for any attribute not in that relation. For example, in the first row, for relation *R1*(*A*, *C*), we place *a*(1) in the first column, for *A*, and *a*(3) in the third column, for *C*. Since *B* does not appear in *R1*, we place *b*(1,2) in its column. Similarly, we place *b*(1,4) in the *D* column and *b*(1,5) in the *E* column, since these attributes do not appear in *R1*. Now we consider the FD $A \rightarrow C$, and look for rows that agree on the value of the left-hand side, *A*. We find that rows 1 and 2 agree on the value *a*(1). Therefore, we can set the *C* values equal. We find that row 1 has an *a* value, *a*(3), in the *C* column, so we set the *C* column value of row 2 equal to *a*(3). Considering the second FD, $AB \rightarrow D$, we cannot find any two rows that agree on both their *A* and *B* values, so we are unable to make any changes. Now considering the FD $D \rightarrow E$, we find that row 2 and row 3 agree on their *D* values, *a*(4), so we can set their *E* values equal. Since row 3 has an E value of *a*(5), we change the *E* value of row 2 to *a*(5) as well. Now we find that the second row has all *a* values, and we conclude that the projection has the lossless join property.

Fifth Normal Form

We have progressed from each normal form to a higher one by decomposition. Fifth normal form, also called Project-Join Normal Form, is the final stage of that process.

Definition of Fifth Normal Form

DEFINITION: A relation is in fifth normal form (5NF) if no remaining nonloss projections are possible, except the trivial one in which the key appears in each projection.

If a design consists of relations that are all 5NF, then there is nothing to be gained by decomposing them further, since that would result in a loss of information. The relations are in their "simplest" useful form.

An alternative formulation of fifth normal form involves the notion of join dependency. A join dependency means that a relation can be reconstructed by taking the join of its projections. Thus if $R(A,B,C)$ is decomposed into $R1(A,B)$ and $R2(B,C)$, a join dependency exists if we can get back R by taking the join of $R1$ and $R2$. As we have seen from our MARKS example, not all projections have this property. Using this formulation, a relation is in fifth normal form if and only if every join dependency is implied by the candidate keys. Fifth normal form is not as easily verified as lower normal forms. In fact, no systematic method exists for obtaining 5NF or for ensuring that a set of relations is indeed 5NF.

7.6 Domain-Key Normal Form

The final normal form defined by Fagin involves the concepts of domain, key, and constraint. Fagin demonstrated that a relation in this form cannot have update, insertion, or deletion anomalies. Therefore, this form represents the ultimate normal form with respect to these defects.

Definition of Domain-Key Normal Form (DKNF)

DEFINITION: A relation is in domain-key normal form (DKNF) if every constraint is a logical consequence of domain constraints or key constraints.

The definition uses the terms domain, key, and constraint. As usual, the domain of an attribute is the set of allowable values for that attribute. Fagin uses the word *key* to mean what we have described as a superkey, a unique identifier for each entity. *Constraint* is a general term meaning a rule or restriction that can be verified by examining static states of the database. For a constraint to exist, we must be able to state it as a logical predicate which we can verify by examining instances of the relation. Although functional dependencies, multivalued dependencies, and join dependencies are constraints, there are other types, called "general constraints", as well. We may have rules about relationships between attributes of a relation (intrarelation constraints) that are not expressed as dependencies. For example, consider the relation STUDENT (STUID,STUNAME,MAJOR,CREDITS). Suppose that we have a rule that the STUID has a prefix that changes as the student progresses; for example, all freshman have a "1," all sophomores a "2," and so on, at the beginning of their IDs. This could be expressed as the general constraint "If the first digit of STUID is 1, then CREDITS must be between 0 and 30. If the first digit of STUID is 2, ..." For a relation to be DKNF, intrarelation constraints must be expressible as domain constraints or key constraints. We could express our constraint on STUDENT by splitting STUDENT into four different relations. For example, for freshman, we might have

```
STU1(STUID, STUNAME, MAJOR, CREDITS) with the domain constraints
1.  STUID must begin with a 1
2.  CREDITS must be between 0 and 30.
```

We would then have STU2 with similar domain constraints for sophomores, STU3 for juniors, and STU4 for seniors. Fagin's definition does not extend to interrelation constraints such as referential integrity, since his objective was to define a form that would allow general constraints to be checked within a relation by checking only that relation's domain and key constraints. Since an interrelation constraint involves two relations, it is not considered in the definition of DKNF.

Unfortunately, although the concept of domain-key normal form is simple, there is no proven method of converting a design to this form, so it remains an ideal rather than a state that can readily be achieved.

7.17 The Normalization Process

As we stated at the beginning of this chapter, the objective of normalization is to find a stable set of relations that is a faithful model of the enterprise. We found that normalization eliminated some problems of data representation and resulted in a "good" schema for the database. Analysis and synthesis are two different processes that could be used to develop the set of normalized relations.

7.17.1 Analysis

The analysis or decomposition approach begins with a list of all the attributes to be represented in the database and assumes that all of them are in a single relation called the universal relation for the database. The designer then identifies functional dependencies among the attributes and uses the techniques of decomposition explained in this chapter to split the universal relation into a set of normalized relations. Our examples in previous sections were all decomposition examples. In particular, in Section 7.11 we considered an example in which we had a single relation called WORK. WORK was actually a universal relation containing all the attributes to be stored in the database. We wrote out our assumptions and identified four functional dependencies. The functional dependencies enabled us to perform lossless projections, so that we developed a set of relations in BCNF that preserved all functional dependencies. Since there were no multivalued dependencies, the relations are also in 4NF, and since there were no remaining nontrivial lossless projections, they are in 5NF. All the constraints expressed in our list of assumptions and all the dependencies we identified in our discussion were considered in developing the final set of relations. The constraints are represented as

key constraints in the resulting relations, so we actually have a set of DKNF relations. This example illustrates the process of analysis from a universal relation. In practice, a designer often begins with some idea of what the basic entities are, and groups attributes into initial relations before beginning the decomposition process on these relations.

7.17.2 *Synthesis*

Synthesis is, in a sense, the "opposite" of analysis. In analysis we begin with a single big relation and break it up until we reach a set of smaller normalized relations. In synthesis we begin with attributes (no relation at all) and combine them into related groups, using functional dependencies to develop a set of normalized relations. If analysis is a "top-down" process, then synthesis is a "bottom-up" one. A synthesis algorithm for producing a set of 3NF relations from a set of attributes was developed by Bernstein. Although we do not use this method in this book, we will describe the basic steps:

(1) Make a list of all FDs.
(2) Group together those with the same determinant.
(3) Construct a relation for each such group.

Difficulties may be introduced in several possible ways. For example, some FDs have more attributes in the determinant than needed, and we must eliminate the extraneous attributes before grouping, or else we will not recognize that two groups actually have the same determinants and we may create a relation that is not 2NF. For example, consider the set of FDs:

$$AB \rightarrow C$$

$$A \rightarrow D$$

$$D \rightarrow B$$

From the second and third we see that, by transitivity, $A \rightarrow B$. This makes B an extraneous attribute in the first FD, since $A \rightarrow B$ by transitivity. Another problem is that some of the FDs may be redundant; meaning that we can derive them from other FDs. In that case we must eliminate the redundant FDs before grouping or third normal form will not result. An example of that occurs in the following:

$$A \rightarrow B$$

$$A \rightarrow C$$

$$B \rightarrow C$$

Here, the second FD can be derived from the first and third. A third problem is that two relations may appear to have different keys when, in fact, the keys are equivalent. Equivalent keys functionally determine each other. For example, if we had STUID and SOCSECNO as attributes of a relation STUDENT we might use STUID as the key of one group of attributes, and SOCSECNO as the key of another, and fail to realize that

the two relations should be combined. This could occur in cases where the equivalence of the candidate keys is obvious, that is, where the list of FDs specifically states that they determine each other, or, worse, where the equivalence is not obvious but could be derived from the list of FDs. A final problem could be that new functional dependencies would be introduced when equivalent keys are found. These new FDs may make some of the existing ones redundant, and the resulting relations would not be 3NF. Once these difficulties are eliminated, the process of constructing relations from attribute groups with identical or equivalent keys can begin.

Regardless of the process used, the end result should be a set of normalized relations that preserve dependencies and form lossless joins over common attributes. An important question is how far to go in the normalization process. Ideally, we try to reach DKNF. However, if that results in a decomposition that does not preserve dependencies, then we settle for less. Similarly, if we try for 5NF, 4NF, or BCNF, we may not be able to get a decomposition that preserves dependencies, so we would settle for 3NF in that case. It is always possible to find a dependency-preserving decomposition in 3NF. Even so, there may be valid reasons for choosing not to implement 3NF as well. For example, if we have attributes that are almost always used together in applications and they end up in different relations, then we will almost always have to do a join operation when we retrieve them. A familiar example of this occurs in storing addresses. Let us assume that we are storing an employee's name and address, in the relation

```
EMP (EMPID, NAME, STREET, CITY, STATE, ZIP)
```

As usual we assume that names are not unique. We have the functional dependency

```
ZIP → CITY, STATE
```

which means that the relation is not 3NF. We could normalize it by decomposition into

```
EMP1 (EMPID, NAME, STREET, ZIP)
CODES (ZIP, CITY, STREET)
```

However, this would mean that we would have to do a join whenever we want a complete address for a person. In this case we would settle for 2NF and implement the original EMP relation. In general, performance requirements must be taken into account in deciding what the final form will be.

7.18 Chapter Summary

In this chapter we dealt with a method for developing a suitable set of relations for the logical model. Three types of dependencies, functional dependencies, multivalued dependencies, and join dependencies, were found to produce problems in representing

information in the database. Many problems involved update, insertion, and deletion anomalies. An attribute *B* is said to be functionally dependent on an attribute *A* in a relation *R* if each *A* value has exactly one *B* value associated with it. Functional dependencies can be manipulated by using Armstrong's axioms or rules of inference. The set *F*+ of all functional dependencies logically implied by a given set *F* of functional dependencies is called its closure. Similarly, the closure, *A*+, of an attribute *A* in a set of functional dependencies is the set of all attributes that it functionally determines. We can compute *A*+ by a simple algorithm.

First normal form means that a relation has no multiple-valued attributes. We can normalize a relation that is not already in 1NF by flattening it. Relations in 1NF can have update, insertion, and deletion anomalies, due to partial dependencies on the key. If a nonkey attribute is not fully functionally dependent on the entire key, the relation is not in second normal form. In that case we normalize it by projection, placing each determinant that is a proper subset of the key in a new relation with its dependent attributes. The original composite key must also be kept in another relation. Relations in 2NF can also have update, insertion, and deletion anomalies. Transitive dependencies, which allow one nonkey attribute to functionally determine another nonkey attribute, cause such anomalies. In relations with only one candidate key, third normal form is achieved by using projection to eliminate transitive dependencies. The determinant causing the transitive dependency is placed in a separate relation with the attributes it determines. It is also kept in the original relation. In third normal form, each nonkey attribute is functionally dependent on the key, the whole key, and nothing but the key.

Boyce–Codd normal form requires that every determinant be a candidate key. This form is also achieved by projection, but in some cases the projection will separate determinants from their functionally dependent attributes, resulting in the loss of an important constraint. If this happens, 3NF is preferable.

Multivalued dependencies can occur when there are at least three attributes in the key and two of them have independent multiple values. Fourth normal form requires that a relation be BCNF and have no multivalued dependencies. We can achieve 4NF by projection, but we do so only if the resulting projections preserve all functional dependencies. Some projections can lose information because when they are undone by a join, spurious tuples can result. We can be sure that a decomposition is lossless if the intersection of the two projections is a determinant for one of them. A relation is in fifth normal form if there are no remaining nontrivial nonloss projections. Alternatively, a relation is in fifth normal form if every join dependency is implied by the candidate keys. There is no proven method for achieving fifth normal form. Domain-key normal form requires that every constraint be a consequence of domain constraints or key constraints. No proven method exists for producing domain-key normal form.

In analysis we begin with a universal relation, identify dependencies, and use projection to achieve a higher normal form. The opposite approach, called synthesis, begins with attributes, finds functional dependencies, and groups together functional dependencies with the same determinant, forming relations.

In deciding what normal form to choose for implementation, we consider lossless projection and dependency preservation. We generally choose the highest normal form that allows for both of these. We must balance performance against normalization in implementation.

Exercises

1. Consider the following universal relation that holds information about books in a bookstore:

```
BOOKS (TITLE, ISBN, AUTHOR, PUBLISHER_NAME, PUBLISHER_ADD,
TOTAL_COPIES_ORDERED, COPIES_IN_STOCK, PUBLICATION_DATE, CATEGORY,
SELLING_PRICE, COST)
```

Assume that:

- 1. The ISBN identifies a book uniquely. (It does not identify each copy of the book, however.)
- 2. If a book has more than one author, only the first is listed.
- 3. An author may have more than one book.
- 4. Each publisher name is unique. Each publisher has one unique address, the address of the firm's headquarters.
- 5. Titles are not unique.
- 6. TOTAL_COPIES_ORDERED is the number of copies of a particular book that the bookstore has ever ordered, while COPIES_IN_STOCK is the number still unsold in the bookstore.
- 7. Each book has only one publication date. A revision of a book is given a new ISBN.
- 8. The category may be biography, science fiction, poetry, and so on. The title alone is not sufficient to determine the category.
- 9. The SELLING_PRICE, which is the amount the bookstore charges for a book, is always 20 percent above the COST, which is the amount the bookstore pays the publisher or distributor for the book.

(a) Using these assumptions and stating any others you need to make, list all the nontrivial functional dependencies for this relation.
(b) What are the candidate keys for this relation? Identify the primary key.
(c) Is the relation in third normal form? If not, find a 3NF lossless join decomposition of BOOK that preserves dependencies.
(d) Is the relation or resulting set of relations in Boyce–Codd normal form? If not, find a lossless join decomposition that is in BCNF, if possible.

2. Consider the following relation that stores information about students living in dormitories at a college:

```
COLLEGE (STUNAME, STUID, HOMEADD, HOMEPHONE, DORMROOM, ROOMMATE_NAME,
DORMADD, STATUS, MEALPLAN, ROOMCHARGE, MEALCHARGE)
```

Assume that

1. Each student is assigned to one dormitory room and has at most one roommate. Names of students are not unique.
2. The college has several dorms. DORMROOM contains a code for the dorm and the number of the particular room assigned to the student. For example, A221 means Adams Hall, room 221. Dorm names are unique.
3. The DORMADD is the address of the dorm building. Each building has its own unique address. For example, Adams Hall may be 123 Main Street, Anytown, NY 10001.
4. STATUS tells the student's status: Freshman, Sophomore, Junior, Senior, or Graduate Student.
5. MEALPLAN tells how many meals per week the student has chosen. Each meal plan has a single MEALCHARGE associated with it.
6. The ROOMCHARGE is different for different dorms, but all students in the same dorm pay the same amount.

For this example, answer parts (a)–(d) as in Exercise 1.

3. Consider the following relations and identify the highest normal form for each, as given, stating any assumptions that you need to make:

(a) WORK1 (EMPID, EMPNAME, DATE_HIRED, JOB_TITLE, JOB_LEVEL)
(b) WORK2 (EMPID, EMPNAME, JOB_TITLE, RATING_DATE, RATER_NAME, RATING)
(c) WORK3 (EMPID, EMPNAME, PROJECT#, PROJECT_NAME, PROJ_BUDGET, EMP_MANAGER, HOURS_ASSIGNED)
(d) WORK4 (EMPID, EMPNAME, SCHOOL_ATTENDED, DEGREE, GRADUATION_DATE)
(e) WORK5 (EMPID, EMPNAME, SOCIAL_SECURITY_NUMBER, DEPENDENT_NAME, DEPENDENT_ADDRESS, RELATION_TO_EMP)

4. For each of the relations in Exercise 3, identify a primary key and

(a) If the relation is not in third normal form, find a 3NF lossless join decomposition that preserves dependencies.
(b) If the relation or resulting set of relations is not in Boyce–Codd normal form, find a lossless join decomposition that is in BCNF.

5. Consider instances of relation $R(A,B,C,D)$ shown in Figure 7.10. For each of the following sets of FDs, determine whether each of the instances is consistent with the FDs given.

(a) $A \rightarrow B,C$
$B \rightarrow D$
(b) $AB \rightarrow C$
$B \rightarrow D$
(c) $AB \rightarrow C,D$
$AC \rightarrow B$

Instance 1:

A	B	C	D
a1	b1	c1	d1
a2	b2	c1	d2
a3	b1	c2	d3
a4	b3	c2	d3

Instance 2:

A	B	C	D
a1	b1	c1	d1
a2	b1	c2	d1
a3	b2	c1	d2
a4	b2	c2	d2

Instance 3:

A	B	C	D
a1	b1	c1	d1
a2	b1	c2	d1
a1	b2	c3	d2
a2	b2	c4	d2

Figure 7.10 Instances of R(A,B,C,D)

6. Given the following set, S, of FDs:

$$S = \{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$$

(a) Find the closure of A, $A+$
(b) Find the closure of the set of FDs, $S+$

7. Examine each of the following sets of FDs and find any redundant FDs in each. Give a minimal covering for each.

(a) $B \rightarrow D$
$E \rightarrow C$
$AC \rightarrow D$
$CD \rightarrow A$
$BE \rightarrow A$

(b) $A \rightarrow C, D, E$
$B \rightarrow C, E$
$AD \rightarrow B$
$CD \rightarrow B$
$BD \rightarrow A$
$CED \rightarrow A, B, D$

(c) $D \rightarrow C$
$AB \rightarrow C$
$AD \rightarrow B$
$BD \rightarrow A$
$AC \rightarrow B$

8. Consider the relation

$$R(A, B, C, D, E)$$

with FDs

$$A \rightarrow B$$

$$BC \rightarrow D$$

$$D \rightarrow B, C$$

$$C \rightarrow A$$

(a) Identify the candidate keys of this relation.
(b) Suppose that the relation is decomposed into

$$R1(A, B)$$

$$R2(B, C, D)$$

Does this decomposition have a lossless join?

Bibliography

Armstrong, W. W. "Dependency Structures of Data Base Relationships," Proceedings of the IFIP Congress (1974) pp. 580-583.

Beeri, C., R. Fagin, and J. Howard. "A Complete Axiomatization for Functional and Multivalued Dependencies," ACM SIGMOD International Conference on Management of Data (1977) pp. 47-61.

Bernstein, P. "Synthesizing Third Normal Form Relations from Functional Dependencies," ACM Transactions on Database Systems 1:4 (December, 1976) pp. 277-298.

Bernstein, P. and N. Goodman. "What Does Boyce-Codd Normal Form Do?" Proceedings of the International Conference on Very Large Data Bases (1980) pp. 245-259.

Biskup, J., U. Dayal, and P. Bernstein. "Synthesizing Independent Database Schemas," ACM SIGMOD Conference on Management of Data (1979) pp. 143-152.

Codd, E. "Further Normalization of the Data Base Relational Model," in Rustin, R. (Ed.) *Data Base Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1972 pp. 33-64.

Date, C. *An Introduction to Database Systems*. Reading, MA: Addison-Wesley, 1986.

Elmasri, R. and S. Navathe. *Fundamentals of Database Systems*. Redwood City, CA: Benjamin/Cummings, 1989.

Fagin, R. "Multivalued Dependencies and a New Normal Form for Relational Databases," ACM Transactions on Database Systems 2:3 (September 1977) pp. 262-278.

Fagin, R. "A Normal Form for Relational Databases That is Based on Domains and Keys," ACM Transactions on Database Systems 6:3 (September, 1981) pp. 387-415.

Gottlob, G. "Computing Covers for Embedded Functional Dependencies," Proceedings of the Sixth ACM Symposium on Principles of Database Systems (1987) pp. 58-69.

Jaeschke, G. and H. Scheck. "Remarks on the Algebra of Non First Normal Form Relations," Proceedings of the First ACM Symposium on Principles of Database Systems (1982) pp. 124-138.

Kent, W. Data and Reality. Amsterdam: North-Holland, 1978.

Korth, H. and A. Silberschatz. Database System Concepts. New York: McGraw-Hill, 1986.

Liu, L. and A. Demers. "An Algorithm for Testing Lossless Joins in Relational Databases," Information Processing Letters 11:1 (1980) pp. 73-76.

Maier, D. The Theory of Relational Databases. Rockville, MD: Computer Science Press, 1983.

Mendelson, A. and D. Maier. "Generalized Mutual Dependencies and the Decomposition of Database Relations," Proceedings of the International Conference on Very Large Data Bases (1979) pp. 75-82.

Mitchell, J. "Inference Rules for Functional and Inclusion Dependencies," Proceedings of the Second ACM Symposium on Principles of Database Systems (1983) pp. 58-69.

Nicolas, J. "Mutual Dependencies and Some Results on Undecomposable Relations," Proceedings of the International Conference on Very Large Data Bases (1978) pp. 360-367.

Ozsoyoglu, Z. and L. Yuan. "A New Normal Form for Nested Relations," Transactions on Database Systems 12:1 (March, 1987) pp. 111-136.

Sadri, F. and J. Ullman. "Template Dependencies: A Large Class of Dependencies in Relational Databases and Their Complete Axiomatization," Journal of the ACM 29:2 (April, 1982) pp. 363-372.

Sciore, E. "A Complete Axiomatization for Full Join Dependencies," Journal of the ACM 29:2 (April, 1982) pp. 373-393.

Ullman, J. Principles of Database and Knowledge-Base Systems Volume I. Rockville, MD: Computer Science Press, 1986.

Vassiliou, Y. "Functional Dependencies and Incomplete Information," Proceedings of the International Conference on Very Large Data Bases (1980) pp. 260-269.

SAMPLE PROJECT

CREATING A NORMALIZED RELATIONAL MODEL FOR CLERICALTEMPS.

In the Sample Project section of Chapter 5 we created an E-R diagram for ClericalTemps. At the end of Chapter 6 we saw how to map that diagram to tables. In the present section we want to normalize the relations.

Step 7.1:

Make a list of all attributes to be stored in the database.
Our universal relation consists of the following attributes:

U = { AVAILCODE, AVERRATING, BILLDATE, BILLINGYTD, BOOKKPING, CHECK#, CLADD, CLID, CLNAME, CLPHONE, CONTACT, DAILYHOURS, DAILYRATE, DATEHIRED, DOB, EMPADD, EMPID, EMPNAME, EMPPHONE, EMPRATING, EXPECTENDDATE, FED, FEDYTD, FICA, FICAYTD, FILING, GROSS, GROSSYTD, INVOICE#, JOB#, JOBSTATUS, JOBTITLE, LASTDATEWORKED, LOCAL, LOCALYTD, NET, NETYTD, NEWBAL, OLDBAL, PAYDATE, PAYMENTSYTD, RATENAME, RATINGDATE, REPORTTOADD, REPORTTONAME, REPORTTOPHONE, SEX, STATE, STATEYTD, SSN, STARTDATE, STENO, TOTCHARGES, TOTPAID, TYPING, WORDPROC }

Although we could begin by trying to isolate functional dependencies among all these attributes, we can make our job easier by using the entities and relationships we identified in Chapter 5 and the tables to which they mapped at the end of Chapter 6.

Step 7.2:

Make a list of entities and relationships using the E-R diagram developed at the end of Chapter 5. Make a list of the tables that the entities and relationships mapped to naturally, from the Sample Project section of Chapter 6.

The entities are WORKER, CLIENT, JOB, PAYROLL, and BILL. There is a one-to-many relationship between WORKER and JOB having descriptive attributes EMPRATING, RATENAME, and RATINGDATE. There are one-to-many relationships with no descriptive attributes between CLIENT and JOB, between CLIENT and BILL, and between WORKER and PAYROLL.

The following tables resulted:

WORKER (EMPID, EMPNAME, SSN, EMPADD, EMPPHONE, DOB, SEX, DATEHIRED, LASTDATEWORKED, AVERRATING, WORDPROC, TYPING, FILING, BOOKKPING, STENO, AVAILCODE)

CLIENT (CLID, CLNAME, CLADD, CLPHONE, CONTACT, BILLINGYTD, PAYMENTSYTD)

JOB (JOB#, JOBTITLE, STARTDATE, EXPECTENDDATE, DAILYRATE, DAILYHOURS, REPORTTONAME, REPORTTOADD, REPORTTOPHONE, JOBSTATUS)

BILL (INVOICE#, BILLDATE, OLDBAL, TOTCHARGES, NEWBAL, TOTPAID)

PAYROLL (CHECK#, PAYDATE, GROSS, FED, FICA, STATE, LOCAL, NET, GROSSYTD, NETYTD, FEDYTD, FICAYTD, STATEYTD, LOCALYTD)

ASSIGNMENT (JOB#, EMPID, RATENAME, RATINGDATE, EMPRATING)

CLIENT-JOB (JOB#, CLID)

CLIENT-BILL (INVOICE#, CLID)

WORKER-PAYROLL (CHECK#, EMPID)

Step 7.3:

For each table identified in the preceding step, identify functional dependencies, multivalued dependencies, and join dependencies, and normalize the relation. Then decide

whether the table should be implemented in the highest normal form. If not, explain why.

Working first with the WORKER table, we find the following functional dependencies:

- EMPID → all other attributes
- SSN → all other attributes
- EMPPHONE → EMPADD (We assume that two different workers could have the same telephone number. For example, two members of the same household might both work for ClericalTemps. Therefore, EMPPHONE is not a candidate key.)

We could decompose WORKER into the following tables:

```
WORKER1 (EMPID, ENAME, SSN, EMPPHONE, DOB, SEX, DATEHIRED,
LASTDATEWORKED, AVERRATING, WORDPROC, TYPING, FILING, BOOKKEEPING,
STENO, AVAILCODE)

WORKER2 (EMPPHONE, EMPADD)
```

WORKER1 is in BCNF because both determinants, EMPID and SSN, are candidate keys. WORKER2 is also BCNF. However, because we would normally want a worker's address when we retrieve his or her record, we would have to do joins of these tables quite often. Therefore, we will combine them and leave the WORKER table in its original state. Note this means we are satisfied with 2NF in this case.

Now we work with the CLIENT table. We find the following functional dependencies:

- CLID → all other attributes
- CLPHONE → all other attributes (Unlike workers, clients would have unique telephone numbers. We assume that we store only one telephone number per client.)

The CLIENT relation is already in BCNF because both determinants are candidate keys.

We move on to the JOB table. The following functional dependencies exist:

- JOB# → all other attributes
- JOBTITLE → DAILYRATE (We assume that JOBTITLE has both a job name and a level, e.g., JUNIOR TYPIST, EXPERT WORDPROC)
- REPORTTOPHONE → REPORTTOADD

Because of the transitive dependencies we could break up this relation into

```
JOB1 (JOB#, JOBTITLE, STARTDATE, EXPECTENDDATE, DAILYHOURS, REPORTTONAME,
REPORTTOPHONE, JOBSTATUS)

JOB2 (REPORTTOPHONE, REPORTTOADD)

JOB3 (JOBTITLE, DAILYRATE)
```

However, because we would normally want the REPORTTOADDRESS with the job record, we will leave it in the record. Similarly, the rate is a piece of information that

would be integral to a job record, so we will choose to leave the rate in the record as well. This means that the original JOB record, which is only 2NF, will be used.

Now we work with the PAYROLL relation. We find the functional dependencies:

- CHECK# → all other attributes
- GROSS,FED,FICA,STATE,LOCAL → NET
- GROSSYTD,FEDYTD, FICAYTD, STATEYTD, LOCALYTD → NETYTD

We could break up this relation into the following relations

```
PAYROLL1 (CHECK#, PAYDATE, GROSS, FED, FICA, STATE, LOCAL,
GROSSYTD, FEDYTD, FICAYTD, STATEYTD, LOCALYTD)

PAYROLL2 (GROSS, FED, FICA, STATE, LOCAL, NET)

PAYROLL3 (GROSSYTD, FEDYTD, FICAYTD, STATEYTD, LOCALYTD, NETYTD)
```

We note that the only nonkey attribute in PAYROLL2, NET, can be calculated by a single formula from the other attributes. The same is true of NETYTD in PAYROLL3. Since all the key attributes in these two relations are already present in PAYROLL1, this is the only relation we need to store for PAYROLL. We will calculate net as needed.

Working now with the BILL relation, we find the following functional dependencies:

- INVOICE# → all other attributes
- OLDBAL, TOTCHARGES → NEWBAL

We can split this up into the following tables

```
BILL1 (INVOICE#, BILLDATE, OLDBAL, TOTCHARGES, TOTPAID)

BILL2 (OLDBAL, TOTCHARGES, NEWBAL)
```

Since NEWBAL is simply the sum of OLDBAL and TOTCHARGES, it can be calculated when needed, so we will not store it. This means that we do not need BILL2.

The ASSIGNMENT relation connects the JOB with the WORKER. Since there is only one employee per JOB#, we have the following functional dependency:

```
JOB1 → EMPID, RATENAME, RATINGDATE, EMPRATING
```

This relation is already in 5NF.

The CLIENT-JOB relation connects the CLIENT with the JOB. It has the following functional dependency

```
JOB1 → CLID
```

Therefore, it is already in 5NF. We note that this relationship table has no descriptive attributes and its only function is to show which client has generated a particular job.