CHAPTER 7

Design Theory for Relational Databases

Our study of database scheme design in Chapter 2 drew heavily on our intuition regarding what was going on in the "real world," and how that world could best be reflected by the database scheme. In most models, there is little more to design than that; we must understand the options and their implications regarding efficiency of implementation, as was discussed in Chapter 6, then rely on skill and experience to create a good design.

In the relational model, it is possible to be somewhat more mechanical in producing our design. We can manipulate our relation schemes (sets of attributes heading the columns of the relation) according to a well-developed theory, to produce a database scheme (collection of relation schemes) with certain desirable properties. In this chapter, we shall study some of the desirable properties of relation schemes and consider several algorithms for obtaining a database scheme with these properties.

Central to the design of database schemes is the idea of a *data dependency*, that is, a constraint on the possible relations that can be the current instance of a relation scheme. For example, if one attribute uniquely determines another, as SNAME apparently determines SADDR in relation SUPPLIERS of Figure 2.8, we say there is a "functional dependency" of SADDR on SNAME, or "SNAME functionally determines SADDR."

In Section 7.2 we introduce functional dependencies formally and in the following section we learn how to "reason" about functional dependencies, that is, to infer new dependencies from given ones. This ability to tell whether a functional dependency does or does not hold in a scheme with a given collection of dependencies is central to the scheme-design process. In Section 7.4 we consider lossless-join decompositions, which are scheme designs that preserve all the information of a given scheme. The following section considers the preservation of given dependencies in a scheme design, which is another desirable

property that, intuitively, says that integrity constraints found in the original design are also found in the new design.

Sections 7.6–7.8 study "normal forms," the properties of relation schemes that say there is no, or almost no, redundancy in the relation. We relate two of these forms, Boyce-Codd normal form and third normal form, to the desirable properties of database schemes as a whole—lossless join and dependency preservation—that were introduced in the previous sections.

Section 7.9 introduces multivalued dependencies, a more complex form of dependency that, like functional dependencies, occurs frequently in practice. The process of reasoning about multivalued and functional dependencies together is discussed in Section 7.9, and Section 7.10 shows how fourth normal form eliminates the redundancy due to multivalued dependencies that is left by the earlier normal forms. We close the chapter with a discussion of more complex forms of dependencies that, while not bearing directly on the database design problem as described here, serve to unify the theory and to relate the subject of dependencies to logical rules and datalog.

7.1 WHAT CONSTITUTES A BAD DATABASE DESIGN?

Before telling how to design a good database scheme, let us see why some schemes might present problems. In particular let us suppose that we had chosen, in Example 2.14, to combine the relations SUPPLIERS and SUPPLIES of Figure 2.8 into one relation SUP_INFO, with scheme:

SUP_INFO(SNAME, SADDR, ITEM, PRICE)

that included all the information about suppliers. We can see several problems with this scheme.

1. *Redundancy.* The address of the supplier is repeated once for each item supplied.
2. *Potential inconsistency* (*update anomalies*). As a consequence of the redundancy, we could update the address for a supplier in one tuple, while leaving it fixed in another. Thus, we would not have a unique address for each supplier as we feel intuitively we should.
3. *Insertion anomalies.* We cannot record an address for a supplier if that supplier does not currently supply at least one item. We might put null values in the ITEM and PRICE components of a tuple for that supplier, but then, when we enter an item for that supplier, will we remember to delete the tuple with the nulls? Worse, ITEM and SNAME together form a key for the relation, and it might be impossible to look up tuples through a primary index, if there were null values in the key field ITEM.

4. *Deletion anomalies.* The inverse to problem (3) is that should we delete all of the items supplied by one supplier, we unintentionally lose track of the supplier's address.

The reader should appreciate that the problems of redundancy and potential inconsistency are ones we have seen before and dealt with in other models. In the network model, virtual fields were introduced for the purpose of eliminating redundancy and inconsistency. In the hierarchical model, we used virtual record types for the same purpose. The object model encourages references to objects to be made by pointers rather than by copying the object.

In the present example, all the above problems go away if we replace SUP_INFO by the two relation schemes

SUPPLIERS(SNAME, SADDR)
SUPPLIES(SNAME, ITEM, PRICE)

as in Figure 2.8. Here, SUPPLIERS, gives the address for each supplier exactly once; hence there is no redundancy. Moreover, we can enter an address for a supplier even if it currently supplies no items.

Yet some questions remain. For example, there is a disadvantage to the above decomposition; to find the addresses of suppliers of Brie, we must now take a join, which is expensive, while with the single relation SUP_INFO we could simply do a selection and projection. How do we determine that the above replacement is beneficial? Are there other problems of the same four kinds present in the two new relation schemes? How do we find a good replacement for a bad relation scheme?

Dependencies and Redundancy

The balance of the chapter is devoted to answering these questions. Before proceeding though, let us emphasize the relationship between dependencies and redundancy. In general, a dependency is a statement that only a subset of all possible relations are "legal," i.e., only certain relations reflect a possible state of the real world. If not all relations are possible, it stands to reason that there will be some sort of redundancy in legal relations. That is to say, given the fact that a relation R is legal, i.e., satisfies certain dependencies, and given certain information about the current value of R, we should be able to deduce other things about the current value of R.

In the case that the dependencies are functional, the form of the redundancy is obvious. If, in our relation SUP_INFO we saw the two tuples:

SNAME	SADDR	ITEM	PRICE
Acme	16 River St.	Brie	3.49
Acme	???	Perrier	1.19

we may use the assumption that SNAME functionally determines SADDR to

deduce that the ??? stands for "16 River St." Thus, the functional dependency makes all but the first SADDR field for a given supplier redundant; we know what it is without seeing it. Conversely, suppose we did not believe the functional dependency of SADDR on SNAME holds. Then there would be no reason to believe that the ??? had any particular value, and that field would not be redundant.

When we have more general kinds of dependencies than functional dependencies, the form redundancy takes is less clear. However, in all cases, it appears that the cause and cure of the redundancy go hand-in-hand. That is, the dependency, such as that of SADDR on SNAME, not only causes the redundancy, but it permits the decomposition of the SUP_INFO relation into the SUPPLIERS and SUPPLIES relations in such a way that the original SUP_INFO relation can be recovered from the SUPPLIERS and SUPPLIES relations. We shall discuss these concepts more fully in Section 7.4.

7.2 FUNCTIONAL DEPENDENCIES

In Section 2.3 we saw that relations could be used to model the "real world" in several ways; for example, each tuple of a relation could represent an entity and its attributes or it could represent a relationship between entities. In many cases, the known facts about the real world imply that not every finite set of tuples could be the current value of some relation, even if the tuples were of the right arity and had components chosen from the right domains. We can distinguish two kinds of restrictions on relations:

1. *Restrictions that depend on the semantics of domain elements.* These restrictions depend on understanding what components of tuples mean. For example, no one is 60 feet tall, and no one with an employment history going back 37 years has age 27. It is useful to have a DBMS check for such implausible values, which probably arose because of an error when entering or computing data. The next chapter covers the expression and use of this sort of "integrity constraint." Unfortunately, they tell us little or nothing about the design of database schemes.
2. *Restrictions on relations that depend only on the equality or inequality of values.* There are other constraints that do not depend on what value a tuple has in any given component, but only on whether two tuples agree in certain components. We shall discuss the most important of these constraints, called functional dependencies, in this section, but there are other types of value-oblivious constraints that will be touched on in later sections. It is value-oblivious constraints that turn out to have the greatest impact on the design of database schemes.

Let $R(A_1, \ldots, A_n)$ be a relation scheme, and let X and Y be subsets of $\{A_1, \ldots, A_n\}$. We say $X \rightarrow Y$, read "X functionally determines Y" or "Y

functionally depends on X" if whatever relation r is the current value for R, it is not possible that r has two tuples that agree in the components for all attributes in the set X yet disagree in one or more components for attributes in the set Y. Thus, the functional dependency of supplier address on supplier name, discussed in Section 7.1, would be expressed

$$\{\text{SNAME}\} \rightarrow \{\text{SADDR}\}$$

Notational Conventions

To remind the reader of the significance of the symbols we use, we adopt the following conventions:

1. Capital letters near the beginning of the alphabet stand for single attributes.
2. Capital letters near the end of the alphabet, $U, V, \ldots, Z$, generally stand for sets of attributes, possibly singleton sets.
3. R is used to denote a relation scheme. We also name relations by their schemes; e.g., a relation with attributes A, B, and C may be called ABC.[1]
4. We use r for a relation, the current instance of scheme R. Note this convention disagrees with the Prolog convention used in Chapter 3, where R was used for the instance of a relation and r for a predicate, i.e., the name of the relation.
5. Concatenation is used for union. Thus, $A_1 \cdots A_n$ is used to represent the set of attributes $\{A_1, \ldots, A_n\}$, and XY is shorthand for $X \cup Y$. Also, XA or AX, where X is a set of attributes and A a single attribute, stands for $X \cup \{A\}$.

Significance of Functional Dependencies

Functional dependencies arise naturally in many ways. For example, if R represents an entity set whose attributes are $A_1, \ldots, A_n$, and X is a set of attributes that forms a key for the entity set, then we may assert $X \rightarrow Y$ for any subset Y of the attributes, even a set Y that has attributes in common with X. The reason is that the tuples of each possible relation r represent entities, and entities are identified by the value of attributes in the key. Therefore, two tuples that agree on the attributes in X must represent the same entity and thus be the same tuple.

Similarly, if relation R represents a many-one relationship from entity set E_1 to entity set E_2, and among the A_i's are attributes that form a key X for E_1 and a key Y for E_2, then $X \rightarrow Y$ would hold, and in fact, X functionally

[1] Unfortunately, there are cases where the natural symbol for a single attribute, e.g., Z for "zip code" or R for "room" conflicts with these conventions, and the reader will be reminded when we use a symbol in a nonstandard way.

determines any set of attributes of R. However, $Y \rightarrow X$ would not hold unless the relationship were one-to-one.

It should be emphasized that functional dependencies are statements about all possible relations that could be the value of relation scheme R. We cannot look at a particular relation r for scheme R and deduce what functional dependencies hold for R. For example, if r is the empty set, then all dependencies appear to hold, but they might not hold in general, as the value of the relation denoted by R changes. We might, however, be able to look at a particular relation for R and discover some dependencies that did not hold.

The only way to determine the functional dependencies that hold for relation scheme R is to consider carefully what the attributes mean. In this sense, dependencies are actually assertions about the real world; they cannot be proved, but we might expect them to be enforced by a DBMS if told to do so by the database designer. As we saw in Chapter 4, many relational systems will enforce those functional dependencies that follow from the fact that a key determines the other attributes of a relation.

Example 7.1: Let us consider some of the functional dependencies that we expect to hold in the YVCB database of Example 2.14 (Figure 2.8). The most basic dependencies are those that say a key determines all the attributes of the relation scheme. Thus, in SUPPLIERS we get

SNAME → SADDR

and in SUPPLIES we get

SNAME ITEM → PRICE

In CUSTOMERS we have

CNAME → CADDR BALANCE

and similar functional dependencies hold in the other relations of Figure 2.8.

We can also observe many trivial dependencies, like

SNAME → SNAME

and some that are less trivial, such as

SNAME ITEM → SADDR PRICE

which is obtained by combining the dependencies from SUPPLIERS and SUPPLIES, and realizing that attribute SNAME represents the same concept (the supplier name) in each relation. The reason we believe this functional dependency holds is that given a supplier's name and an item, we can uniquely determine an address; we ignore the item and take the address of the supplier. We can also determine a unique price, the price the given supplier charges for the given item.

The reader should understand, however, that the above dependency, unlike the others we have mentioned in this example, is not associated with a particular

relation; it is rather something we deduce from our understanding about the "semantics" of suppliers, items, addresses, and prices. We expect that this dependency will have influence on any relation scheme in which some or all of the attributes mentioned appear, but the nature of that influence, which we discuss in Section 7.4, is often subtle.

On might wonder whether a dependency like

CADDR → CNAME

holds. Looking at the sample data of Figure 4.2(a), we do not find two tuples that agree on the address but disagree on the name, simply because there are no two tuples with the same address. However, in principle, there is nothing that rules out the possibility that two customers have the same address, so we must not assert this dependency, even though it appears to hold in the only sample relation we have seen. □

Satisfaction of Dependencies

We say a relation r *satisfies* functional dependency $X \rightarrow Y$ if for every two tuples μ and ν in r such that $\mu[X] = \nu[X]$, it is also true that $\mu[Y] = \nu[Y]$. Note that like every "if ··· then" statement, it can be satisfied either by $\mu[X]$ differing from $\nu[X]$ or by $\mu[Y]$ agreeing with $\nu[Y]$. If r does not satisfy $X \rightarrow Y$, then r *violates* that dependency.

If r is an instance of scheme R, and we have declared that $X \rightarrow Y$ holds for R, then we expect that r will satisfy $X \rightarrow Y$. However, if $X \rightarrow Y$ does not hold for R in general, then r may coincidentally satisfy $X \rightarrow Y$, or it might violate $X \rightarrow Y$.

7.3 REASONING ABOUT FUNCTIONAL DEPENDENCIES

Suppose R is a relation scheme and A, B, and C are some of its attributes. Suppose also that the functional dependencies $A \rightarrow B$ and $B \rightarrow C$ are known to hold in R. We claim that $A \rightarrow C$ must also hold in R. In proof, suppose r is a relation that satisfies $A \rightarrow B$ and $B \rightarrow C$, but there are two tuples μ and ν in r such that μ and ν agree in the component for A but disagree in C. Then we must ask whether μ and ν agree on attribute B. If not, then r would violate $A \rightarrow B$. If they do agree on B, then since they disagree on C, r would violate $B \rightarrow C$. Hence r must satisfy $A \rightarrow C$.

In general, let F be a set of functional dependencies for relation scheme R, and let $X \rightarrow Y$ be a functional dependency. We say F *logically implies* $X \rightarrow Y$, written $F \models X \rightarrow Y$, if every relation r for R that satisfies the dependencies in F also satisfies $X \rightarrow Y$. We saw above that if F contains $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$ is logically implied by F. That is,

$$\{A \rightarrow B, B \rightarrow C\} \models A \rightarrow C$$

Closure of Dependency Sets

We define F^+, the *closure* of F, to be the set of functional dependencies that are logically implied by F; i.e.,

$$F^+ = \left\{ X \rightarrow Y \;\middle|\; F \models X \rightarrow Y \right\}$$

Example 7.2: Let $R = ABC$ and $F = \{A \rightarrow B, B \rightarrow C\}$. Then F^+ consists of all those dependencies $X \rightarrow Y$ such that either

1. X contains A, e.g., $ABC \rightarrow AB$, $AB \rightarrow BC$, or $A \rightarrow C$,
2. X contains B but not A, and Y does not contain A, e.g., $BC \rightarrow B$, $B \rightarrow C$, or $B \rightarrow \emptyset$, and
3. $X \rightarrow Y$ is one of the three dependencies $C \rightarrow C$, $C \rightarrow \emptyset$, or $\emptyset \rightarrow \emptyset$.

We shall discuss how to prove the above contention shortly. □

Keys

When talking about entity sets we assumed that there was a key, a set of attributes that uniquely determined an entity. There is an analogous concept for relations with functional dependencies. If R is a relation scheme with attributes $A_1A_2 \cdots A_n$ and functional dependencies F, and X is a subset of $A_1A_2 \cdots A_n$, we say X is a *key* of R if:

1. $X \rightarrow A_1A_2 \cdots A_n$ is in F^+. That is, the dependency of all attributes on the set of attributes X is given or follows logically from what is given, and
2. For no proper subset $Y \subseteq X$ is $Y \rightarrow A_1A_2 \cdots A_n$ in F^+.

We should observe that minimality, condition (2) above, was not present when we talked of keys for entity sets in Section 2.2 or keys for files in Chapter 6. The reason is that without a formalism like functional dependencies, we can not verify that a given set of attributes is minimal. The reader should be aware that in this chapter the term "key" does imply minimality. Thus, the given key for an entity set will only be a key for the relation representing that entity set if the given key was minimal. Otherwise, one or more subsets of the key for the entity set will serve as a key for the relation.

As there may be more than one key for a relation, we sometimes designate one as the "primary key." The primary key might serve as the file key when the relation is implemented, for example. However, any key could be the primary key if we desired. The term *candidate key* is sometimes used to denote any minimal set of attributes that functionally determine all attributes, with the term "key" reserved for one designated ("primary") candidate key. We also use the term *superkey* for any superset of a key. Remember that a key is a special case of a superkey.

Example 7.3: For relation R and set of dependencies F of Example 7.2 there is only one key, A, since $A \rightarrow ABC$ is in F^+, but for no set of attributes X that does not contain A, is $X \rightarrow ABC$ true.

A more interesting example is the relation scheme R(CITY, ST, ZIP), where ST stands for street address and ZIP for zip code. We expect tuple (c, s, z) in a relation for R only if city c has a building with street address s, and z is the zip code for that address in that city. It is assumed that the nontrivial functional dependencies are:

CITY ST $\rightarrow$ ZIP
ZIP $\rightarrow$ CITY

That is, the address (city and street) determines the zip code, and the zip code determines the city, although not the street address. One can easily check that {CITY, ST} and {ST, ZIP} are both keys. □

Axioms for Functional Dependencies

To determine keys, and to understand logical implications among functional dependencies in general, we need to compute F^+ from F, or at least, to tell, given F and functional dependency $X \rightarrow Y$, whether $X \rightarrow Y$ is in F^+. To do so requires that we have inference rules telling how one or more dependencies imply other dependencies. In fact, we can do more; we can provide a *complete* set of inference rules, meaning that from a given set of dependencies F, the rules allow us to deduce all the true dependencies, i.e., those in F^+. Moreover, the rules are *sound*, meaning that using them, we cannot deduce from F any false dependency, i.e., a dependency that is not in F^+.

The set of rules is often called *Armstrong's axioms*, from Armstrong [1974], although the particular rules we shall present differ from Armstrong's. In what follows we assume we are given a relation scheme with set of attributes U, the *universal set* of attributes, and a set of functional dependencies F involving only attributes in U. The inference rules are:

A1: *Reflexivity.* If $Y \subseteq X \subseteq U$, then $X \rightarrow Y$ is logically implied by F. This rule gives the *trivial dependencies*, those that have a right side contained in the left side. The trivial dependencies hold in every relation, which is to say, the use of this rule depends only on U, not on F.

A2: *Augmentation.* If $X \rightarrow Y$ holds, and Z is any subset of U, then $XZ \rightarrow YZ$. Recall that X, Y, and Z are sets of attributes, and XZ is conventional shorthand for $X \cup Z$. It is also important to remember that the given dependency $X \rightarrow Y$ might be in F, or it might have been derived from dependencies in F using the axioms we are in the process of describing.

A3: *Transitivity.* If $X \rightarrow Y$ and $Y \rightarrow Z$ hold, then $X \rightarrow Z$ holds.

Example 7.4: Consider the relation scheme $ABCD$ with functional dependencies $A \rightarrow C$ and $B \rightarrow D$. We claim AB is a key for $ABCD$ (in fact, it is the only key). We can show AB is a superkey by the following steps:

1. $A \rightarrow C$ (given)
2. $AB \rightarrow ABC$ [augmentation of (1) by AB]
3. $B \rightarrow D$ (given)
4. $ABC \rightarrow ABCD$ [augmentation of (3) by ABC]
5. $AB \rightarrow ABCD$ [transitivity applied to (2) and (4)]

To show AB is a key, we must also show that neither A nor B by themselves functionally determine all the attributes. We could show that A is not a superkey by exhibiting a relation that satisfies the given dependencies (1) and (3) above, yet does not satisfy $A \rightarrow ABCD$, and we could proceed similarly for B. However, we shall shortly develop an algorithm that makes this test mechanical, so we omit this step here. □

Soundness of Armstrong's Axioms

It is relatively easy to prove that Armstrong's axioms are sound; that is, they lead only to true conclusions. It is rather more difficult to prove completeness, that they can be used to make every valid inference about dependencies. We shall tackle the soundness issue first.

Lemma 7.1: Armstrong's axioms are sound. That is, if $X \rightarrow Y$ is deduced from F using the axioms, then $X \rightarrow Y$ is true in any relation in which the dependencies of F are true.

Proof: A1, the reflexivity axiom, is clearly sound. We cannot have a relation r with two tuples that agree on X yet disagree on some subset of X. To prove A2, augmentation, suppose we have a relation r that satisfies $X \rightarrow Y$, yet there are two tuples μ and ν that agree on the attributes of XZ but disagree on YZ. Since they cannot disagree on any attribute of Z, μ and ν must disagree on some attribute in Y. But then μ and ν agree on X but disagree on Y, violating our assumption that $X \rightarrow Y$ holds for r. The soundness of A3, the transitivity axiom, is a simple extension of the argument given previously that $A \rightarrow B$ and $B \rightarrow C$ imply $A \rightarrow C$. We leave this part of the proof as an exercise. □

Additional Inference Rules

There are several other inference rules that follow from Armstrong's axioms. We state three of them in the next lemma. Since we have proved the soundness of A1, A2, and A3, we are entitled to use them in the proof that follows.

Lemma 7.2:

a) *The union rule.* $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.
b) *The pseudotransitivity rule.* $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

c) *The decomposition rule.* If $X \rightarrow Y$ holds, and $Z \subseteq Y$, then $X \rightarrow Z$ holds.

Proof:

a) We are given $X \rightarrow Y$, so we may augment by X to infer $X \rightarrow XY$. We are also given $X \rightarrow Z$, so we may augment by Y to get $XY \rightarrow YZ$. By transitivity, $X \rightarrow XY$ and $XY \rightarrow YZ$ imply $X \rightarrow YZ$.

b) Given $X \rightarrow Y$, we may augment by W to get $WX \rightarrow WY$. Since we are given $WY \rightarrow Z$, transitivity tells us $WX \rightarrow Z$.

c) $Y \rightarrow Z$ follows from reflexivity, so by the transitivity rule, $X \rightarrow Z$. □

An important consequence of the union and decomposition rules is that if $A_1, \ldots, A_n$ are attributes, then $X \rightarrow A_1, \ldots, A_n$ holds if and only if $X \rightarrow A_i$ holds for each i. Thus, singleton right sides on functional dependencies are sufficient. We shall discuss this matter in more detail when we take up the subject of "minimal covers" for sets of functional dependencies.

Closures of Attribute Sets

Before tackling the completeness issue, it is important to define the closure of a set of attributes with respect to a set of functional dependencies. Let F be a set of functional dependencies on set of attributes U, and let X be a subset of U. Then X^+, the *closure* of X (*with respect to* F) is the set of attributes A such that $X \rightarrow A$ can be deduced from F by Armstrong's axioms.[2] The central fact about the closure of a set of attributes is that it enables us to tell at a glance whether a dependency $X \rightarrow Y$ follows from F by Armstrong's axioms. The next lemma tells how.

Lemma 7.3: $X \rightarrow Y$ follows from a given set of dependencies F using Armstrong's axioms if and only if $Y \subseteq X^+$; here, the closure of X is taken with respect to F.

Proof: Let $Y = A_1 \cdots A_n$ for set of attributes $A_1, \ldots, A_n$, and suppose $Y \subseteq X^+$. By definition of X^+, $X \rightarrow A_i$ is implied by Armstrong's axioms for all i. By the union rule, Lemma 7.2(a), $X \rightarrow Y$ follows.

Conversely, suppose $X \rightarrow Y$ follows from the axioms. For each i, $X \rightarrow A_i$ holds by the decomposition rule, Lemma 7.2(c), so $Y \subseteq X^+$. □

Completeness of Armstrong's Axioms

We are now ready to prove that Armstrong's axioms are complete. We do so by showing that if F is the given set of dependencies, and $X \rightarrow Y$ cannot be proved by Armstrong's axioms, then there must be a relation in which the dependencies of F all hold but $X \rightarrow Y$ does not; that is, F does not logically imply $X \rightarrow Y$.

[2] Do not confuse closures of sets of dependencies with closures of sets of attributes, even though the same notation is used for each.

Theorem 7.1: Armstrong's axioms are sound and complete.

Proof: Soundness is Lemma 7.1, so we have to prove completeness. Let F be a set of dependencies over attribute set U, and suppose $X \rightarrow Y$ cannot be inferred from the axioms. Consider the relation r with the two tuples shown in Figure 7.1. First we show that all dependencies in F are satisfied by r. Intuitively, a dependency $V \rightarrow W$ violated by r allows us to "push out" X^+ beyond the value that it rightfully has when given set of dependencies F.

Suppose $V \rightarrow W$ is in F but is not satisfied by r. Then $V \subseteq X^+$, or else the two tuples of r disagree on some attribute of V, and therefore, could not violate $V \rightarrow W$. Also, W cannot be a subset of X^+, or $V \rightarrow W$ would be satisfied by the relation r. Let A be an attribute of W not in X^+. Since $V \subseteq X^+$, $X \rightarrow V$ follows from Armstrong's axioms by Lemma 7.3. Dependency $V \rightarrow W$ is in F, so by transitivity we have $X \rightarrow W$. By reflexivity, $W \rightarrow A$, so by transitivity again, $X \rightarrow A$ follows from the axioms. But then, by definition of the closure, A is in X^+, which we assumed not to be the case. We conclude by contradiction that each $V \rightarrow W$ in F is satisfied by r.

Attributes of X^+				Other attributes			
1	1	⋯	1	1	1	⋯	1
1	1	⋯	1	0	0	⋯	0

Figure 7.1 A relation r showing F does not logically imply $X \rightarrow Y$.

Now we must show that $X \rightarrow Y$ is not satisfied by r. Suppose it is satisfied. As $X \subseteq X^+$ is obvious, it follows that $Y \subseteq X^+$, else the two tuples of r agree on X but disagree on Y. But then Lemma 7.3 tells us that $X \rightarrow Y$ can be inferred from the axioms, which we assumed not to be the case. Therefore, $X \rightarrow Y$ is not satisfied by r, even though each dependency in F is. We conclude that whenever $X \rightarrow Y$ does not follow from F by Armstrong's axioms, F does not logically imply $X \rightarrow Y$. That is, the axioms are complete. □

Theorem 7.1 has some interesting consequences. We defined X^+ to be the set of attributes A such that $X \rightarrow A$ followed from the given dependencies F using the axioms. We now see that an equivalent definition of X^+ is the set of A such that $F \models X \rightarrow A$. Another consequence is that although we defined F^+ to be the set of dependencies that were logically implied by F, we can also take F^+ to mean the set of dependencies that follow from F by Armstrong's axioms.

Computing Closures

It turns out that computing F^+ for a set of dependencies F is a time-consuming task in general, simply because the set of dependencies in F^+ can be large even if F itself is small. Consider the set

$$F = \{A \to B_1, A \to B_2, \ldots, A \to B_n\}$$

Then F^+ includes all of the dependencies $A \to Y$, where Y is a subset of $\{B_1, B_2, \ldots, B_n\}$. As there are 2^n such sets Y, we could not expect to list F^+ conveniently, even for reasonably sized n.

At the other extreme, computing X^+, for a set of attributes X, is not hard; it takes time proportional to the length of all the dependencies in F, written out. By Lemma 7.3 and the fact that Armstrong's axioms are sound and complete, we can tell whether $X \to Y$ is in F^+ by computing X^+ with respect to F. A simple way to compute X^+ is the following.

Algorithm 7.1: Computation of the Closure of a Set of Attributes with Respect to a Set of Functional Dependencies.

INPUT: A finite set of attributes U, a set of functional dependencies F on U, and a set $X \subseteq U$.

OUTPUT: X^+, the closure of X with respect to F.

METHOD: We compute a sequence of sets of attributes $X^{(0)}, X^{(1)}, \ldots$ by the rules:

1. $X^{(0)}$ is X.
2. $X^{(i+1)}$ is $X^{(i)}$ union the set of attributes A such that there is some dependency $Y \to Z$, in F, A is in Z, and $Y \subseteq X^{(i)}$.

Since $X = X^{(0)} \subseteq \cdots \subseteq X^{(i)} \subseteq \cdots \subseteq U$, and U is finite, we must eventually reach i such that $X^{(i)} = X^{(i+1)}$. Since each $X^{(j+1)}$ is computed only in terms of $X^{(j)}$, it follows that $X^{(i)} = X^{(i+1)} = X^{(i+2)} = \cdots$. There is no need to compute beyond $X^{(i)}$ once we discover $X^{(i)} = X^{(i+1)}$. We can (and shall) prove that X^+ is $X^{(i)}$ for this value of i. □

Example 7.5: Let F consist of the following eight dependencies:

$$\begin{array}{ll} AB \to C & D \to EG \\ C \to A & BE \to C \\ BC \to D & CG \to BD \\ ACD \to B & CE \to AG \end{array}$$

and let $X = BD$. To apply Algorithm 7.1, we let $X^{(0)} = BD$. To compute $X^{(1)}$ we look for dependencies that have a left side B, D, or BD. There is only one, $D \to EG$, so we adjoin E and G to $X^{(0)}$ and make $X^{(1)} = BDEG$. For $X^{(2)}$, we look for left sides contained in $X^{(1)}$ and find $D \to EG$ and $BE \to C$. Thus, $X^{(2)} = BCDEG$. Then, for $X^{(3)}$ we look for left sides contained in $BCDEG$ and find, in addition to the two previously found, $C \to A$, $BC \to D$, $CG \to BD$, and $CE \to AG$. Thus $X^{(3)} = ABCDEG$, the set of all attributes. It therefore comes as no surprise that $X^{(3)} = X^{(4)} = \cdots$. Thus, $(BD)^+ = ABCDEG$. □

Now we must address ourselves to the problem of proving that Algorithm 7.1 is correct. It is easy to prove that every attribute placed in some $X^{(j)}$ belongs in X^+, but harder to show that every attribute in X^+ is placed in some $X^{(j)}$.

Theorem 7.2: Algorithm 7.1 correctly computes X^+.

Proof: First we show by induction on j that if A is placed in $X^{(j)}$ during Algorithm 7.1, then A is in X^+; i.e., if A is in the set $X^{(i)}$ returned by Algorithm 7.1, then A is in X^+.

Basis: $j = 0$. Then A is in X, so by reflexivity, $X \to A$.

Induction: Let $j > 0$ and assume that $X^{(j-1)}$ consists only of attributes in X^+. Suppose A is placed in $X^{(j)}$ because A is in Z, $Y \to Z$ is in F, and $Y \subseteq X^{(j-1)}$. Since $Y \subseteq X^{(j-1)}$, we know $Y \subseteq X^+$ by the inductive hypothesis. Thus, $X \to Y$ by Lemma 7.3. By transitivity, $X \to Y$ and $Y \to Z$ imply $X \to Z$. By reflexivity, $Z \to A$, so $X \to A$ by another application of the transitivity rule. Thus, A is in X^+.

Now we prove the converse: if A is in X^+, then A is in the set returned by Algorithm 7.1. Suppose A is in X^+, but A is not in that set $X^{(i)}$ returned by Algorithm 7.1. Notice that $X^{(i)} = X^{(i+1)}$, because that is the condition under which Algorithm 7.1 produces an answer.

Consider a relation r similar to that of Figure 7.1; r has two tuples that agree on the attributes of $X^{(i)}$ and disagree on all other attributes. We claim r satisfies F. If not, let $U \to V$ be a dependency in F that is violated by r. Then $U \subseteq X^{(i)}$ and V cannot be a subset of $X^{(i)}$, if the violation occurs (the same argument was used in the proof of Theorem 7.1). Thus, $X^{(i+1)}$ cannot be the same as $X^{(i)}$ as supposed.

Thus, relation r must also satisfy $X \to A$. The reason is that A is assumed to be in X^+, and therefore, $X \to A$ follows from F by Armstrong's axioms. Since these axioms are sound, any relation satisfying F satisfies $X \to A$. But the only way $X \to A$ could hold in r is if A is in $X^{(i)}$, for if not, then the two tuples of r, which surely agree on X, would disagree on A and violate $X \to A$. We conclude that A is in the set $X^{(i)}$ returned by Algorithm 7.1. □

Equivalences Among Sets of Dependencies.

Let F and G be sets of dependencies. We say F and G are *equivalent* if

$$F^+ = G^+$$

It is easy to test whether F and G are equivalent. For each dependency $Y \to Z$

in F, test whether $Y \to Z$ is in G^+ using Algorithm 7.1 to compute Y^+ with respect to G and then checking whether $Z \subseteq Y^+$. If some dependency $Y \to Z$ in F is not in G^+, then surely $F^+ \neq G^+$. If every dependency in F is in G^+, then every dependency $V \to W$ in F^+ is in G^+, because a proof that $V \to W$ is in G^+ can be formed by taking a proof that each $Y \to Z$ in F is in G^+, and following it by a proof from F that $V \to W$ is in F^+.

To test whether each dependency in G is also in F^+, we proceed in an analogous manner. Then F and G are equivalent if and only if every dependency in F is in G^+, and every dependency in G is in F^+.

Minimal Covers

We can find, for a given set of dependencies, an equivalent set with a number of useful properties. A simple and important property is that the right sides of dependencies be split into single attributes.

Lemma 7.4: Every set of functional dependencies F is equivalent to a set of dependencies G in which no right side has more than one attribute.

Proof: Let G be the set of dependencies $X \to A$ such that for some $X \to Y$ in F, A is in Y. Then $X \to A$ follows from $X \to Y$ by the decomposition rule. Thus $G \subseteq F^+$. But $F \subseteq G^+$, since if $Y = A_1 \cdots A_n$, then $X \to Y$ follows from $X \to A_1, \ldots, X \to A_n$ using the union rule. Thus, F and G are equivalent. □

It turns out to be useful, when we develop a design theory for database schemes, to consider a stronger restriction on covers than that the right sides have but one attribute. We say a set of dependencies F is *minimal* if:

1. Every right side of a dependency in F is a single attribute.
2. For no $X \to A$ in F is the set $F - \{X \to A\}$ equivalent to F.
3. For no $X \to A$ in F and proper subset Z of X is $F - \{X \to A\} \cup \{Z \to A\}$ equivalent to F.

Intuitively, (2) guarantees that no dependency in F is redundant. Incidentally, it is easy to test whether $X \to A$ is redundant by computing X^+ with respect to $F - \{X \to A\}$. We leave this observation as an exercise.

Condition (3) guarantees that no attribute on any left side is redundant. We also leave as an exercise the fact that the following test checks for redundant attributes on the left. Attribute B in X is redundant for the dependency $X \to A$ if and only if A is in $(X - \{B\})^+$ when the closure is taken with respect to F.

As each right side has only one attribute by (1), surely no attribute on the right is redundant. If G is a set of dependencies that is minimal in the above sense, and G is equivalent to F, then we say G is a *minimal cover* for F.

Theorem 7.3: Every set of dependencies F has a minimal cover.

Proof: By Lemma 7.4, assume no right side in F has more than one attribute. We repeatedly search for violations of conditions (2) [redundant dependencies] and (3) [redundant attributes in left sides], and modify the set of dependencies accordingly. As each modification either deletes a dependency or deletes an attribute in a dependency, the process cannot continue forever, and we eventually reach a set of dependencies with no violations of (1), (2), or (3).

For condition (2), we consider each dependency $X \to Y$ in the current set of dependencies F, and if $F - \{X \to Y\}$ is equivalent to F, then delete $X \to Y$ from F. Note that considering dependencies in different orders may result in the elimination of different sets of dependencies. For example, given the set F:

$$\begin{array}{ll} A \to B & A \to C \\ B \to A & C \to A \\ B \to C & \end{array}$$

we can eliminate both $B \to A$ and $A \to C$, or we can eliminate $B \to C$, but we cannot eliminate all three.

For condition (3), we consider each dependency $A_1 \cdots A_k \to B$ in the current set F, and each attribute A_i in its left side, in some order. If

$$F - \{A_1 \cdots A_k \to B\} \cup \{A_1 \cdots A_{i-1} A_{i+1} \cdots A_k \to B\}$$

is equivalent to F, then delete A_i from the left side of $A_1 \cdots A_k \to B$. Again, the order in which attributes are eliminated may affect the result. For example, given

$$AB \to C \qquad A \to B \qquad B \to A$$

we can eliminate either A or B from $AB \to C$, but we cannot eliminate them both.

We leave as an exercise the proof that it is sufficient first to eliminate all violations of (3), then all violations of (2), but not vice versa. □

Example 7.6: Let us consider the dependency set F of Example 7.5. If we use the algorithm of Lemma 7.4 to split right sides we are left with:

$$\begin{array}{lll} AB \to C & D \to E & CG \to B \\ C \to A & D \to G & CG \to D \\ BC \to D & BE \to C & CE \to A \\ ACD \to B & & CE \to G \end{array}$$

Clearly $CE \to A$ is redundant, since it is implied by $C \to A$. $CG \to B$ is redundant, since $CG \to D$, $C \to A$, and $ACD \to B$ imply $CG \to B$. Then no more dependencies are redundant. However, $ACD \to B$ can be replaced by $CD \to B$, since $C \to A$ is given, and therefore $CD \to B$ can be deduced from $ACD \to B$ and $C \to A$. Now, no further reduction by (2) or (3) is possible. Thus, one minimal cover for F is that shown in Figure 7.2(a).

Another minimal cover, constructed from F by eliminating $CE \to A$,

$CG \rightarrow D$, and $ACD \rightarrow B$, is shown in Figure 7.2(b). Note that the two minimal covers have different numbers of dependencies. □

(a)	(b)
$AB \rightarrow C$	$AB \rightarrow C$
$C \rightarrow A$	$C \rightarrow A$
$BC \rightarrow D$	$BC \rightarrow D$
$CD \rightarrow B$	$D \rightarrow E$
$D \rightarrow E$	$D \rightarrow G$
$D \rightarrow G$	$BE \rightarrow C$
$BE \rightarrow C$	$CG \rightarrow B$
$CG \rightarrow D$	$CE \rightarrow G$
$CE \rightarrow G$	

Figure 7.2 Two minimal covers.

7.4 LOSSLESS-JOIN DECOMPOSITION

The *decomposition* of a relation scheme $R = \{A_1, A_2, \ldots, A_n\}$ is its replacement by a collection $\rho = \{R_1, R_2, \ldots, R_k\}$ of subsets of R such that

$$R = R_1 \cup R_2 \cup \cdots \cup R_k$$

There is no requirement that the R_i's be disjoint. One of the motivations for performing a decomposition is that it may eliminate some of the problems mentioned in Section 7.1.

Example 7.7: Let us reconsider the SUP_INFO relation scheme introduced in Section 7.1, but as a shorthand, let the attributes be S (SNAME), A (SADDR), I (ITEM), and P (PRICE). The functional dependencies we have assumed are $S \rightarrow A$ and $SI \rightarrow P$. We mentioned in Section 7.1 that replacement of the relation scheme $SAIP$ by the two schemes SA and SIP makes certain problems go away. For example, in $SAIP$ we cannot store the address of a supplier unless the supplier provides at least one item. In SA, there does not have to be an item supplied to record an address for the supplier. □

One might question whether all is as rosey as it looks, when we replace $SAIP$ by SA and SIP in Example 7.7. For example, suppose we have a relation r as the current value of $SAIP$. If the database uses SA and SIP instead of $SAIP$, we would naturally expect the current relation for these two relation schemes to be the projection of r onto SA and SIP, that is $r_{SA} = \pi_{SA}(r)$ and $r_{SIP} = \pi_{SIP}(r)$.

How do we know that r_{SA} and r_{SIP} contain the same information as r? One way to tell is to check that r can be computed knowing only r_{SA} and r_{SIP}.

We claim that the only way to recover r is by taking the natural join of r_{SA} and r_{SIP}. The reason is that, as we shall prove in the next lemma, if we let $s = r_{SA} \bowtie r_{SIP}$, then $\pi_{SA}(s) = r_{SA}$, and $\pi_{SIP}(s) = r_{SIP}$. If $s \neq r$, then given r_{SA} and r_{SIP} there is no way to tell whether r or s was the original relation for scheme $SAIP$. That is, if the natural join doesn't recover the original relation, then there is no way whatsoever to recover it uniquely.

Lossless Joins

If R is a relation scheme decomposed into schemes $R_1, R_2, \ldots, R_k$, and D is a set of dependencies, we say the decomposition *has a lossless join* (with respect to D), or *is a lossless-join decomposition* (with respect to D) if for every relation r for R satisfying D:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \cdots \bowtie \pi_{R_k}(r)$$

that is, every relation r is the natural join of its projections onto the R_i's. As we saw, the lossless-join property is necessary if the decomposed relation is to be recoverable from its decomposition.

Some basic facts about project-join mappings follow in Lemma 7.5. First we introduce some notation. If $\rho = (R_1, R_2, \ldots, R_k)$ is a decomposition, then m_ρ is the mapping defined by $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$. That is, $m_\rho(r)$ is the join of the projections of r onto the relation schemes in ρ. Thus, the lossless join condition with respect to a set of dependencies D can be expressed as: for all r satisfying D, we have $r = m_\rho(r)$.

Lemma 7.5: Let R be a relation scheme, $\rho = (R_1, \ldots, R_k)$ be any decomposition of R, and r be any relation for R. Define $r_i = \pi_{R_i}(r)$. Then

a) $r \subseteq m_\rho(r)$.
b) If $s = m_\rho(r)$, then $\pi_{R_i}(s) = r_i$.
c) $m_\rho(m_\rho(r)) = m_\rho(r)$.

Proof:

a) Let μ be in r, and for each i, let $\mu_i = \mu[R_i]$.[3] Then μ_i is in r_i for all i. By definition of the natural join, μ is in $m_\rho(r)$, since μ agrees with μ_i on the attributes of R_i for all i.

b) As $r \subseteq s$ by (a), it follows that $\pi_{R_i}(r) \subseteq \pi_{R_i}(s)$. That is, $r_i \subseteq \pi_{R_i}(s)$. To show $\pi_{R_i}(s) \subseteq r_i$, suppose for some particular i that μ_i is in $\pi_{R_i}(s)$. Then there is some tuple μ in s such that $\mu[R_i] = \mu_i$. As μ is in s, there is some ν_j in r_j for each j such that $\mu[R_j] = \nu_j$. Thus, in particular, $\mu[R_i]$ is in r_i. But $\mu[R_i] = \mu_i$, so μ_i is in r_i, and therefore $\pi_{R_i}(s) \subseteq r_i$. We conclude that $r_i = \pi_{R_i}(s)$.

---

[3] Recall that $\nu[X]$ refers to the tuple ν projected onto the set of attributes X.

c) If $s = m_\rho(r)$, then by (b), $\pi_{R_i}(s) = r_i$. Thus $m_\rho(s) = \bowtie_{i=1}^{k} r_i = m_\rho(r)$. □

Let us observe that if for each i, r_i is some relation for R_i, and

$$s = \bowtie_{i=1}^{k} r_i$$

then $\pi_{R_i}(s)$ is not necessarily equal to r_i. The reason is that r_i may contain "dangling" tuples that do not match with anything when we take the join. For example, if $R_1 = AB$, $R_2 = BC$, $r_1 = \{a_1b_1\}$, and $r_2 = \{b_1c_1, b_2c_2\}$, then $s = \{a_1b_1c_1\}$ and $\pi_{BC}(s) = \{b_1c_1\} \neq r_2$. However, in general, $\pi_{R_i}(s) \subseteq r_i$, and if the r_i's are each the projection of some one relation r, then $\pi_{R_i}(s) = r_i$.

The ability to store "dangling" tuples is an advantage of decomposition. As we mentioned previously, this advantage must be balanced against the need to compute more joins when we answer queries, if relation schemes are decomposed, than if they are not. When all things are considered, it is generally believed that decomposition is desirable when necessary to cure the problems, such as redundancy, described in Section 7.1, but not otherwise.

Testing Lossless Joins

It turns out to be fairly easy to tell whether a decomposition has a lossless join with respect to a set of functional dependencies.

Algorithm 7.2: Testing for a Lossless Join.

INPUT: A relation scheme $R = A_1 \cdots A_n$, a set of functional dependencies F, and a decomposition $\rho = (R_1, \ldots, R_k)$.

OUTPUT: A decision whether ρ is a decomposition with a lossless join.

METHOD: We construct a table with n columns and k rows; column j corresponds to attribute A_j, and row i corresponds to relation scheme R_i. In row i and column j put the symbol a_j if A_j is in R_i. If not, put the symbol b_{ij} there.

Repeatedly "consider" each of the dependencies $X \to Y$ in F, until no more changes can be made to the table. Each time we "consider" $X \to Y$, we look for rows that agree in all of the columns for the attributes of X. If we find two such rows, equate the symbols of those rows for the attributes of Y. When we equate two symbols, if one of them is a_j, make the other be a_j. If they are b_{ij} and b_{lj}, make them both b_{ij} or both b_{lj}, as you wish. It is important to understand that when two symbols are equated, all occurrences of those symbols in the table become the same; it is not sufficient to equate only the occurrences involved in the violation of the dependency $X \to Y$.

If after modifying the rows of the table as above, we discover that some row has become $a_1 \cdots a_n$, then the join is lossless. If not, the join is lossy (not lossless). □

Example 7.8: Let us consider the decomposition of $SAIP$ into SA and SIP as in Example 7.7. The dependencies are $S \to A$ and $SI \to P$, and the initial table is

S	A	I	P
a_1	a_2	b_{13}	b_{14}
a_1	b_{22}	a_3	a_4

Since $S \to A$, and the two rows agree on S, we may equate their symbols for A, making b_{22} become a_2. The resulting table is

S	A	I	P
a_1	a_2	b_{13}	b_{14}
a_1	a_2	a_3	a_4

Since some row, the second, has all a's, the join is lossless.

For a more complicated example, let $R = ABCDE$, $R_1 = AD$, $R_2 = AB$, $R_3 = BE$, $R_4 = CDE$, and $R_5 = AE$. Let the functional dependencies be:

$$A \to C \qquad DE \to C$$
$$B \to C \qquad CE \to A$$
$$C \to D$$

The initial table is shown in Figure 7.3(a). We can apply $A \to C$ to equate b_{13}, b_{23}, and b_{53}. Then we use $B \to C$ to equate these symbols with b_{33}; the result is shown in Figure 7.3(b), where b_{13} has been chosen as the representative symbol. Now use $C \to D$ to equate a_4, b_{24}, b_{34}, and b_{54}; the resulting symbol must be a_4. Then $DE \to C$ enables us to equate b_{13} with a_3, and $CE \to A$ lets us equate b_{31}, b_{41}, and a_1. The result is shown in Figure 7.3(c). Since the middle row is all a's, the decomposition has a lossless join. □

It is interesting to note that one might assume Algorithm 7.2 could be simplified by only equating symbols if one was an a_i. The above example shows this is not the case; if we do not begin by equating b_{13}, b_{23}, b_{33}, and b_{53}, we can never get a row of all a's.

Theorem 7.4: Algorithm 7.2 correctly determines if a decomposition has a lossless join.

Proof: Suppose the final table produced by Algorithm 7.2 does not have a row of all a's. We may view this table as a relation r for scheme R; the rows are tuples, and the a_j's and b_{ij}'s are distinct symbols, each chosen from the domain of A_j. Relation r satisfies the dependencies F, since Algorithm 7.2 modifies the table whenever a violation of the dependencies is found. We claim that $r \neq m_\rho(r)$. Clearly r does not contain the tuple $a_1a_2 \cdots a_n$. But for each R_i, there is a tuple μ_i in r, namely the tuple that is row i, such that $\mu_i[R_i]$ consists of all a's. Thus, the join of the $\pi_{R_i}(r)$'s contains the tuple with all a's, since that tuple agrees with μ_i for all i. We conclude that if the final table from

A	B	C	D	E
a_1	b_{12}	b_{13}	a_4	b_{15}
a_1	a_2	b_{23}	b_{24}	b_{25}
b_{31}	a_2	b_{33}	b_{34}	a_5
b_{41}	b_{42}	a_3	a_4	a_5
a_1	b_{52}	b_{53}	b_{54}	a_5

(a)

A	B	C	D	E
a_1	b_{12}	b_{13}	a_4	b_{15}
a_1	a_2	b_{13}	b_{24}	b_{25}
b_{31}	a_2	b_{13}	b_{34}	a_5
b_{41}	b_{42}	a_3	a_4	a_5
a_1	b_{52}	b_{13}	b_{54}	a_5

(b)

A	B	C	D	E
a_1	b_{12}	a_3	a_4	b_{15}
a_1	a_2	a_3	a_4	b_{25}
a_1	a_2	a_3	a_4	a_5
a_1	b_{42}	a_3	a_4	a_5
a_1	b_{52}	a_3	a_4	a_5

(c)

Figure 7.3 Applying Algorithm 7.2.

Algorithm 7.2 does not have a row with all a's, then the decomposition ρ does not have a lossless join; we have found a relation r for R such that $m_\rho(r) \neq r$.

Conversely, suppose the final table has a row with all a's. We can in general view any table T as shorthand for the domain relational calculus expression

$$\{a_1 a_2 \cdots a_n \mid (\exists b_{11}) \cdots (\exists b_{kn})\big(R(w_1) \wedge \cdots \wedge R(w_k)\big)\} \tag{7.1}$$

where w_i is the ith row of T. When T is the initial table, formula (7.1) defines the function m_ρ. In proof, note $m_\rho(r)$ contains tuple $a_1 \cdots a_n$ if and only if for each i, r contains a tuple with a_j in the jth component if A_j is an attribute of R_i and some arbitrary value, represented by b_{ij}, in each of the other attributes.

Since we assume that any relation r for scheme R satisfies the dependencies F, we can infer that each of the transformations to the table performed by

Algorithm 7.2 changes the table (by identifying symbols) in a way that does not affect the set of tuples produced by (7.1), as long as that expression changes to mirror the changes to the table. The detailed proof of this claim is complex, but the intuition should be clear: we are only identifying symbols if in (7.1) applied to a relation R which satisfies F, those symbols could only be assigned the same value anyway.

Since the final table contains a row with all a's, the domain calculus expression for the final table is of the form:

$$\{a_1 \cdots a_n \mid (\exists b_{11}) \cdots (\exists b_{kn})\big(R(a_1 \cdots a_n) \wedge \cdots\big)\} \tag{7.2}$$

Clearly the value of (7.2) applied to relation r for R, is a subset of r. However, if r satisfies F, then the value of (7.2) is $m_\rho(r)$, and by Lemma 7.5(a), $r \subseteq m_\rho(r)$. Thus, whenever r satisfies F, (7.2) computes exactly r, so $r = m_\rho(r)$. That is to say, the decomposition ρ has a lossless join with respect to F. □

Algorithm 7.2 can be applied to decompositions into any number of relation schemes. However, for decompositions into two schemes we can give a simpler test, the subject of the next theorem.

Theorem 7.5: If $\rho = (R_1, R_2)$ is a decomposition of R, and F is a set of functional dependencies, then ρ has a lossless join with respect to F if and only if $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ or $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$. Note that these dependencies need not be in the given set F; it is sufficient that they be in F^+.

	$R_1 \cap R_2$	$R_1 - R_2$	$R_2 - R_1$
row for R_1	$aa \cdots a$	$aa \cdots a$	$bb \cdots b$
row for R_2	$aa \cdots a$	$bb \cdots b$	$aa \cdots a$

Figure 7.4 A general two row table.

Proof: The initial table used in an application of Algorithm 7.2 is shown in Figure 7.4, although we have omitted the subscripts on a and b, which are easily determined and immaterial anyway. It is easy to show by induction on the number of symbols identified by Algorithm 7.2 that if the b in the column for attribute A is changed to an a, then A is in $(R_1 \cap R_2)^+$. It is also easy to show by induction on the number of steps needed to prove $(R_1 \cap R_2) \rightarrow Y$ by Armstrong's axioms, that any b's in the columns for attributes in Y are changed to a's. Thus, the row for R_1 becomes all a's if and only if $R_2 - R_1 \subseteq (R_1 \cap R_2)^+$, that is $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$, and similarly, the row for R_2 becomes all a's if and only if $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$. □

Example 7.9: Suppose $R = ABC$ and $F = \{A \to B\}$. Then the decomposition of R into AB and AC has a lossless join, since $AB \cap AC = A$, $AB - AC = B$,[4] and $A \to B$ holds. However, if we decompose R into $R_1 = AB$ and $R_2 = BC$, we discover that $R_1 \cap R_2 = B$, and B functionally determines neither $R_1 - R_2$, which is A, nor $R_2 - R_1$, which is C. Thus, the decomposition AB and BC does not have a lossless join with respect to $F = \{A \to B\}$, as can be seen by considering the relation $r = \{a_1b_1c_1, a_2b_1c_2\}$ for R. Then $\pi_{AB}(r) = \{a_1b_1, a_2b_1\}$, $\pi_{BC}(r) = \{b_1c_1, b_1c_2\}$, and

$$\pi_{AB}(r) \bowtie \pi_{BC}(r) = \{a_1b_1c_1, a_1b_1c_2, a_2b_1c_1, a_2b_1c_2\}$$

which is a proper superset of r. □

7.5 DECOMPOSITIONS THAT PRESERVE DEPENDENCIES

We have seen that it is desirable for a decomposition to have the lossless-join property, because it guarantees that any relation can be recovered from its projections. Another important property of a decomposition of relation scheme R into $\rho = (R_1, \ldots, R_k)$ is that the set of dependencies F for R be implied by the projection of F onto the R_i's. Formally, the *projection* of F onto a set of attributes Z, denoted $\pi_Z(F)$, is the set of dependencies $X \to Y$ in F^+ such that $XY \subseteq Z$. (Note that $X \to Y$ need not be in F; it need only be in F^+.) We say decomposition ρ *preserves* a set of dependencies F if the union of all the dependencies in $\pi_{R_i}(F)$, for $i = 1, 2, \ldots, k$ logically implies all the dependencies in F.[5]

The reason it is desirable that ρ preserves F is that the dependencies in F can be viewed as integrity constraints for the relation R. If the projected dependencies do not imply F, then should we represent R by $\rho = (R_1, \ldots, R_k)$, we could find that the current value of the R_i's represented a relation R that did not satisfy F, even if ρ had a lossless-join with respect to F. Alternatively, every update to one of the R_i's would require a join to check that the constraints were not violated.

Example 7.10: Let us reconsider the problem of Example 7.3, where we had attributes CITY, ST, and ZIP, which we here abbreviate C, S, and Z. We observed the dependencies $CS \to Z$ and $Z \to C$. The decomposition of the relation scheme CSZ into SZ and CZ has a lossless join, since

$$(SZ \cap CZ) \to (CZ - SZ)$$

That is, $Z \to C$. However, the projection of $F = \{CS \to Z,\ Z \to C\}$ onto SZ gives only the trivial dependencies (those that follow from reflexivity), while

[4] To make sense of equations like these do not forget that $A_1A_2 \cdots A_n$ stands for the set of attributes $\{A_1, A_2, \ldots, A_n\}$.

[5] Note that the converse is always true; that is, F always implies all its projections, and therefore implies their union.

the projection onto CZ gives $Z \to C$ and the trivial dependencies. It can be checked that $Z \to C$ and trivial dependencies do not imply $CS \to Z$, so the decomposition does not preserve dependencies.

For example, the join of the two relations in Figure 7.5(a) and (b) is the relation of Figure 7.5(c). Figure 7.5(a) satisfies the trivial dependencies, as any relation must. Figure 7.5(b) satisfies the trivial dependencies and the dependency $Z \to C$. However, their join in Figure 7.5(c) violates $CS \to Z$. □

S	Z
545 Tech Sq.	02138
545 Tech Sq.	02139

(a)

C	Z
Cambridge, Mass.	02138
Cambridge, Mass.	02139

(b)

C	S	Z
Cambridge, Mass.	545 Tech Sq.	02138
Cambridge, Mass.	545 Tech Sq.	02139

(c)

Figure 7.5 A join violating a functional dependency.

We should note that a decomposition may have a lossless join with respect to set of dependencies F, yet not preserve F. Example 7.10 gave one such instance. Also, the decomposition could preserve F yet not have a lossless join. For example, let $F = \{A \to B, C \to D\}$, $R = ABCD$, and $\rho = (AB, CD)$.

Testing Preservation of Dependencies

In principle, it is easy to test whether a decomposition $\rho = (R_1, \ldots, R_k)$ preserves a set of dependencies F. Just compute F^+ and project it onto all of the R_i's. Take the union of the resulting sets of dependencies, and test whether this set is equivalent to F.

However, in practice, just computing F^+ is a formidable task, since the number of dependencies it contains is often exponential in the size of F. Therefore, it is fortunate that there is a way to test preservation without actually computing F^+; this method takes time that is polynomial in the size of F.

Algorithm 7.3: Testing Preservation of Dependencies.

INPUT: A decomposition $\rho = (R_1, \ldots, R_k)$ and a set of functional dependencies F.

OUTPUT: A decision whether ρ preserves F.

METHOD: Define G to be $\cup_{i=1}^{k} \pi_{R_i}(F)$. Note that we do not compute G; we merely wish to see whether it is equivalent to F. To test whether G is equivalent to F, we must consider each $X \to Y$ in F and determine whether X^+, computed with respect to G, contains Y. The trick we use to compute X^+ without having G available is to consider repeatedly what the effect is of closing X with respect to the projections of F onto the various R_i's.

That is, define an *R-operation* on set of attributes Z with respect to a set of dependencies F to be the replacement of Z by $Z \cup \big((Z \cap R)^+ \cap R\big)$, the closure being taken with respect to F. This operation adjoins to Z those attributes A such that $(Z \cap R) \to A$ is in $\pi_R(F)$. Then we compute X^+ with respect to G by starting with X, and repeatedly running through the list of R_i's, performing the R_i-operation for each i in turn. If at some pass, none of the R_i-operations make any change in the current set of attributes, then we are done; the resulting set is X^+. More formally, the algorithm is:

```
Z := X
while changes to Z occur do
    for i := 1 to k do
        Z := Z ∪ ((Z ∩ R_i)^+ ∩ R_i)  /* closure wrt F */
```

If Y is a subset of the Z that results from executing the above steps, then $X \to Y$ is in G^+. If each $X \to Y$ in F is thus found to be in G^+, answer "yes," otherwise answer "no." □

Example 7.11: Consider set of attributes $ABCD$ with decomposition

$$\{AB, BC, CD\}$$

and set of dependencies $F = \{A \to B,\ B \to C,\ C \to D,\ D \to A\}$. That is, in F^+, each attribute functionally determines all the others. We might first imagine that when we project F onto AB, BC, and CD, we fail to get the dependency $D \to A$, but that intuition is wrong. When we project F, we really project F^+ onto the relation schemes, so projecting onto AB we get not only $A \to B$, but also $B \to A$. Similarly, we get $C \to B$ in $\pi_{BC}(F)$ and $D \to C$ in $\pi_{CD}(F)$, and these three dependencies logically imply $D \to A$. Thus, we should expect that Algorithm 7.3 will tell us that $D \to A$ follows logically from

$$G = \pi_{AB}(F) \cup \pi_{BC}(F) \cup \pi_{CD}(F)$$

We start with $Z = \{D\}$. Applying the AB-operation does not help, since

$$\{D\} \cup \big((\{D\} \cap \{A, B\})^+ \cap \{A, B\}\big)$$

is just $\{D\}$. Similarly, the BC-operation does not change Z. However, when we apply the CD-operation we get

$$\begin{aligned} Z &= \{D\} \cup \big((\{D\} \cap \{C, D\})^+ \cap \{C, D\}\big) \\ &= \{D\} \cup (\{D\}^+ \cap \{C, D\}) \\ &= \{D\} \cup (\{A, B, C, D\} \cap \{C, D\}) \\ &= \{C, D\} \end{aligned}$$

Similarly, on the next pass, the BC-operation applied to the current $Z = \{C, D\}$ produces $Z = \{B, C, D\}$, and on the third pass, the AB-operation sets Z to $\{A, B, C, D\}$, whereupon no more changes to Z are possible.

Thus, with respect to G, $\{D\}^+ = \{A, B, C, D\}$, which contains A, so we conclude that $G \models D \to A$. Since it is easy to check that the other members of F are in G^+ (in fact they are in G), we conclude that this decomposition preserves the set of dependencies F. □

Theorem 7.6: Algorithm 7.3 correctly determines if $X \to Y$ is in G^+.

Proof: Each time we add an attribute to Z, we are using a dependency in G, so when the algorithm says "yes," it must be correct. Conversely, suppose $X \to Y$ is in G^+. Then there is a sequence of steps whereby, using Algorithm 7.1 to take the closure of X with respect to G, we eventually include all the attributes of Y. Each of these steps involves the application of a dependency in G, and that dependency must be in $\pi_{R_i}(F)$ for some i, since G is the union of these projections. Let one such dependency be $U \to V$. An easy induction on the number of dependencies applied in Algorithm 7.1 shows that eventually U becomes a subset of Z, and then on the next pass the R_i-operation will surely cause all attributes of V to be added to Z if they are not already there. □

7.6 NORMAL FORMS FOR RELATION SCHEMES

A number of different properties, or "normal forms" for relation schemes with dependencies have been defined. The most significant of these are called "third normal form" and "Boyce-Codd normal form." Their purpose is to avoid the problems of redundancy and anomalies discussed in Section 7.1.

Boyce-Codd Normal Form

The stronger of these normal forms is called Boyce-Codd. A relation scheme R with dependencies F is said to be in *Boyce-Codd normal form* (*BCNF*) if whenever $X \to A$ holds in R, and A is not in X, then X is a superkey for R; that is, X is a key or contains a key. Put another way, the only nontrivial dependencies are those in which a key functionally determines one or more other attributes. In principal, we must look for violating dependencies $X \to A$ not only among the given dependencies, but among dependencies derived from them. However, we leave as an exercise the fact that if there are no violations among the given set F, and F consists only of dependencies with single attributes on the right, then there are no violations among any of the dependencies in F^+.

Example 7.12: Consider the relation scheme CSZ of Example 7.10, with dependencies $CS \rightarrow Z$ and $Z \rightarrow C$. The keys for this relation scheme are CS and SZ, as one can easily check by computing the closures of these sets of attributes and of the other nontrivial sets (CZ, C, S, and Z). The scheme CSZ with these dependencies is not in BCNF, because $Z \rightarrow C$ holds in CSZ, yet Z is not a key of CSZ, nor does it contain a key. □

Third Normal Form

In some circumstances BCNF is too strong a condition, in the sense that it is not possible to bring a relation scheme into that form by decomposition, without losing the ability to preserve dependencies. Third normal form provides most of the benefits of BCNF, as far as elimination of anomalies is concerned, yet it is a condition we can achieve for an arbitrary database scheme without giving up either dependency preservation or the lossless-join property.

Before defining third normal form, we need a preliminary definition. Call an attribute A in relation scheme R a *prime* attribute if A is a member of any key for R (recall there may be many keys). If A is not a member of any key, then A is *nonprime*.

Example 7.13: In the relation scheme CSZ of Example 7.12, all attributes are prime, since given the dependencies $CS \rightarrow Z$ and $Z \rightarrow C$, both CS and SZ are keys.

In the relation scheme $ABCD$ with dependencies $AB \rightarrow C$, $B \rightarrow D$, and $BC \rightarrow A$, we can check that AB and BC are the only keys. Thus, A, B, and C are prime, and D is nonprime. □

A relation scheme R is in *third normal form*[6] (*3NF*) if whenever $X \rightarrow A$ holds in R and A is not in X, then either X is a superkey for R, or A is prime. Notice that the definitions of Boyce-Codd and third normal forms are identical except for the clause "or A is prime" that makes third normal form a weaker condition than Boyce-Codd normal form. As with BCNF, we in principle must consider not only the given set of dependencies F, but all dependencies in F^+ to check for a 3NF violation. However, we can show that if F consists only of dependencies that have been decomposed so they have single attributes on the right, then it is sufficient to check the dependencies of F only.

Example 7.14: The relation scheme $SAIP$ from Example 7.7, with dependencies $SI \rightarrow P$ and $S \rightarrow A$ violates 3NF. A is a nonprime attribute, since the only key is SI. Then $S \rightarrow A$ violates the 3NF condition, since S is not a superkey.

However, the relation scheme CSZ from Example 7.12 is in 3NF. Since all

[6] Yes Virginia, there is a first normal form and there is a second normal form. First normal form merely states that the domain of each attribute is an elementary type, rather than a set or a record structure, as fields in the object model (Section 2.7) can be. Second normal form is only of historical interest and is mentioned in the exercises.

of its attributes are prime, no dependency could violate the conditions of third normal form. □

Motivation Behind Normal Forms

The purpose behind BCNF is to eliminate the redundancy that functional dependencies are capable of causing. Suppose we have a relation scheme R in BCNF, yet there is some redundancy that lets us predict the value of an attribute by comparing two tuples and applying a functional dependency. That is, we have two tuples that agree in set of attributes X and disagree in set of attributes Y, while in the remaining attribute A, the value in one of the tuples, lets us predict the value in the other. That is, the two tuples look like

X	Y	A
x	y_1	a
x	y_2	?

Here, x, y_1, and y_2 represent lists of values for the sets of attributes X and Y.

If we can use a functional dependency to infer the value indicated by a question mark, then that value must be a, and the dependency used must be $Z \rightarrow A$, for some $Z \subseteq X$. However, Z cannot be a superkey, because if it were, then the two tuples above, which agree in Z, would be the same tuple. Thus, R is not in BCNF, as supposed. We conclude that in a BCNF relation, no value can be predicted from any other, using functional dependencies only. In Section 7.9 we shall see that there are other ways redundancy can arise, but these are "invisible" as long as we consider functional dependencies to be the only way the set of legal relations for a scheme can be defined.

Naturally, 3NF, being weaker than BCNF, cannot eliminate all redundancy. The canonical example is the CSZ scheme of Example 7.12, which is in 3NF, yet allows pairs of tuples like

C	S	Z
c	s_1	z
?	s_2	z

where we can deduce from the dependency $Z \rightarrow C$ that the unknown value is c. Note that these tuples cannot violate the other dependency, $CS \rightarrow Z$.

7.7 LOSSLESS-JOIN DECOMPOSITION INTO BCNF

We have now been introduced to the properties we desire for relation schemes: BCNF or, failing that, 3NF. In Sections 7.4 and 7.5 we saw the two most important properties of database schemes as a whole, the lossless-join and dependency-preservation properties. Now we must attempt to put these ideas together, that is, construct database schemes with the properties we desire for

database schemes, and with each individual relation scheme having the properties we desire for relation schemes.

It turns out that any relation scheme has a lossless join decomposition into Boyce-Codd Normal Form, and it has a decomposition into 3NF that has a lossless join and is also dependency-preserving. However, there may be no decomposition of a relation scheme into Boyce-Codd normal form that is dependency-preserving. The CSZ relation scheme is the canonical example. It is not in BCNF because the dependency $Z \rightarrow C$ holds, yet if we decompose CSZ in any way such that CSZ is not one of the schemes in the decomposition, then the dependency $CS \rightarrow Z$ is not implied by the projected dependencies.

Before giving the decomposition algorithm, we need the following property of lossless-join decompositions.

Lemma 7.6: Suppose R is a relation scheme with functional dependencies F. Let $\rho = (R_1, \ldots, R_n)$ be a decomposition of R with a lossless join with respect to F, and let $\sigma = (S_1, S_2)$ be a lossless-join decomposition of R_1 with respect to $\pi_{R_1}(F)$. Then the decomposition of R into $(S_1, S_2, R_2, \ldots, R_n)$ also has a lossless join with respect to F.

Proof: Suppose we take a relation r for R, and project it onto $R_1, \ldots, R_n$ to get relations $r_1, \ldots, r_n$, respectively. Then we project r_1 onto S_1 and S_2 to get s_1 and s_2. The lossless-join property tells us we can join s_1 and s_2 to recover exactly r_1, and we can then join $r_1, \ldots, r_n$ to recover r. Since the natural join is an associative operation, by Theorem 2.1(a), the order in which we perform the join doesn't matter, so we recover r no matter in what order we take the join of $s_1, s_2, r_2, \ldots, r_n$. □

We can apply Lemma 7.6 to get a simple but time-consuming algorithm to decompose a relation scheme R into BCNF. If we find a violation of BCNF in R, say $X \rightarrow A$, we decompose R into schemes $R - A$ and XA. These are both smaller than R, since XA could not be all attributes of R (then X would surely be a superkey, and $X \rightarrow A$ would not violate BCNF). The join of $R - A$ and XA is lossless, by Theorem 7.5, because the intersection of the schemes is X, and $X \rightarrow XA$. We compute the projections of the dependencies for R onto $R - A$ and XA, then apply this decomposition step recursively to these schemes. Lemma 7.6 assures that the set of schemes we obtain by decomposing until all the schemes are in BCNF will be a lossless-join decomposition.

The problem is that the projection of dependencies can take exponential time in the size of the scheme R and the original set of dependencies. However, it turns out that there is a way to find some lossless-join decomposition into BCNF relation schemes in time that is polynomial in the size of the given set of dependencies and scheme. The technique will sometimes decompose a relation that is already in BCNF, however. The next lemma gives some useful properties of BCNF schemes.

Lemma 7.7:

a) Every two-attribute scheme is in BCNF.

b) If R is not in BCNF, then we can find attributes A and B in R, such that $(R - AB) \rightarrow A$. It may or may not be the case that $(R - AB) \rightarrow B$ as well.

Proof: For part (a), let AB be the scheme. There are only two nontrivial dependencies that can hold: $A \rightarrow B$ and $B \rightarrow A$. If neither hold, then surely there is no BCNF violation. If only $A \rightarrow B$ holds, then A is a key, so we do not have a violation. If only $B \rightarrow A$ holds, then B is a key, and if both hold, both A and B are keys, so there can never be a BCNF violation.

For (b), suppose there is a BCNF violation $X \rightarrow A$ in R. Then R must have some other attribute B, not in XA, or else X is a superkey, and $X \rightarrow A$ is not a violation. Thus, $(R - AB) \rightarrow A$ as desired. □

Lemma 7.7 lets us look for BCNF violations in a scheme R with n attributes by considering only the $n(n-1)/2$ pairs of attributes $\{A, B\}$ and computing the closure of $R - AB$ with respect to the given dependencies F, by using Algorithm 7.1. As stated, that algorithm takes $O(n^2)$ time, but a carefully designed data structure can make it run in time $O(n)$; in any event, the time is polynomial in the size of R. If for no A and B does $(R - AB)^+$ contain either A or B, then by Lemma 7.7(b) we know R is in BCNF.

It is important to realize that the converse of Lemma 7.7(b) is not true. Possibly, R is in BCNF, and yet there is such a pair $\{A, B\}$. For example, if $R = ABC$, and $F = \{C \rightarrow A,\ C \rightarrow B\}$, then R is in BCNF, yet $R - AB = C$, and C does functionally determine A (and B as well).

Before proceeding to the algorithm for BCNF decomposition, we need one more observation, about projections of dependencies. Specifically:

Lemma 7.8: If we have a set of dependencies F on R, and we project them onto $R_1 \subseteq R$ to get F_1, and then project F_1 onto $R_2 \subseteq R_1$ to get F_2, then

$$F_2 = \pi_{R_2}(F)$$

That is, we could have assumed that F was the set of dependencies for R_1, even though F presumably mentions attributes not found in R_1.

Proof: If $XY \subseteq R_2$, then $X \rightarrow Y$ is in F^+ if and only if it is in ${F_1}^+$. □

Lemma 7.8 has an important consequence. It says that if we decompose relation schemes as in Lemma 7.6, then we never actually have to compute the projected dependencies as we decompose. It is sufficient to work with the given dependencies, taking closures of attribute sets by Algorithm 7.1 when we need to, rather than computing whole projections of dependencies, which are exponential in the number of attributes in the scheme. It is this observation, together with Lemma 7.7(b), that allows us to take time that is polynomial in the size of the given scheme and the given dependencies, and yet discover some

lossless-join, BCNF decomposition of the given scheme.

Algorithm 7.4: Lossless Join Decomposition into Boyce-Codd Normal Form.

INPUT: Relation scheme R and functional dependencies F.

OUTPUT: A decomposition of R with a lossless join, such that every relation scheme in the decomposition is in Boyce-Codd normal form with respect to the projection of F onto that scheme.

METHOD: The heart of the algorithm is to take relation scheme R, and decompose it into two schemes. One will have set of attributes XA; it will be in BCNF, and the dependency $X \to A$ will hold. The second will be $R - A$, so the join of $R - A$ with XA is lossless. We then apply the decomposition procedure recursively, with $R - A$ in place of R, until we come to a scheme that meets the condition of Lemma 7.7(b); we know that scheme is in BCNF. Then, Lemma 7.6 assures us that this scheme plus the BCNF schemes generated at each step of the recursion have a lossless join.

```
Z := R; /* at all times, Z is the one scheme
  of the decomposition that may not be in BCNF */
repeat
    decompose Z into Z − A and XA, where XA is in BCNF
      and X → A; /* use the subroutine of Figure 7.6(b) */
    add XA to the decomposition;
    Z := Z − A;
until Z cannot be decomposed by Lemma 7.7(b);
add Z to the decomposition
```

(a) Main program.

```
if Z contains no A and B such that A is in (Z − AB)+ then
    /* remember all closures are taken with respect to F */
    return that Z is in BCNF and cannot be decomposed
else begin
    find one such A and B;
    Y := Z − B;
    while Y contains A and B such that (Y − AB)+ → A do
        Y := Y − B;
    return the decomposition Z − A and Y;
    /* Y here is XA in the main program */
end
```

(b) Decomposition subroutine.

Figure 7.6 Details of Algorithm 7.4.

The details of the algorithm are given in Figure 7.6. Figure 7.6(a) is the main routine, which repeatedly decomposes the one scheme Z that we do not know to be in BCNF; initially, Z is R. Figure 7.6(b) is the decomposition procedure that either determines Z cannot be decomposed, or decomposes Z into $Z - A$ and XA, where $X \to A$. The set of attributes XA is selected by starting with $Y = Z$, and repeatedly throwing out the attribute B, the one of the pair AB such that we found $X \to A$, where $X = Y - AB$. Recall that it does not matter whether $X \to B$ is true or not. □

Example 7.15: Let us consider the relation scheme $CTHRSG$, where C = course, T = teacher, H = hour, R = room, S = student, and G = grade. The functional dependencies F we assume are

$C \to T$	Each course has one teacher.
$HR \to C$	Only one course can meet in a room at one time.
$HT \to R$	A teacher can be in only one room at one time.
$CS \to G$	Each student has one grade in each course.
$HS \to R$	A student can be in only one room at one time.

Since Algorithm 7.4 does not specify the order in which pairs AB are to be considered, we shall adopt the uniform strategy of preserving the order $CTHRSG$ for the attributes and trying the first attribute against the others, in turn, then the second against the third through last, and so on.

We begin with the entire scheme, $CTHRSG$, and the first pair to consider is CT. We find that $(HRSG)^+$ contains C; it also contains T, but that is irrelevant. Thus, we begin the while-loop of Figure 7.6(b) with $A = C$, $B = T$, and $Y = CHRSG$.

Now, we try the CH pair as $\{A, B\}$, but $(RSG)^+$ contains neither C nor H. We have better luck with the next pair, CR, because $(HSG)^+$ contains R. Thus, we have $A = R$, $B = C$, and we set Y to $HRSG$, by throwing out B, as usual. With $Y = HRSG$, we have no luck until we try pair RG, when we find $(HS)^+$ contains R. Thus, we have $A = R$ and $B = G$, whereupon Y is set to HRS.

At this point, no further attributes can be thrown out of Y, because the test of Lemma 7.7(b) fails for each pair. We may therefore decompose $CTHRSG$ into

1. HRS, which plays the role of XA, with $X = HS$ and $A = R$, and
2. $Z = CTHRSG - R$, which is $CTHSG$.

We now work on $Z = CTHSG$ in the main program. The list of pairs AB that work and the remaining sets of attributes after throwing out B, is:

1. In $CTHSG$: $A = T$, $B = H$, leaves $Y = CTSG$.
2. In $CTSG$: $A = T$, $B = S$, leaves $Y = CTG$.
3. In CTG: $A = T$, $B = G$, leaves $Y = CT$.

Surely, CT is in BCNF, by Lemma 7.7(a). We thus add CT to our decomposition. Attribute T plays the role of A, so in the main program we eliminate T and progress to the scheme $Z = CHSG$, which is still not in Boyce-Codd normal form.

In $CHSG$, the first successful pair is $A = G$ and $B = H$, which leaves $Y = CSG$. No more pairs allow this scheme to be decomposed by Lemma 7.7(b), so we add CSG to the decomposition, and we apply the main program to the scheme with A removed, that is, $Z = CHS$.

This scheme, we find, cannot be decomposed by Lemma 7.7(b), so it too is in BCNF, and we are done. Notice that we get lossless joins at each stage, if we combine the schemes in the reverse of the order in which they were found. That is, $CHS \bowtie CSG$ is lossless because of the dependency $CS \rightarrow G$; $CHSG \bowtie CT$ is lossless because of the dependency $C \rightarrow T$, and $CTHSG \bowtie HRS$ is lossless because of $HS \rightarrow R$. In each case, the required functional dependency is the one of the form $X \rightarrow A$ that gets developed by the subroutine of Figure 7.6(b). By Lemma 7.6, these lossless joins imply that the complete decomposition, (HRS, CT, CSG, CHS) is lossless. □

Problems with Arbitrary BCNF Decompositions

In the decomposition of Example 7.15, the four relation schemes store the following kinds of information:

1. The location (room) of each student at each hour.
2. The teacher of each course.
3. Grades for students in courses, i.e., the students' transcripts.
4. The schedule of courses and hours for each student.

This is not exactly what we might have designed had we attempted by hand to find a lossless-join decomposition into BCNF. In particular, we cannot tell where a course meets without joining the CHS and HRS relations, and even then we could not find out if there were no students taking the course. We probably would have chosen to replace HRS by CHR, which gives the allocation of rooms by courses, rather than by students, and corresponds to the published schedule of courses found at many schools. Unfortunately, the question of "merit" of different decompositions is not one we can address theoretically. If one does not have a particular scheme in mind, for which we can simply verify that it has a lossless join and that each of its components is in BCNF, then one can try picking AB pairs at random in Algorithm 7.4, in the hope that after a few tries, one will get a decomposition that looks "natural."

Another problem with the chosen decomposition (one which is not fixed by replacing HRS by CHR) is that some of the dependencies in F, specifically $TH \rightarrow R$ and $HR \rightarrow C$, are not preserved by the decomposition. That is, the projection of F onto HRS, CT, CSG, and CHS is the closure of the following

dependencies, as the reader may check.

$$CS \rightarrow G \qquad HS \rightarrow R$$
$$C \rightarrow T \qquad HS \rightarrow C$$

Note that the last of these, $HS \rightarrow C$ is in the projection of F onto CHS, but is not a given dependency; the other three are members of F itself. These four dependencies do not imply $TH \rightarrow R$ or $HR \rightarrow C$. For example, the relation for $CTHRSG$ shown below

C	T	H	R	S	G
c_1	t	h	r_1	s_1	g_1
c_2	t	h	r_2	s_2	g_2
c_2	t	h	r_1	s_3	g_3

satisfies neither $TH \rightarrow R$ nor $HR \rightarrow C$, yet its projections onto HRS, CT, CSG, and CHS satisfy all the projected dependencies.

Efficiency of BCNF Decomposition

We claim that Algorithm 7.4 takes time that is polynomial in n, which is the length of the relation scheme R and the dependencies F, written down. We already observed that computing closures with respect to F takes time that is polynomial in n; in fact $O(n)$ time suffices if the proper data structures are used. The subroutine of Figure 7.6(b) runs on a subset Z of the attributes, which surely cannot be more than n attributes. Each time through the loop, the set Y decreases in size, so at most n iterations are possible. There are at most n^2 pairs of attributes A and B, so the test for $(Y - AB)^+ \rightarrow A$ is done at most n^3 times. Since this test takes polynomial time, and its time dominates the time of the other parts of the loop body, we conclude that the algorithm of Figure 7.6(b) takes polynomial time.

The principal cost of the main program of Figure 7.6(a) is the call to the subroutine, and this call is made only once per iteration of the loop. Since Z decreases in size going around the loop, at most n iterations are possible, and the entire algorithm is thus polynomial.

7.8 DEPENDENCY-PRESERVING 3NF DECOMPOSITIONS

We saw from Examples 7.12 and 7.14 that it is not always possible to decompose a relation scheme into BCNF and still preserve the dependencies. However, we can always find a dependency-preserving decomposition into third normal form, as the next algorithm and theorem show.

Algorithm 7.5: Dependency-Preserving Decomposition into Third Normal Form.

INPUT: Relation scheme R and set of functional dependencies F, which we assume without loss of generality to be a minimal cover.

OUTPUT: A dependency-preserving decomposition of R such that each relation scheme is in 3NF with respect to the projection of F onto that scheme.

METHOD: If there are any attributes of R not involved in any dependency of F, either on the left or right, then any such attribute can, in principle, form a relation scheme by itself, and we shall eliminate it from R.[7] If one of the dependencies in F involves all the attributes of R, then output R itself. Otherwise, the decomposition ρ to be output consists of scheme XA for each dependency $X \to A$ in F. □

Example 7.16: Reconsider the relation scheme $CTHRSG$ of Example 7.15, whose dependencies have minimal cover F:

$$\begin{array}{ll} C \to T & CS \to G \\ HR \to C & HS \to R \\ HT \to R & \end{array}$$

Algorithm 7.5 yields the set of relation schemes CT, CHR, THR, CSG, and HRS. □

Theorem 7.7: Algorithm 7.5 yields a dependency-preserving decomposition into third normal form.

Proof: Since the projected dependencies include a cover for F, the decomposition clearly preserves dependencies. We must show that the relation scheme YB, for each functional dependency $Y \to B$ in the minimal cover, is in 3NF. Suppose $X \to A$ violates 3NF for YB; that is, A is not in X, X is not a superkey for YB, and A is nonprime. Of course, we also know that $XA \subseteq YB$, and $X \to A$ follows logically from F. We shall consider two cases, depending on whether or not $A = B$.

Case 1: $A = B$. Then since A is not in X, we know $X \subseteq Y$, and since X is not a superkey for YB, X must be a proper subset of Y. But then $X \to B$, which is also $X \to A$, could replace $Y \to B$ in the supposed minimal cover, contradicting the assumption that $Y \to B$ was part of the given minimal cover.

Case 2: $A \neq B$. Since Y is a superkey for YB, there must be some $Z \subseteq Y$ that is a key for YB. But A is in Y, since we are assuming $A \neq B$, and A cannot be in Z, because A is nonprime. Thus Z is a proper subset of Y, yet $Z \to B$ can replace $Y \to B$ in the supposedly minimal cover, again providing a contradiction. □

There is a modification to Algorithm 7.5 that avoids unnecessary decomposition. If $X \to A_1, \ldots, X \to A_n$ are dependencies in a minimal cover, then we may use the one relation scheme $XA_1 \cdots A_n$ in place of the n relation schemes $XA_1, \ldots, XA_n$. It is left as an exercise that the scheme $XA_1 \cdots A_n$ is in 3NF.

[7] Sometimes it is desirable to have two or more attributes, say A and B, appear together in a relation scheme, even though there is no functional dependency involving them. There may simply be a many-many relationship between A and B. An idea of Bernstein [1976] is to introduce a dummy attribute θ and functional dependency $AB \to \theta$, to force this association. After completing the design, attribute θ is eliminated.

Decompositions into Third Normal Form with a Lossless Join and Preservation of Dependencies

As seen, we can decompose any relation scheme R into a set of schemes

$$\rho = (R_1, \ldots, R_k)$$

such that ρ has a lossless join and each R_i is in BCNF (and therefore in 3NF). We can also decompose R into $\sigma = (S_1, \ldots, S_m)$ such that σ preserves the set of dependencies F, and each S_j is in 3NF. Can we find a decomposition into 3NF that has both the lossless join and dependency-preservation properties? We can, if we simply adjoin to σ a relation scheme X that is a key for R, as the next theorem shows.

Theorem 7.8: Let σ be the 3NF decomposition of R constructed by Algorithm 7.5, and let X be a key for R. Then $\tau = \sigma \cup \{X\}$ is a decomposition of R with all relation schemes in 3NF; the decomposition preserves dependencies and has the lossless join property.

Proof: It is easy to show that any 3NF violation in X implies that a proper subset of X functionally determines X, and therefore R, so X would not be a key in that case. Thus X, as well as the members of σ, are in 3NF. Clearly τ preserves dependencies, since σ does.

To show that τ has a lossless join, apply the tabular test of Algorithm 7.2. We can show that the row for X becomes all a's, as follows. Consider the order $A_1, A_2, \ldots, A_k$ in which the attributes of $R - X$ are added to X^+ in Algorithm 7.1. Surely all attributes are added eventually, since X is a key. We show by induction on i that the column corresponding to A_i in the row for X is set to a_i in the test of Algorithm 7.2.

The basis, $i = 0$, is trivial. Assume the result for $i - 1$. Then A_i is added to X^+ because of some given functional dependency $Y \to A_i$, where

$$Y \subseteq X \cup \{A_1, \ldots, A_{i-1}\}$$

Then YA_i is in σ, and the rows for YA_i and X agree on Y (they are all a's) after the columns of the X-row for $A_1, \ldots, A_{i-1}$ are made a's. Thus, these rows are made to agree on A_i during the execution of Algorithm 7.2. Since the YA_i-row has a_i there, so must the X-row. □

Obviously, in some cases τ is not the smallest set of relation schemes with the properties of Theorem 7.8. We can throw out relation schemes in τ one at a time as long as the desired properties are preserved. Many different database schemes may result, depending on the order in which we throw out schemes, since eliminating one may preclude the elimination of others.

Example 7.17: We could take the union of the database scheme produced for $CTHRSG$ in Example 7.16 with the key SH, to get a decomposition that has a lossless join and preserves dependencies. It happens that SH is a subset of HRS, which is one of the relation schemes already selected. Thus, SH may be eliminated, and the database scheme of Example 7.16, that is

$$(CT, CHR, THR, CSG, HRS)$$

suffices. Although some proper subsets of this set of five relation schemes are lossless join decompositions, we can check that the projected dependencies for any four of them do not imply the complete set of dependencies F. □

A Cautionary Word About Decompositions

Given tools like Algorithms 7.4 and 7.5, one is often tempted to "decompose the heck" out of a relation scheme. It is important to remember that not every lossless-join decomposition step is beneficial, and some can be harmful. The most common mistake is to decompose a scheme that is already in BCNF, just because it happens to have a lossless-join decomposition that preserves dependencies.

For example, we might have a relation giving information about employees, say I, the unique ID-number for employees, N, the employee's name, D the department in which he works, and S, the salary. Since I is the only key in this situation, we have $I \rightarrow A$ for each other attribute A. It is therefore possible to decompose this scheme into IN, ID, and IS. This decomposition is easily seen to have a lossless join, because I, the only attribute in the intersection of any pair, functionally determines all the attributes; it also clearly preserves the dependencies $I \rightarrow NDS$.

However, the scheme $INDS$ is itself in BCNF, and offers significant advantages for the answering of queries relating attributes other than I. For example, if we wanted to know the name and salary of all of the employees in the toy department, we would have to join $IN \bowtie ID \bowtie IS$ in the decomposed database scheme, yet we could answer the query without taking any joins if we left the relation scheme intact (and with an index on department, we could answer this query quite efficiently). Further, the decomposed scheme requires that the employee ID number be repeated in many places, although it is not, technically, redundant.

The moral is that when applying the theory of decomposition, one should remember that decomposition is a last resort, used to solve the problems of redundancy and anomalies, not as an end in itself. When applying Algorithm 7.4, we should avoid doing a decomposition, even if Lemma 7.7(b) tells us it can be done, should the scheme already be in BCNF. When using Algorithm 7.5, consider combining the relation schemes that result, should there be no ~~BCNF~~ 3NF violations created by doing so.

7.9 MULTIVALUED DEPENDENCIES

In previous sections we have assumed that the only possible kind of data dependency is functional. In fact there are many plausible kinds of dependencies, and at least one other, the multivalued dependency, appears frequently in the "real world." Suppose we are given a relation scheme R, and X and Y are subsets of R. Intuitively, we say that $X \twoheadrightarrow Y$, read "X *multidetermines* Y," or "there is a *multivalued dependency* of Y on X," if given values for the attributes of X there is a set of zero or more associated values for the attributes of Y, and this set of Y-values is not connected in any way to values of the attributes in $R - X - Y$.

Formally, we say $X \twoheadrightarrow Y$ holds in R if whenever r is a relation for R, and μ and ν are two tuples in r, with $\mu[X] = \nu[X]$ (that is, μ and ν agree on the attributes of X), then r also contains tuples ϕ and ψ, where

1. $\phi[X] = \psi[X] = \mu[X] = \nu[X]$.
2. $\phi[Y] = \mu[Y]$ and $\phi[R - X - Y] = \nu[R - X - Y]$.
3. $\psi[Y] = \nu[Y]$ and $\psi[R - X - Y] = \mu[R - X - Y]$.[8]

That is, we can exchange the Y-values of μ and ν to obtain two new tuples ϕ and ψ that must also be in r. Note we did not assume that X and Y are disjoint in the above definition.

Example 7.18: Let us reconsider the relation scheme $CTHRSG$ introduced in the previous section. In Figure 7.7 we see a possible relation for this relation scheme. In this simple case there is only one course with two students, but we see several salient facts that we would expect to hold in any relation for this relation scheme. A course can meet for several hours, in different rooms each time. Each student has a tuple for each class taken and each session of that class. His grade for the class is repeated for each tuple.

C	T	H	R	S	G
CS101	Deadwood	M9	222	Weenie	B+
CS101	Deadwood	W9	333	Weenie	B+
CS101	Deadwood	F9	222	Weenie	B+
CS101	Deadwood	M9	222	Grind	C
CS101	Deadwood	W9	333	Grind	C
CS101	Deadwood	F9	222	Grind	C

Figure 7.7 A sample relation for scheme $CTHRSG$.

[8] Note we could have eliminated clause (3). The existence of tuple ψ follows from the existence of ϕ when we apply the definition with μ and ν interchanged.

Thus, we expect that in general the multivalued dependency $C \twoheadrightarrow HR$ holds; that is, there is a set of hour-room pairs associated with each course and disassociated from the other attributes. For example, in the formal definition of a multivalued dependency we may take $X \twoheadrightarrow Y$ to be $C \twoheadrightarrow HR$ and choose

$\mu =$	CS101	Deadwood	M9	222	Weenie	B+
$\nu =$	CS101	Deadwood	W9	333	Grind	C

i.e., μ is the first tuple, and ν the fifth, in Figure 7.7. Then we would expect to be able to exchange $\mu[HR] =$ (M9, 222) with $\nu[HR] =$ (W9, 333) to get the two tuples

$\phi =$	CS101	Deadwood	M9	222	Grind	C
$\psi =$	CS101	Deadwood	W9	333	Weenie	B+

A glance at Figure 7.7 affirms that ϕ and ψ are indeed in r; they are the fourth and second tuples, respectively.

It should be emphasized that $C \twoheadrightarrow HR$ holds not because it held in the one relation of Figure 7.7. It holds because any course c, if it meets at hour h_1 in room r_1, with teacher t_1 and student s_1 who is getting grade g_1, and it also meets at hour h_2 in room r_2 with teacher t_2 and student s_2 who is getting grade g_2, will also meet at hour h_1 in room r_1 with teacher t_2 and student s_2 who is getting grade g_2.

Note also that $C \twoheadrightarrow H$ does not hold, nor does $C \twoheadrightarrow R$. In proof, consider relation r of Figure 7.7 with tuples μ and ν as above. If $C \twoheadrightarrow H$ held, we would expect to find tuple

CS101	Deadwood	M9	333	Grind	C

in r, which we do not. A similar observation about $C \twoheadrightarrow R$ can be made. There are a number of other multivalued dependencies that hold, however, such as $C \twoheadrightarrow SG$ and $HR \twoheadrightarrow SG$. There are also trivial multivalued dependencies like $HR \twoheadrightarrow R$. We shall in fact prove that every functional dependency $X \to Y$ that holds implies that the multivalued dependency $X \twoheadrightarrow Y$ holds as well. □

Axioms for Functional and Multivalued Dependencies

We shall now present a sound and complete set of axioms for making inferences about a set of functional and multivalued dependencies over a set of attributes U. The first three are Armstrong's axioms for functional dependencies only; we repeat them here.

A1: *Reflexivity for functional dependencies.* If $Y \subseteq X \subseteq U$, then $X \to Y$.

A2: *Augmentation for functional dependencies.* If $X \to Y$ holds, and $Z \subseteq U$, then $XZ \to YZ$.

A3: *Transitivity for functional dependencies.* $\{X \to Y, Y \to Z\} \models X \to Z$.

The next three axioms apply to multivalued dependencies.

A4: *Complementation for multivalued dependencies.*

$$\{X \twoheadrightarrow Y\} \models X \twoheadrightarrow (U - X - Y)$$

A5: *Augmentation for multivalued dependencies.* If $X \twoheadrightarrow Y$ holds, and $V \subseteq W$, then $WX \twoheadrightarrow VY$.

A6: *Transitivity for multivalued dependencies.*

$$\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$$

It is worthwhile comparing A4–A6 with A1–A3. Axiom A4, the complementation rule, has no counterpart for functional dependencies. Axiom A1, reflexivity, appears to have no counterpart for multivalued dependencies, but the fact that $X \twoheadrightarrow Y$ whenever $Y \subseteq X$, follows from A1 and the rule (Axiom A7, to be given) that if $X \to Y$ then $X \twoheadrightarrow Y$. A6 is more restrictive than its counterpart transitivity axiom, A3. The more general statement, that $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$ imply $X \twoheadrightarrow Z$, is false. For instance, we saw in Example 7.18 that $C \twoheadrightarrow HR$ holds, and surely $HR \twoheadrightarrow H$ is true, yet $C \twoheadrightarrow H$ is false. To compensate partially for the fact that A6 is weaker than A3, we use a stronger version of A5 than the analogous augmentation axiom for functional dependencies, A2. We could have replaced A2 by: $X \to Y$ and $V \subseteq W$ imply $WX \to VY$, but for functional dependencies, this rule is easily proved from A1, A2, and A3.

Our last two axioms relate functional and multivalued dependencies.

A7: $\{X \to Y\} \models X \twoheadrightarrow Y$.

A8: If $X \twoheadrightarrow Y$ holds, $Z \subseteq Y$, and for some W disjoint from Y, we have $W \to Z$, then $X \to Z$ also holds.

Soundness and Completeness of the Axioms

We shall not give a proof that axioms A1–A8 are sound and complete. Rather, we shall prove that some of the axioms are sound, that is, they follow from the definitions of functional and multivalued dependencies, leaving the soundness of the rest of the axioms, as well as a proof that any valid inference can be made using the axioms (completeness of the axioms), for an exercise.

Let us begin by proving A6, the transitivity axiom for multivalued dependencies. Suppose some relation r over set of attributes U satisfies $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, but violates $X \twoheadrightarrow (Z - Y)$. Then there are tuples μ and ν in r, where $\mu[X] = \nu[X]$, but the tuple ϕ, where $\phi[X] = \mu[X]$, $\phi[Z - Y] = \mu[Z - Y]$, and

$$\phi[U - X - (Z - Y)] = \nu[U - X - (Z - Y)]$$

is not in r.[9] Since $X \twoheadrightarrow Y$ holds, it follows that the tuple ψ, where $\psi[X] =$

[9] Recall that we pointed out the definition of multivalued dependencies could require

$\mu[X]$, $\psi[Y] = \nu[Y]$, and

$$\psi[U - X - Y] = \mu[U - X - Y]$$

is in r. Now ψ and ν agree on Y, so since $Y \twoheadrightarrow Z$, it follows that r has a tuple ω, where $\omega[Y] = \nu[Y]$, $\omega[Z] = \psi[Z]$, and

$$\omega[U - Y - Z] = \nu[U - Y - Z]$$

We claim that $\omega[X] = \mu[X]$, since on attributes in $Z \cap X$, ω agrees with ψ, which agrees with μ. On attributes of $X - Z$, ω agrees with ν, and ν agrees with μ on X. We also claim that $\omega[Z - Y] = \mu[Z - Y]$, since ω agrees with ψ on $Z - Y$, and ψ agrees with μ on $Z - Y$. Finally, we claim that $\omega[V] = \nu[V]$, where $V = U - X - (Z - Y)$. In proof, surely ω agrees with ν on $V - Z$, and by manipulating sets we can show $V \cap Z = (Y \cap Z) - X$. But ω agrees with ψ on Z, and ψ agrees with ν on Y, so ω agrees with ν on $V \cap Z$ as well as on $V - Z$. Therefore ω agrees with ν on V. If we look at the definition of ϕ, we now see that $\omega = \phi$. But we claimed that ω is in r, so ϕ is in r, contrary to our assumption. Thus $X \twoheadrightarrow Z - Y$ holds after all, and we have proved A6.

Now let us prove A8. Suppose in contradiction that we have a relation r in which $X \twoheadrightarrow Y$ and $W \rightarrow Z$ hold, where $Z \subseteq Y$, and $W \cap Y$ is empty, but $X \rightarrow Z$ does not hold. Then there are tuples ν and μ in r such that $\nu[X] = \mu[X]$, but $\nu[Z] \neq \mu[Z]$. By $X \twoheadrightarrow Y$ applied to ν and μ, there is a tuple ϕ in r, such that $\phi[X] = \mu[X] = \nu[X]$, $\phi[Y] = \mu[Y]$, and $\phi[U - X - Y] = \nu[U - X - Y]$. Since $W \cap Y$ is empty, ϕ and ν agree on W. As $Z \subseteq Y$, ϕ and μ agree on Z. Since ν and μ disagree on Z, it follows that ϕ and ν disagree on Z. But this contradicts $W \rightarrow Z$, since ϕ and ν agree on W but disagree on Z. We conclude that $X \rightarrow Z$ did not fail to hold, and we have verified rule A8.

The remainder of the proof of the following theorem is left as an exercise.

Theorem 7.9: (Beeri, Fagin, and Howard [1977]). Axioms A1–A8 are sound and complete for functional and multivalued dependencies. That is, if D is a set of functional and multivalued dependencies over a set of attributes U, and D^+ is the set of functional and multivalued dependencies that follow logically from D (i.e., every relation over U that satisfies D also satisfies the dependencies in D^+), then D^+ is exactly the set of dependencies that follow from D by A1–A8. □

Additional Inference Rules for Multivalued Dependencies

There are a number of other rules that are useful for making inferences about functional and multivalued dependencies. Of course, the union, decomposition,

only the existence of ϕ, not the additional existence of ψ as in the third clause of the definition. Thus, the violation of a multivalued dependency can be stated as the absence of ϕ (not ϕ or ψ) from the relation r.

and pseudotransitivity rules mentioned in Lemma 7.1 still apply to functional dependencies. Some other rules are:

1. *Union rule for multivalued dependencies.*
   $$\{X \twoheadrightarrow Y, X \twoheadrightarrow Z\} \models X \twoheadrightarrow YZ$$
2. *Pseudotransitivity rule for multivalued dependencies.*
   $$\{X \twoheadrightarrow Y, WY \twoheadrightarrow Z\} \models WX \twoheadrightarrow (Z - WY)$$
3. *Mixed pseudotransitivity rule.* $\{X \twoheadrightarrow Y, XY \rightarrow Z\} \models X \rightarrow (Z - Y)$.
4. *Decomposition rule for multivalued dependencies.* If $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$ hold, then $X \twoheadrightarrow (Y \cap Z)$, $X \twoheadrightarrow (Y - Z)$, and $X \twoheadrightarrow (Z - Y)$ hold.

We leave the proof that these rules are valid as an exercise; techniques similar to those used for A6 and A8 above will suffice, or we can prove them from axioms A1–A8.

We should note that the decomposition rule for multivalued dependencies is weaker than the corresponding rule for functional dependencies. The latter rule allows us to deduce immediately from $X \rightarrow Y$ that $X \rightarrow A$ for each attribute A in Y. The rule for multivalued dependencies only allows us to conclude $X \twoheadrightarrow A$ from $X \twoheadrightarrow Y$ if we can find some Z such that $X \twoheadrightarrow Z$, and either $Z \cap Y = A$ or $Y - Z = A$.

The Dependency Basis

However, the decomposition rule for multivalued dependencies, along with the union rule, allows us to make the following statement about the sets Y such that $X \twoheadrightarrow Y$ for a given X.

Theorem 7.10: If U is the set of all attributes, then we can partition $U - X$ into sets of attributes $Y_1, \ldots, Y_k$, such that if $Z \subseteq U - X$, then $X \twoheadrightarrow Z$ if and only if Z is the union of some of the Y_i's.

Proof: Start the partition of $U - X$ with all of $U - X$ in one block. Suppose at some point we have partition $W_1, \ldots, W_n$, and $X \twoheadrightarrow W_i$ for $i = 1, 2, \ldots, n$. If $X \twoheadrightarrow Z$, and Z is not the union of some W_i's, replace each W_i such that $W_i \cap Z$ and $W_i - Z$ are both nonempty by $W_i \cap Z$ and $W_i - Z$. By the decomposition rule, $X \twoheadrightarrow (W_i \cap Z)$ and $X \twoheadrightarrow (W_i - Z)$. As we cannot partition a finite set of attributes indefinitely, we shall eventually find that every Z such that $X \twoheadrightarrow Z$ is the union of some blocks of the partition. By the union rule, X multidetermines the union of any set of blocks. □

We call the above sets $Y_1, \ldots, Y_k$ constructed for X from a set of functional and multivalued dependencies D the *dependency basis* for X (with respect to D).

Example 7.19: In Example 7.18 we observed that $C \twoheadrightarrow HR$. Thus, by the complementation rule, $C \twoheadrightarrow TSG$. We also know that $C \rightarrow T$. Thus, by axiom A7, $C \twoheadrightarrow T$. By the decomposition rule, $C \twoheadrightarrow SG$. One can check that no single attribute except T or C itself is multidetermined by C. Thus, the dependency basis for C is $\{T, HR, SG\}$. Intuitively, associated with each course are independent sets of teachers (there is only one), hour-room pairs that tell when and where the course meets, and student-grade pairs, the roll for the course. □

Closures of Functional and Multivalued Dependencies

Given a set of functional and multivalued dependencies D, we would like to find the set D^+ of all functional and multivalued dependencies logically implied by D. We can compute D^+ by starting with D and applying axioms A1–A8 until no more new dependencies can be derived. However, this process can take time that is exponential in the size of D. Often we only want to know whether a particular dependency $X \rightarrow Y$ or $X \twoheadrightarrow Y$ follows from D. For example, Theorem 7.11, below, requires such inferences to find lossless-join decompositions of relation schemes in the presence of multivalued dependencies.

To test whether a multivalued dependency $X \twoheadrightarrow Y$ holds, it suffices to determine the dependency basis of X and see whether $Y - X$ is the union of some sets thereof. For example, referring to Example 7.19, we know that $C \twoheadrightarrow CTSG$, since TSG is the union of T and SG. Also, $C \twoheadrightarrow HRSG$, but $C \twoheadrightarrow TH$ is false, since TH intersects block HR of the dependency basis, yet TH does not include all of HR. In computing the dependency basis of X with respect to D, a theorem of Beeri [1980] tells us it suffices to compute the basis with respect to the set of multivalued dependencies M, where M consists of

1. All multivalued dependencies in D, and
2. For each functional dependency $X \rightarrow Y$ in D, the set of multivalued dependencies $X \twoheadrightarrow A_1, \ldots, X \twoheadrightarrow A_n$, where $Y = A_1 \cdots A_n$, and each A_i is a single attribute.

Another theorem of Beeri [1980] gives us a way to extract the nontrivial functional dependencies from the dependency basis computed according to the set of multivalued dependencies M. It can be shown that if X does not include A, then $X \rightarrow A$ holds if and only if

1. A is a singleton set of the dependency basis for X according to the set of dependencies M, and
2. There is some set of attributes Y, excluding A, such that $Y \rightarrow Z$ is one of the given dependencies of D, and A is in Z.

Furthermore, Beeri [1980] gives the following polynomial time algorithm for computing the dependency basis of X with respect to M. Note that while Theorem 7.10 convinces us that the dependency basis exists, it does not tell us how to find the multivalued dependencies needed to apply the decomposition rule.

Algorithm 7.6: Computing the Dependency Basis.

INPUT: A set of multivalued dependencies M over set of attributes U, and a set $X \subseteq U$.

OUTPUT: The dependency basis for X with respect to M.

METHOD: We start with a collection of sets $\mathcal{S}$, which eventually becomes the dependency basis we desire. Initially, $\mathcal{S}$ consists of only one set, $U - X$; that is, $\mathcal{S} = \{U - X\}$. Until no more changes can be made to $\mathcal{S}$, look for dependencies $V \twoheadrightarrow W$ in M and a set Y in $\mathcal{S}$ such that Y intersects W but not V. Replace Y by $Y \cap W$ and $Y - W$ in $\mathcal{S}$. The final collection of sets $\mathcal{S}$ is the dependency basis for X. □

Since Algorithm 7.6 only causes sets in $\mathcal{S}$ to be split, and it terminates when no more splitting can be done, it is straightforward the algorithm takes time that is polynomial in the size of M and U. In fact, careful implementation allows the algorithm to run in time proportional to the number of dependencies in M times the cube of the number of attributes in U. A proof of this fact and a proof of correctness for Algorithm 7.6 can be found in Beeri [1980].

Lossless Joins

Algorithm 7.2 helps us determine when a decomposition of a relation scheme R into $(R_1, \ldots, R_k)$ has a lossless join, on the assumption that the only dependencies to be satisfied by the relations for R are functional. That algorithm can be generalized to handle multivalued dependencies, as we shall see in the next section. In the case of a decomposition of R into two schemes, there is a simple test for a lossless join.

Theorem 7.11: Let R be a relation scheme and $\rho = (R_1, R_2)$ a decomposition of R. Let D be a set of functional and multivalued dependencies on the attributes of R. Then ρ has a lossless join with respect to D if and only if

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$$

[or equivalently, by the complementation rule, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$].

Proof: Decomposition ρ has a lossless join if and only if for any relation r satisfying D, and any two tuples μ and ν in r, the tuple ϕ such that $\phi[R_1] = \mu[R_1]$ and $\phi[R_2] = \nu[R_2]$ is in r if it exists. That is, ϕ is what we get by joining the projection of μ onto R_1 with the projection of ν onto R_2. But ϕ exists if and only if $\mu[R_1 \cap R_2] = \nu[R_1 \cap R_2]$. Thus, the condition that ϕ is always in r is exactly the condition that

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$$

or equivalently, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$. □

Note that by axiom A7, Theorem 7.5 implies Theorem 7.11 when the only dependencies are functional, but Theorem 7.5 says nothing at all if there are multivalued dependencies that must be satisfied.

7.10 FOURTH NORMAL FORM

There is a generalization of Boyce-Codd normal form, called fourth normal form, that applies to relation schemes with multivalued dependencies. Let R be a relation scheme and D the set of dependencies applicable to R. We say R is in *fourth normal form* (*4NF*) if whenever there is, in D^+, a multivalued dependency $X \twoheadrightarrow Y$, where Y is not a subset of X, and XY does not include all the attributes of R, it is the case that X is a superkey of R. Note that the definitions of "key" and "superkey" have not changed because multivalued dependencies are present; "superkey" still means a set of attributes that functionally determines R.

Observe that if R is in 4NF, then it is in BCNF; i.e., 4NF is a stronger condition than BCNF. In proof, suppose R is not in Boyce-Codd normal form, because there is some functional dependency $X \rightarrow A$, where X is not a superkey, and A is not in X. If $XA = R$, then surely X includes a key. Therefore XA does not include all attributes. By A8, $X \rightarrow A$ implies $X \twoheadrightarrow A$. Since $XA \neq R$ and A is not in X, $X \twoheadrightarrow A$ is a 4NF violation.

We can find a decomposition of R into $\rho = (R_1, \ldots, R_k)$, such that ρ has a lossless join with respect to D, and each R_i is in 4NF, as follows. We start with ρ consisting only of R, and we repeatedly decompose relation schemes when we find a violation of 4NF, as in the discussion of the simple but time-consuming decomposition algorithm for BCNF decomposition preceding Algorithm 7.4. If there is a relation scheme S in ρ that is not in 4NF with respect to D projected onto S,[10] then there must be in S a dependency $X \twoheadrightarrow Y$, where X is not a superkey of S, Y is not empty or a subset of X, and $XY \neq S$. We may assume X and Y are disjoint, since $X \twoheadrightarrow (Y - X)$ follows from $X \twoheadrightarrow Y$ using A1, A7, and the decomposition rule. Then replace S by $S_1 = XY$ and $S_2 = S - Y$, which must be two relation schemes with fewer attributes than S. By Theorem 7.11, since $(S_1 \cap S_2) \twoheadrightarrow (S_1 - S_2)$, the join of S_1 and S_2 is lossless with respect to $\pi_S(D)$, which we take in this section to be the set of functional and multivalued dependencies that follow from D and involve only attributes in the set S.

We leave it as an exercise the generalization of Lemma 7.6 to sets of functional and multivalued dependencies; that is, the repeated decomposition as above produces a set of relation schemes that has a lossless join with respect

[10] We shall discuss later how to find the projection of a set of functional and multivalued dependencies.

to D. The only important detail remaining is to determine how one computes $\pi_S(D)$, given R, D, and $S \subseteq R$. It is a theorem of Aho, Beeri, and Ullman [1979] that $\pi_S(D)$ can be computed as follows.

1. Compute D^+.
2. For each $X \rightarrow Y$ in D^+, if $X \subseteq S$, then $X \rightarrow (Y \cap S)$ holds in S.[11]
3. For each $X \twoheadrightarrow Y$ in D^+, if $X \subseteq S$, then $X \twoheadrightarrow (Y \cap S)$ holds in S.
4. No other functional or multivalued dependencies for S may be deduced from the fact that D holds for R.

Example 7.20: Let us reinvestigate the $CTHRSG$ relation scheme first introduced in Example 7.15. We have several times noted the minimal cover

$$\begin{array}{ll} C \rightarrow T & CS \rightarrow G \\ HR \rightarrow C & HS \rightarrow R \\ HT \rightarrow R & \end{array}$$

for the pertinent functional dependencies. It turns out that one multivalued dependency, $C \twoheadrightarrow HR$, together with the above functional dependencies, allows us to derive all the multivalued dependencies that we would intuitively feel are valid. We saw, for example, that $C \twoheadrightarrow HR$ and $C \rightarrow T$ imply $C \twoheadrightarrow SG$. We also know that $HR \rightarrow CT$, so $HR \twoheadrightarrow CT$. By the complementation rule, $HR \twoheadrightarrow SG$. That is to say, given an hour and room, there is an associated set of student-grade pairs, namely the students enrolled in the course meeting in that room and that hour, paired with the grades they got in that course. The reader is invited to explore further the set of multivalued dependencies following from the given five functional dependencies and one multivalued dependency.

To place relation scheme $CTHRSG$ in 4NF, we might start with

$$C \twoheadrightarrow HR$$

which violates the 4NF conditions since C is not a superkey (SH is the only key for $CTHRSG$). We decompose $CTHRSG$ into CHR and $CTSG$. The relation scheme CHR has key HR. The multivalued dependency $C \twoheadrightarrow HR$ does not violate fourth normal form for CHR, since the left and right sides together include all the attributes of CHR. No other functional or multivalued dependency projected onto CHR violates 4NF, so we need not decompose CHR any further.

Such is not the case for $CTSG$. The only key is CS, yet we see the multivalued dependency $C \twoheadrightarrow T$, which follows from $C \rightarrow T$. We therefore split $CTSG$ into CT and CSG. These are both in 4NF with respect to their projected dependencies, so we have obtained the decomposition $\rho = (CHR, CT, CSG)$, which has a lossless join and all relation schemes in fourth normal form.

[11] Note that since $X \rightarrow Y \cap S$ is also in D^+, this rule is equivalent to the rule for projecting functional dependencies given earlier.

It is interesting to note that when we ignore the multivalued dependency $C \twoheadrightarrow HR$, the decomposition ρ does not necessarily have a lossless join, but if we are allowed to use $C \twoheadrightarrow HR$, it is easy to prove by Theorem 7.11 that the join of these relations is lossless. As an exercise, the reader should find a relation r for $CTHRSG$ such that $m_\rho(r) \neq r$, yet r satisfies all of the given functional dependencies (but not $C \twoheadrightarrow HR$, of course). □

Embedded Multivalued Dependencies

One further complication that enters when we try to decompose a relation scheme R into 4NF is that there may be certain multivalued dependencies that we expect to hold when we project any plausible relation r for R onto a subset $X \subseteq R$, yet we do not expect these dependencies to hold in r itself. Such a dependency is said to be *embedded* in R, and we must be alert, when writing down all the constraints that we believe hold in relations r for R, not to ignore an embedded multivalued dependency. Incidentally, embedded functional dependencies never occur; it is easy to show that if $Y \rightarrow Z$ holds when relation r over R is projected onto X, then $Y \rightarrow Z$ holds in r as well. The same is not true for multivalued dependencies, as the following example shows.

Example 7.21: Suppose we have the attributes C (course), S (student), P (prerequisite), and Y (year in which the student took the prerequisite). The only nontrivial functional or multivalued dependency is $SP \rightarrow Y$, so we may decompose $CSPY$ into CSP and SPY; the resulting schemes are apparently in 4NF.

The multivalued dependency $C \twoheadrightarrow S$ does not hold. For example, we might have in relation r for $CSPY$ the tuples

CS402	Jones	CS311	1988
CS402	Smith	CS401	1989

yet not find the tuple

CS402	Jones	CS401	1989

Presumably Jones took CS401, since it is a prerequisite for CS402, but perhaps he did not take it in 1989. Similarly, $C \twoheadrightarrow P$ does not hold in $CSPY$.

However, if we project any legal relation r for $CSPY$ onto CSP, we would expect $C \twoheadrightarrow S$ and, by the complementation rule, $C \twoheadrightarrow P$ to hold, provided every student enrolled in a course is required to have taken each prerequisite for the course at some time. Thus, $C \twoheadrightarrow S$ and $C \twoheadrightarrow P$ are embedded multivalued dependencies for CSP. As a consequence, CSP is really not in 4NF, and it should be decomposed into CS and CP. This replacement avoids repeating the student name once for each prerequisite of a course in which he is enrolled.

It is interesting to observe that the decomposition $\rho = (CS, CP, SPY)$ has a lossless join if we acknowledge that $C \twoheadrightarrow S$ is an embedded dependency for CSP. For then, given any relation r for $CSPY$ that satisfies $SP \rightarrow Y$ and the dependency $C \twoheadrightarrow S$ in CSP, we can prove that $m_\rho(r) = r$. Yet we could not prove this assuming only the functional dependency $SP \rightarrow Y$; the reader is invited to find a relation r satisfying $SP \rightarrow Y$ (but not the embedded dependency) such that $m_\rho(r) \neq r$. □

We shall consider embedded multivalued dependencies further in the next section. Here let us introduce the standard notation for such dependencies. A relation r over relation scheme R satisfies the embedded multivalued dependency $X \twoheadrightarrow Y \mid Z$ if the multivalued dependency $X \twoheadrightarrow Y$ is satisfied by the relation $\pi_{X \cup Y \cup Z}(r)$, which is the projection of r onto the set of attributes mentioned in the embedded dependency. Note that there is no requirement that X, Y, and Z be disjoint, and by the union, decomposition, and complementation rules, $X \twoheadrightarrow Y$ holds in $\pi_{X \cup Y \cup Z}(r)$ if and only if $X \twoheadrightarrow Z$ does, so $X \twoheadrightarrow Y \mid Z$ means the same as $X \twoheadrightarrow Z \mid Y$. As an example, the embedded multivalued dependency from Example 7.21 is written $C \twoheadrightarrow S \mid P$ or $C \twoheadrightarrow P \mid S$.

7.11 GENERALIZED DEPENDENCIES

In this section we introduce a notation for dependencies that generalizes both functional and multivalued dependencies. Modeling the "real world" does not demand such generality; probably, functional and multivalued dependencies are sufficient in practice.[12] However, there are some key ideas, such as the "chase" algorithm for inferring dependencies, that are better described in the general context to which the ideas apply than in the special case of functional or multivalued dependencies. The ideas associated with generalized dependencies also get used in query optimization, and they help relate dependencies to logical rules (Horn clauses), thereby allowing some of this theory to apply to optimization of logic programs as well.

We view both functional and multivalued dependencies as saying of relations that "if you see a certain pattern, then you must also see this." In the case of functional dependencies, "this" refers to the equality of certain of the symbols seen, while for multivalued dependencies, "this" is another tuple that must also be in the relation. For example, let $U = ABCD$ be our set of attributes. Then the functional dependency $A \rightarrow B$ says that whenever we see, in

[12] Often, one observes *inclusion dependencies*, as well. These are constraints that say a value appearing in one attribute of one relation must also appear in a particular attribute of some other relation. For example, we would demand of the YVCB database that a customer name appearing in the CUST field of an ORDERS tuple also appear in the CNAME field of the CUSTOMERS relation; i.e., each order must have a real customer behind it. The desire to enforce inclusion dependencies explains the mechanics of insertion and deletion in the DBTG proposal (Section 5.3), and the constraints System R places on a pair of relations that are stored "via set" (Section 6.11). As inclusion dependencies do not influence the normalization process, their theory is mentioned only in the exercises.

some relation r, two tuples $ab_1c_1d_1$ and $ab_2c_2d_2$, then $b_1 = b_2$ in those tuples. The multivalued dependency $A \twoheadrightarrow B$ says of the same two tuples that we must also see the tuple $ab_1c_2d_2$ in r, which is a weaker assertion than saying $b_1 = b_2$. A convenient tabular form of such dependencies is shown in Figure 7.8.

a	b_1	c_1	d_1
a	b_2	c_2	d_2
	$b_1 = b_2$		

(a) The functional dependency $A \to B$.

a	b_1	c_1	d_1
a	b_2	c_2	d_2
a	b_1	c_2	d_2

(b) The multivalued dependency $A \twoheadrightarrow B$.

Figure 7.8 Dependencies in tabular notation.

The two dependencies of Figure 7.8 are different in the kind of conclusion they allow us to draw. The functional dependency [Figure 7.8(a)] is called an *equality-generating dependency*, because its conclusion is that two symbols must in fact represent the same symbol. The multivalued dependency [Figure 7.8(b)] is called a *tuple-generating dependency*, because it allows us to infer that a particular tuple is in the relation to which the dependency applies. In the following pages, we wish to allow more than two tuples as hypotheses of dependencies, and we wish to allow various combinations of symbols appearing in their components. The conclusions, though, will continue to be either equalities or new tuples.

Define a *generalized dependency* over a relation scheme $A_1 \cdots A_n$ to be an expression of the form

$$(t_1, \ldots, t_k)/t$$

where the t_i's are n-tuples of symbols, and t is either another n-tuple (in which case we have a tuple-generating dependency) or an expression $x = y$, where x and y are symbols appearing among the t_i's (then we have an equality-generating dependency). We call the t_i's the *hypotheses* and t the *conclusion*. Intuitively, the dependency means that for every relation in which we find the hypotheses, the conclusion holds. To see the hypothesis tuples, we may have to rename some or all of the symbols used in the hypotheses to make them match the symbols used in the relation. Any renaming of symbols that is done applies to the conclusion as well as the hypotheses, and of course it applies to all occurrences of a symbol. We shall give a more formal definition after some examples and discussion.

Frequently we shall display these dependencies as in Figure 7.8, with the hypotheses listed in rows above a line and the conclusion below. It is sometimes useful as well to show the attributes to which the columns correspond, above a line at the top. In all cases, we assume that the order of the attributes in the relation scheme is fixed and understood.

Typed and Typeless Dependencies

Frequently, we find, as in Figure 7.8, that each symbol appearing in a generalized dependency is associated with a unique column. Functional and multivalued dependencies have this property, for example. Such dependencies are called *typed*, because we can associate a "type," i.e., an attribute, with each symbol. Dependencies in which some symbol appears in more than one column are called *typeless*.

Example 7.22: The second part of Example 7.8 showed that given a certain collection of functional dependencies, the decomposition

$$(AD, AB, BE, CDE, AE)$$

is a lossless-join decomposition. What was really shown there was that any relation that satisfies the functional dependencies $A \to C$, $B \to C$, $C \to D$, $DE \to C$, and $CE \to A$ must also satisfy the "join dependency" $\bowtie (AD, AB, BE, CDE, AE)$. In general, a *join dependency* is a typed tuple-generating dependency that says, about a relation r for scheme R, that if we project r onto some set of schemes $R_1, \ldots, R_k$, then take the natural join of the projections, the tuples we get are all in r. We use the notation $\bowtie (R_1, \ldots, R_k)$ for this dependency.

We can write our example join dependency as in Figure 7.9. In that figure we use blanks to denote symbols that appear only once. The reader may have noticed the similarity between the tabular representation of generalized dependencies and the Query-by-Example notation of Sections 4.4 and 4.5; the convention that a blank stands for a symbol that appears nowhere else is borrowed from there.

In general, the join dependency $\bowtie (R_1, \ldots, R_k)$, expressed in the tabular notation, has one hypothesis row for each R_i, and this row has the same symbol as the conclusion row in the columns for the attributes in R_i; elsewhere in that row are symbols, each of which appears nowhere else. The justification is that the join dependency says about a relation r that whenever we have a tuple, such as the conclusion row, that agrees with some tuple μ_i of r in the attributes of R_i for $i = 1, 2, \ldots, k$, then that tuple is itself in r. □

A	B	C	D	E
a			d	
a	b			
	b			e
		c	d	e
a				e
a	b	c	d	e

Figure 7.9 A join dependency in tabular notation.

Full and Embedded Dependencies

We shall not require that a symbol appearing in the conclusion of a tuple-generating dependency also appear in the hypotheses. A symbol of the conclusion appearing nowhere else is called *unique*. A generalized dependency is called *embedded* if it has one or more unique symbols and *full* if it has no unique symbols. This use of the term "embedded" generalizes our use of the term in connection with multivalued dependencies. That is, if a multivalued dependency is embedded within a set of attributes S, it must have unique symbols in all the components not S.

Example 7.23: We could write the embedded multivalued dependency

$$C \twoheadrightarrow S \mid P$$

of Example 7.21 as

C	S	P	Y
c	s_1	p_1	y_1
c	s_2	p_2	y_2
c	s_1	p_2	y_3

Notice that y_3 is a unique symbol. □

As a general rule, we can write any embedded multivalued dependency $X \twoheadrightarrow Y \mid Z$ over a set of attributes U by writing two hypothesis rows that agree in the columns for the attributes in X and disagree in all other attributes. The conclusion agrees with both hypotheses on X, agrees with the first hypothesis on Y, agrees with the second on the attributes in Z, and has a unique symbol everywhere else.

The justification is that the embedded multivalued dependency

$$X \twoheadrightarrow Y \mid Z$$

says that if we have two tuples μ and ν in relation r that project onto

$$X \cup Y \cup Z$$

to give tuples μ' and ν', and $\mu'[X] = \nu'[X]$, then there is some tuple ω in r that projects to ω' and satisfies $\omega'[X] = \mu'[X] = \nu'[X]$, $\omega'[Y] = \mu'[Y]$, and $\omega'[Z] = \nu'[Z]$. Notice that nothing at all is said about the value of ω for attributes in $U - X - Y - Z$. Clearly, we can express all the above in our generalized dependency notation, where μ and ν are the first and second hypotheses, and ω is the conclusion. Since we can only conclude that the tuple ω has some values in the attributes $U - X - Y - Z$, but we cannot relate those values to the values in μ or ν, we must use unique symbols in our conclusion.

One reason for introducing the generalized dependency notation is that it leads to a conceptually simple way to infer dependencies. The test works for full dependencies of all sorts, although it may take exponential time, and therefore, is not preferable to Algorithm 7.1 for inferring functional dependencies from other functional dependencies, or to the method outlined before Algorithm 7.6 (computation of the dependency basis) when only functional and multivalued dependencies are concerned. When there are embedded dependencies, the method may succeed in making the inference, but it may also give an inconclusive result. There is in fact, no known algorithm for testing whether an embedded dependency follows logically from others, even when the dependencies are restricted to an apparently simple class, such as embedded multivalued dependencies.

Generalized Dependencies and Horn Clauses

Notice the similarity between full, tuple-generating dependencies and datalog rules. Since dependencies apply to single relations, the head and all the subgoals of the body have the same predicate symbol, but any datalog rule with no negation and only one predicate symbol can be thought of as a (typeless) tuple-generating dependency. For example, the dependency of Figure 7.8(b) can be written as a rule:

```
r(A,B1,C2,D2) :- r(A,B1,C1,D1) & r(A,B2,C2,D2).
```

We could even view a full equality-generating dependency as a rule with a built-in predicate at the head, and we could make inferences with such rules as with any logical rules. For example, Figure 7.8(a) would appear as

```
B1 = B2 :- r(A,B1,C1,D1) & r(A,B2,C2,D2).
```

However, we should be more careful interpreting embedded dependencies as rules. If we blindly translated the embedded multivalued dependency of Example 7.23 into a rule

```
r(C,S1,P2,Y3) :- r(C,S1,P1,Y1) & r(C,S2,P2,Y2).
```

we would get a rule with a variable, $Y3$, that appears in the head but not in the

body. The correct interpretation of such a rule is that, given values of C, $S1$, and $P2$ that, together with values for the other variables of the body, satisfy the subgoals of the body, the conclusion of the head is true *for all* values of $Y3$. However, the meaning of the embedded dependency is that *there exists* some value of $Y3$ that makes the head true for these values of C, $S1$, and $P2$.

Symbol Mappings

Before giving the inference test for generalized dependencies, we need to introduce an important concept, the *symbol mapping*, which is a function h from one set of symbols S to another set T; that is, for each symbol a in S, $h(a)$ is a symbol in T. We allow $h(a)$ and $h(b)$ to be the same member of T, even if $a \neq b$.

If $\mu = a_1a_2\cdots a_n$ is a tuple whose symbols are in S, we may apply the symbol mapping h to μ and obtain the tuple $h(\mu) = h(a_1)h(a_2)\cdots h(a_n)$. If $\{\mu_1, \ldots, \mu_k\}$ is a set of tuples whose symbols are in S, and $\{\nu_1, \ldots, \nu_m\}$ are tuples whose symbols are in T, we say there is a symbol mapping from the first set of tuples to the second if there is some h such that for all $i = 1, 2, \ldots, k$, $h(\mu_i)$ is ν_j for some j. It is possible that two or more μ_i's are mapped to the same ν_j, and some ν_j's may be the target of no μ_i.

Example 7.24: Let $A = \{abc, ade, fbe\}$ and $B = \{xyz, wyz\}$. There are several symbol mappings from A to B. One has $h(a) = h(f) = x$, $h(b) = h(d) = y$, and $h(c) = h(e) = z$. Thus, h maps all three tuples in A to xyz. Another symbol mapping has $g(a) = x$, $g(b) = g(d) = y$, $g(c) = g(e) = z$, and $g(f) = w$. Symbol mapping g sends abc and ade to xyz, but sends fbe to wyz. □

Our most important use for symbol mappings is as maps between sets of rows as in Example 7.24. The reader should observe a duality that holds in that situation. We defined symbol mappings as functions on symbols, and when applied to sets of rows, we added the requirement that the mapping applied to each row of the first set is a row of the second set. Dually, we could have defined mappings from rows to rows, and added the requirement that no symbol be mapped by two different rows to different symbols. Thus, in Example 7.24, we could not map abc to xyz and also map ade to wyz, because a would be mapped to both x and w.

Formal Definition of Generalized Dependency

With the notion of a symbol mapping, we can formally define the meaning of generalized dependencies. We say a relation r *satisfies* the tuple-generating dependency $(t_1, \ldots, t_n)/t$ if whenever h is a symbol mapping from all the hypotheses $\{t_1, \ldots, t_n\}$ to r, we can extend h to any unique symbols in t in such a way that $h(t)$ is in r. We also say that r *satisfies* the equality-generating dependency $(t_1, \ldots, t_n)/a = b$ if whenever h is a symbol mapping from the hypotheses to r, it must be that $h(a) = h(b)$.

Example 7.25: Let d be the generalized dependency in Figure 7.10(a), and let r be the relation of Figure 7.10(b). Notice that d is not the same as the multivalued dependency $A \twoheadrightarrow B$, since the symbol a_2, which is a unique symbol in Figure 7.10(a), would have to be a_1 instead. In fact, Figure 7.10(a) is an example of a two-hypothesis tuple-generating dependency that is neither a full nor embedded multivalued dependency; such dependencies were called *subset dependencies* by Sagiv and Walecka [1982].

a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_2

(a) The dependency d.

A	B	C
0	1	2
0	3	4
0	3	2
5	1	4

(b) The relation r.

Figure 7.10 A generalized dependency and a relation satisfying it.

To see that r satisfies d, let us consider a symbol mapping h and the tuples of r to which each of the hypotheses of d could be mapped. Since the two hypotheses agree in the A-column, and $h(a_1)$ can have only one value, we know that either both hypotheses are mapped to the last tuple of r [if $h(a_1) = 5$], or both are mapped among the first three tuples [if $h(a_1) = 0$]. In the first case, h maps b_1 and b_2 to 1 and c_1 and c_2 to 4. Then we can extend h to the unique symbol a_2 by defining $h(a_2) = 5$. In that case, $h(a_2b_1c_2) = 514$, which is a member of r, so we obtain no violation of d with mappings that have $h(a_1) = 5$.

Now consider what happens if $h(a_1) = 0$, so the only possible mappings send the two hypotheses into the first three tuples of r. Any such mapping h has $h(b_1)$ equal to either 1 or 3, and it has $h(c_2)$ equal to either 2 or 4. In any of the four combinations, there is a tuple in r that has that combination of values in its B and C components. Thus, we can extend h to the unique symbol a_2 by setting $h(a_2) = 5$ if $h(b_1) = 1$ and $h(c_2) = 4$, and setting $h(a_2) = 0$ otherwise.

We have now considered all symbol mappings that map each of the hypotheses of d into a tuple of r, and have found that in each case, we can extend

the mapping to the unique symbol a_2 in such a way that the conclusion of d is present in r. Therefore, r satisfies d. □

Applying Dependencies to Relations

Suppose we have an equality-generating dependency

$$d = (s_1, \ldots, s_k)/a = b$$

and a relation $r = \{\mu_1, \ldots, \mu_m\}$. We can *apply* d to r if we find a symbol mapping h from $\{s_1, \ldots, s_k\}$ to $\{\mu_1, \ldots, \mu_m\}$. The effect of applying d to r using symbol mapping h is to equate the symbols $h(a)$ and $h(b)$ wherever they appear among the μ_i's; either may replace the other.

If we have a tuple-generating dependency instead, say $e = (s_1, \ldots, s_k)/s$, we *apply* e to r using h by adjoining to r the tuple $h(s)$. However, if e is an embedded dependency, then s will have one or more unique symbols, so h will not be defined for all symbols of s. In that case, if c is a unique symbol in s, create a new symbol, one that appears nowhere else in r, and extend h by defining $h(c)$ to be that symbol. Of course, we create distinct symbols for each of the unique symbols of s.

It may be possible, however, the unique symbols can all be replaced by existing symbols of r so that $h(s)$ becomes a member of r. In that case, the requirement that $h(s)$ be in r is already satisfied, and we have the option (which we should take, because it simplifies matters) of not changing r at all.

Example 7.26: Let us consider the equality-generating dependency

$$(abc, ade, fbe)/a = f$$

applied to the relation $r = \{xyz, wyz\}$. If we use the symbol mapping g of Example 7.24, we find that $g(a) = x$ and $g(f) = w$. We apply the dependency using this symbol mapping, by equating x and w; say we replace them both by x. Then the effect on r of applying the dependency in this way is to change r into $\{xyz\}$.

Suppose instead we had the tuple-generating dependency

$$(abc, ade, fbe)/abq$$

Then using the same symbol mapping, we would adjoin to r a tuple whose first two components were $g(a) = x$ and $g(b) = y$ and whose third component was a new symbol, not appearing in r, say u; that is, r becomes $\{xyz, wyz, xyu\}$. However, we could replace u by the existing symbol z, and the result would be xyz, a tuple already in r. Thus, we have the preferred option of leaving r unchanged. □

The Chase Algorithm for Inference of Dependencies

Now we can exhibit a process that helps us resolve the question whether $D \models d$,

where D is a set of generalized dependencies, and d is another generalized dependency. The procedure is an algorithm when D has full dependencies only. However, if D has some embedded dependencies, it tells the truth if it answers at all, but it may run on forever inconclusively. We call the process the *chase*, because we "chase down" all the consequences of the dependencies D.

The intuitive idea behind the chase process is that we start with the hypotheses of the dependency d we wish to test, and we treat them as if they formed a relation. We then apply the given dependencies D to this relation. If we obtain a tuple that is the conclusion of d, then we have a proof that d follows from D. The reason this test works is that, generalizing Algorithm 7.4, should we fail to draw the desired conclusion, the relation that results when we finish the process is a counterexample; it satisfies D but not d.[13]

First suppose that d is a tuple-generating dependency $(t_1, \ldots, t_m)/t$. We begin with the relation $r = \{t_1, \ldots, t_m\}$. We then apply all of the dependencies in D, in any order, repeatedly, until either

1. We cannot apply the dependencies in any way that changes r, or
2. We discover in r a tuple that agrees with t on all components except, perhaps, those places where t has a unique symbol.

However, when applying an equality-generating dependency, if one of the symbols being equated appears in t, change the other symbol to that one.

In case (2) above, we conclude that $D \models d$ is true. If (1) holds, but not (2), then we say that $D \models d$ is false. In fact, the resulting r will be a counterexample. To see why, first notice that r satisfies all dependencies in D (or else one of them could be applied).

Second, we must show that r does not satisfy d. In proof, note that as we apply equality-generating dependencies to r, the symbols in the original rows $t_1, \ldots, t_m$ may change, but there is always a symbol mapping h that sends each symbol of the hypothesis rows to what that symbol has become after these equalities have been performed. Then $h(t_i)$ is in the final relation r for each i. When we equate symbols, we do not change symbols in t, so $h(a) = a$ for any symbol a that appears in t, except for the unique symbols of t, for which h is not defined. Thus, if d holds in r, that relation must have some tuple that agrees with t on all but the unique symbols. However, (2) was assumed false, so there can be no such tuple in r at the end of the chase process. Thus, r does not satisfy d, and therefore serves as a counterexample to $D \models d$.

Now, let us consider the converse, why the implication holds whenever case (2) applies. Recall the proof of Theorem 7.4, which is really a special case of

[13] However, there is the problem that if some dependencies are embedded, the process may not stop. In principle, it generates an infinite relation, and that infinite relation forms a counterexample. Unfortunately, with embedded dependencies we cannot tell, as we work, whether the process will go on forever or not, so the "test" is sometimes inconclusive.

our present claim. That is, Algorithm 7.2, the lossless join test, can now be seen as a use of the chase process to test whether $F \models j$, where j is the join dependency made from the decomposition to which Algorithm 7.2 is applied, that is $\bowtie (R_1, \ldots, R_k)$. As in Theorem 7.4, we can see the relation r used in the chase as saying that certain tuples are in a hypothetical relation that satisfies D.

Initially, these tuples are the hypotheses $\{t_1, \ldots, t_m\}$ of the dependency being tested. Each time we apply a dependency, we are making an inference about other tuples that must be in this hypothetical relation (if we use a tuple-generating dependency), or about two symbols that must be equal (if we use an equality-generating dependency). Thus, each application is a valid inference from D, and if we infer the presence of t, that too is valid, i.e., we have shown that any relation containing $t_1, \ldots, t_m$ also contains t (or a tuple that agrees with t on nonunique symbols).

However, the dependency d says more than that a relation that contains the exact tuples $\{t_1, \ldots, t_m\}$ also contains t. It says that if any relation whatsoever contains the tuples formed by some symbol mapping h of the t_i's, then h can be extended to the unique symbols of t, and $h(t)$ will also be in the relation. We can show this more general statement by following the sequence of applications of dependencies in D during the chase. That is, start with $\{h(t_1), \ldots, h(t_m)\}$ and apply the same sequence of dependencies from D by composing the symbol mapping used to apply each dependency, with the symbol mapping h, to get another symbol mapping. The result will be the image, under h, of the sequence of changes made to the original relation $r = \{t_1, \ldots, t_m\}$.

We must also explain how to test, using the chase process, whether an equality-generating dependency $(t_1, \ldots, t_m)/a = b$ follows from a set of dependencies D. Follow the same process, but end and say yes if we ever equate the symbols a and b; say no as for tuple-generating dependencies, if we can make no more changes to r, yet we have not equated a and b. The validity of the inferences follows in essentially the same way as for tuple-generating dependencies.

We can sum up our claim in the following theorem.

Theorem 7.12: The chase process applied to a set of full generalized dependencies D and a (possibly embedded) generalized dependency d determines correctly whether $D \models d$.

Proof: Above, we argued informally why the procedure, if it makes an answer at all, answers correctly. We shall not go into further detail; Maier, Mendelzon, and Sagiv [1979] contains a complete proof of the result.

We must, however, show that if D has only full dependencies, then the process is an algorithm; that is, it always halts. The observation is a simple one. When we apply a full dependency, we introduce no new symbols. Thus,

the relation r only has tuples composed of the original symbols of the hypotheses of d. But there are only a finite number of such symbols, and therefore r is always a subset of some finite set. We have only to rule out the possibility that r exhibits an oscillatory behavior; that is, it assumes after successive applications of dependencies, a sequence of values

$$r_1, r_2, \ldots, r_n = r_1, r_2 \cdots$$

Tuple-generating dependencies always make the size of r increase, while equality-generating dependencies either leave the size the same or decrease it. Thus, the cycle must contain at least one equality-generating dependency. But here, an equality of symbols permanently reduces the number of different symbols, since only the application of an embedded dependency could increase the number of different symbols in r, and D was assumed to contain full dependencies only. Thus no cycle could involve an equality-generating dependency and full tuple-generating dependencies only, proving that no cycle exists. We conclude that either we reach a condition where no change to r is possible, or we discover that the conclusion of d is in r. □

a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_1	c_2	d_3

(a) $A \twoheadrightarrow B \mid C$

a_2	b_3	c_3	d_4
a_3	b_3	c_4	d_5

$d_4 = d_5$

(b) $B \rightarrow D$

a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_6	c_5	d_7

(c) $A \twoheadrightarrow C \mid D$

Figure 7.11 Example dependencies.

Example 7.27: Example 7.8 was really an application of the chase algorithm to make the inferences $\{S \rightarrow A, SI \rightarrow P\} \models \bowtie (SA, SIP)$ and

$$\{A \to C,\ B \to C,\ C \to D,\ DE \to C,\ CE \to A\} \models \bowtie (AD, AB, BE, CDE, AE)$$

As another example, we can show that over the set of attributes $ABCD$

$$\{A \twoheadrightarrow B \mid C,\ B \to D\} \models A \twoheadrightarrow C \mid D$$

We can write the three dependencies involved in tabular notation, as in Figure 7.11.

We begin with the hypotheses of Figure 7.11(c), as shown in Figure 7.12(a). We can apply the dependency of Figure 7.11(a) by using the symbol mapping $h(a_1) = a_4$, $h(b_1) = b_5$, $h(c_1) = c_6$, $h(d_1) = d_7$, $h(b_2) = b_4$, $h(c_2) = c_5$, and $h(d_2) = d_6$. This mapping sends the two hypothesis rows of Figure 7.11(a) to the two rows of Figure 7.12(a), in the opposite order. If we extend h to map d_3 to a new symbol, say d_8, then we can infer that the tuple $a_4 b_5 c_5 d_8$ is in r, as shown in Figure 7.12(b). Then, we can apply the dependency of Figure 7.11(b), using a symbol mapping that the reader can deduce, to map the two hypotheses of Figure 7.11(b) to the second and third rows of Figure 7.12(b) and prove that $d_7 = d_8$. The substitution of d_7 for d_8 is reflected in Figure 7.12(c). The third tuple in Figure 7.12(c) agrees with the conclusion of Figure 7.11(c), except in the B-column, where the latter has a unique symbol, b_6. We conclude that the inference is valid. □

a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7

(a) Initial relation.

a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_5	c_5	d_8

(b) After applying Figure 7.11(a).

a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_5	c_5	d_7

(c) After applying Figure 7.11(b).

Figure 7.12 Sequence of relations constructed by the chase.

EXERCISES

7.1: Suppose we have a database for an investment firm, consisting of the following attributes: B (broker), O (office of a broker), I (investor), S (stock), Q (quantity of stock owned by an investor), and D (dividend paid by a stock), with the following functional dependencies: $S \to D$, $I \to B$, $IS \to Q$, and $B \to O$.

a) Find a key for the relation scheme $R = BOSQID$.
b) How many keys does relation scheme R have? Prove your answer.
c) Find a lossless join decomposition of R into Boyce-Codd normal form.
d) Find a decomposition of R into third normal form, having a lossless join and preserving dependencies.

7.2: Suppose we choose to represent the relation scheme R of Exercise 7.1 by the two schemes $ISQD$ and IBO. What redundancies and anomalies do you forsee?

7.3: Suppose we instead represent R by SD, IB, ISQ, and BO. Does this decomposition have a lossless join?

7.4: Suppose we represent R of Exercise 7.1 by ISQ, IB, SD, and ISO. Find minimal covers for the dependencies (from Exercise 7.1) projected onto each of these relation schemes. Find a minimal cover for the union of the projected dependencies. Does this decomposition preserve dependencies?

7.5: In the database of Exercise 7.1, replace the functional dependency $S \to D$ by the multivalued dependency $S \twoheadrightarrow D$. That is, D now represents the dividend "history" of the stock.

a) Find the dependency basis of I.
b) Find the dependency basis of BS
c) Find a fourth normal form decomposition of R.

7.6: Consider a database of ship voyages with the following attributes: S (ship name), T (type of ship), V (voyage identifier), C (cargo carried by one ship on one voyage), P (port), and D (day). We assume that a voyage consists of a sequence of events where one ship picks up a single cargo, and delivers it to a sequence of ports. A ship can visit only one port in a single day. Thus, the following functional dependencies may be assumed: $S \to T$, $V \to SC$, and $SD \to PV$.

a) Find a lossless-join decomposition into BCNF.
b) Find a lossless-join, dependency-preserving decomposition into 3NF.
* c) Explain why there is no lossless-join, dependency-preserving BNCF decomposition for this database.

7.7: Let U be a set of attributes and D a set of dependencies (of any type) on the attributes of U. Define $\text{SAT}(D)$ to be the set of relations r over U such that r satisfies each dependency in D. Show the following.

a) $\text{SAT}(D_1 \cup D_2) = \text{SAT}(D_1) \cap \text{SAT}(D_2)$.

b) If D_1 logically implies all the dependencies in D_2, then

$$\text{SAT}(D_1) \supseteq \text{SAT}(D_2)$$

7.8: Complete the proof of Lemma 7.1; i.e., show that the transitivity axiom for functional dependencies is sound.

7.9: Complete the proof of Theorem 7.2 by showing statement $(*)$:

$$\text{If } X_1 \subseteq X_2 \text{ then } X_1^{(j)} \subseteq X_2^{(j)} \text{ for all } j$$

7.10: Let F be a set of functional dependencies.

a) Show that $X \to A$ in F is redundant if and only if X^+ contains A, when the closure is computed with respect to $F - \{X \to A\}$.

b) Show that attribute B in the left side X of a functional dependency $X \to A$ is redundant if and only if A is in $(X - \{B\})^+$, when the closure is taken with respect to F.

* 7.11: Show that singleton left sides are insufficient for functional dependencies. That is, show there is some functional dependency that is not equivalent to any set of functional dependencies $\{A_1 \to B_1, \ldots, A_k \to B_k\}$, where the A's and B's are single attributes.

* 7.12: Develop the theory of functional dependencies with single attributes on the left and right sides (call them *SAFD*'s). That is:

a) Give a set of axioms for SAFD's; show that your axioms are sound and complete.

b) Give an algorithm for deciding whether a set of SAFD's implies another SAFD.

c) Give an algorithm to test whether two sets of SAFD's are equivalent.

d) SAFD's look like a familiar mathematical model. Which?

* 7.13: In Theorem 7.3 we used two transformations on sets of functional dependencies to obtain a minimal cover:

i) Eliminate a redundant dependency.

ii) Eliminate a redundant attribute from a left side.

Show the following:

a) If we first apply (*ii*) until no more applications are possible and then apply (*i*) until no more applications are possible, we always obtain a minimal cover.

b) If we apply first (*i*) until no longer possible, then apply (*ii*) until no longer possible, we do not necessarily reach a minimal cover.

7.14: A relation scheme R is said to be in *second normal form* if whenever $X \to A$ is a dependency that holds in R, and A is not in X, then either A is prime or X is not a proper subset of any key (the possibility that X is neither a subset nor a superset of any key is not ruled out by second normal form). Show that the relation scheme $SAIP$ from Example 7.14 violates second normal form.

7.15: Show that if a relation scheme is in third normal form, then it is in second normal form.

7.16: Consider the relation scheme with attributes S (store), D (department), I (item), and M (manager), with functional dependencies $SI \to D$ and $SD \to M$.

a) Find all keys for $SDIM$.

b) Show that $SDIM$ is in second normal form but not third normal form.

* 7.17: Give an $O(n)$ algorithm for computing X^+, where X is a set of at most n attributes, with respect to a set of functional dependencies that require no more than n characters, when written down.

* 7.18: Complete the proof of Theorem 7.5 by providing a formal proof that in the row for R_1, an a is entered if and only if $R_1 \cap R_2 \to A$.

7.19: Complete the proof of Lemma 7.5 by showing that if $r \subseteq s$ then

$$\pi_{R_i}(r) \subseteq \pi_{R_i}(s)$$

7.20: In Example 7.10 we contended that $Z \to C$ does not imply $CS \to Z$. Prove this contention.

7.21: At the end of Section 7.5 it was claimed that $\rho = (AB, CD)$ was a dependency-preserving, but not lossless-join decomposition of $ABCD$, given the dependencies $A \to B$ and $C \to D$. Verify this claim.

7.22: Let $F = \{AB \to C,\ A \to D,\ BD \to C\}$.

a) Find a minimal cover for F.

b) Give a 3NF, dependency-preserving decomposition of $ABCD$ into only two schemes (with respect to the set of functional dependencies F).

c) What are the projected dependencies for each of your schemes?

d) Does your answer to (a) have a lossless join? If not, how could you modify the database scheme to have a lossless join and still preserve dependencies?

7.23: Let $F = \{AB \to C,\ A \to B\}$.

a) Find a minimal cover for F.

b) When (a) was given on an exam at a large western university, more than half the class answered $G = \{A \to B,\ B \to C\}$. Show that answer is wrong by giving a relation that satisfies F but violates G.

7.24: Suppose we are given relation scheme $ABCD$ with functional dependencies $\{A \to B,\ B \to C,\ A \to D,\ D \to C\}$. Let ρ be the decomposition (AB, AC, BD).

a) Find the projected dependencies for each of the relation schemes of ρ.

b) Does ρ have a lossless join with respect to the given dependencies?

c) Does ρ preserve the given dependencies?

7.25: Show that (AB, ACD, BCD) is not a lossless-join decomposition of $ABCD$ with respect to the functional dependencies $\{A \to C,\ D \to C,\ BD \to A\}$.

7.26: Consider the relation scheme $ABCD$ with dependencies

$$F = \{A \to B,\ B \to C,\ D \to B\}$$

We wish to find a lossless-join decomposition into BCNF.

a) Suppose we choose, as our first step, to decompose $ABCD$ into ACD and BD. What are the projected dependencies in these two schemes?

b) Are these schemes in BNCF? If not, what further decomposition is necessary?

7.27: For different sets of assumed dependencies, the decomposition

$$\rho = (AB, BC, CD)$$

may or may not have a lossless join. For each of the following sets of dependencies, either prove the join is lossless or give a counterexample relation to show it is not.

a) $\{A \to B,\ B \to C\}$.

b) $\{B \to C,\ C \to D\}$.

c) $\{B \twoheadrightarrow C\}$.

* 7.28: At most how many passes does Algorithm 7.3 (the test for dependency-preservation) need if F is a set of n functional dependencies over m attributes (an order-of-magnitude estimate is sufficient).

* 7.29: Let F be a set of functional dependencies with singleton right sides.

a) Show that if a relation scheme R has a BCNF violation $X \to A$, where $X \to A$ is in F^+, then there is some $Y \to B$ in F itself such that $Y \to B$ is a BCNF violation for R.

b) Show the same for third normal form.

7.30: Show the following observation, which is needed in Theorem 7.8. If R is a relation scheme, and $X \subseteq R$ is a key for R with respect to set of functional dependencies F, then X cannot have a 3NF violation with respect to the set of dependencies $\pi_X(F)$.

7.31: Prove that there is no such thing as an "embedded functional dependency." That is, if $S \subseteq R$, and $X \to Y$ holds in $\pi_S(R)$, then $X \to Y$ holds in R.

* 7.32: Complete the proof of Theorem 7.9 by showing that axioms A1–A8 are sound and complete. *Hint*: The completeness proof follows Theorem 7.1. To find a counterexample relation for $X \twoheadrightarrow Y$, we generally need more than a two-tuple relation as was used for functional dependencies; the relation could have 2^b tuples, if b is the number of blocks in the dependency basis for X.

* 7.33: Verify the union, pseudotransitivity, and decomposition rules for multivalued dependencies.

* 7.34: Verify the contention in Example 7.21, that there is a relation r satisfying $SP \to Y$, such that $\pi_{CS}(r) \bowtie \pi_{CP}(r) \bowtie \pi_{SPY}(r) \neq r$. Check that your relation does not satisfy $C \twoheadrightarrow S \mid P$.

7.35: Given the dependencies $\{A \twoheadrightarrow B,\ C \to B\}$, what other nontrivial multivalued and functional dependencies hold over the set of attributes ABC?

* 7.36: Prove that in $ABCD$ we can infer $A \twoheadrightarrow D$ from $\{A \twoheadrightarrow B,\ A \to C\}$ in each of the following ways.

a) Directly from the definitions of functional and multivalued dependencies.

b) From axioms A1–A8.

c) By converting to generalized dependencies and "chasing."

* 7.37: Near the beginning of Section 7.10 we claimed that we could project a set of multivalued and functional dependencies D onto a set of attributes S by the following rules (somewhat restated).

i) $X \to Y$ is in $\pi_S(D)$ if and only if $XY \subseteq S$ and $X \to Y$ is in D^+.

ii) $X \twoheadrightarrow Y$ is in $\pi_S(D)$ if and only if $X \subseteq S$, and there is some multivalued dependency $X \twoheadrightarrow Z$ in D^+, such that $Y = Z \cap S$.

Prove this contention.

7.38: Show that the decomposition (CHR, CT, CSG) obtained in Example 7.20 is not lossless with respect the the given functional dependencies only; i.e., the multivalued dependency $C \twoheadrightarrow HR$ is essential to prove the lossless join.

7.39: Use the chase algorithm to tell whether the following inferences are valid over the set of attributes $ABCD$.

a) $\{A \twoheadrightarrow B,\ A \rightarrow C\} \models A \twoheadrightarrow D$

b) $\{A \twoheadrightarrow B \mid C,\ B \twoheadrightarrow C \mid D\} \models A \twoheadrightarrow C \mid D$

c) $\{A \twoheadrightarrow B \mid C,\ A \rightarrow D\} \models A \twoheadrightarrow C \mid D$

** d) $\{A \twoheadrightarrow B \mid C,\ A \twoheadrightarrow C \mid D\} \models A \twoheadrightarrow B \mid D$

* 7.40: Show that no collection of tuple-generating dependencies can imply an equality-generating dependency.

7.41: State an algorithm to determine, given a collection of functional, (full) multivalued, and (full) join dependencies, whether a given decomposition has a lossless join.

7.42: Show that the multivalued dependency $X \twoheadrightarrow Y$ over the set of attributes U is equivalent to the join dependency $\bowtie (XY, XZ)$, where $Z = U - X - Y$. *Hint*: Write both as generalized dependencies.

7.43: What symbol mapping explains the application of Figure 7.11(b) to Figure 7.12(b) to deduce Figure 7.12(c)?

* 7.44: Show that Theorem 7.11, stated for functional and multivalued dependencies, really holds for arbitrary generalized dependencies. That is, (R_1, R_2) has a lossless join with respect to a set of generalized dependencies D if and only if $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$.

* 7.45: Show that if decomposition $\rho = (R_1, \ldots, R_k)$ has a lossless join with respect to a set of generalized dependencies D, then the decomposition $(R_1, \ldots, R_k, S)$ also has a lossless join with respect to D, where S is an arbitrary relation scheme over the same set of attributes as ρ.

* 7.46 Show that it is $\mathcal{NP}$-hard ($\mathcal{NP}$-complete or harder—see Garey and Johnson [1979]) to determine:

a) Given a relation scheme R and a set of functional dependencies F on the attributes of R, whether R has a key of size k or less with respect to F?

b) Given R and F as in (a), and given a subset $S \subseteq R$, is S in ~~BNCF~~ BCNF with respect to F?

c) Whether a given set of multivalued dependencies implies a given join dependency.

* 7.47: A *unary inclusion dependency* $A \subseteq B$, where A and B are attributes (perhaps from different relations) says that in any legal values of the relation(s), every value that appears in the column for A also appears in the column for B. Show that the following axioms

i) $A \subseteq A$ for all A.

ii) If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$.

Are sound and complete for unary inclusion dependencies.

* 7.48: Suppose for some even n we have attributes $A_1, \ldots, A_n$. Also suppose that $A_i \subseteq A_{i+1}$ for odd i, that is, $i = 1, 3, \ldots, n-1$. Finally, suppose that for $i = 3, 5, \ldots, n-1$ we have $A_i \rightarrow A_{i-1}$, and we have $A_1 \rightarrow A_n$.

a) Show that if relations are assumed to be finite, then all the above dependencies can be reversed; that is,

$$A_2 \subseteq A_1,\ A_2 \rightarrow A_3,\ A_4 \subseteq A_3,\ A_4 \rightarrow A_5, \ldots, A_n \subseteq A_{n-1},\ A_n \rightarrow A_1$$

b) Show that there are infinite relations for which (a) does not hold; that is, they satisfy all the given dependencies but not of their reverses.

* 7.49 Show that if D is a set of functional dependencies only, then a relation R is in BCNF with respect to D if and only if R is in 4NF with respect to D.

* 7.50 Show that if $X \rightarrow A_1, \ldots, X \rightarrow A_n$ are functional dependencies in a minimal cover, then the scheme $XA_1 \cdots A_n$ is in 3NF.

BIBLIOGRAPHIC NOTES

Maier [1983] is a text devoted to relational database theory, and provides a more detailed treatment of many of the subjects covered in this chapter. Fagin and Vardi [1986] and Vardi [1988] are surveys giving additional details in the area of dependency theory. Beeri, Bernstein, and Goodman [1978] is an early survey of the theory that provided the motivation for the area.

Functional Dependencies

Functional dependencies were introduced by Codd [1970]. Axioms for functional dependencies were first given by Armstrong [1974]; the particular set of axioms used here (called "Armstrong's axioms") is actually from Beeri, Fagin, and Howard [1977]. Algorithm 7.1, the computation of the closure of a set of attributes, is from Bernstein [1976].

Lossless-Join Decomposition

Algorithm 7.2, the lossless join test for schemes with functional dependencies, is from Aho, Beeri, and Ullman [1979]. The special case of the join of two relations, Theorem 7.5, was shown in the "if" direction by Heath [1971] and Delobel and Casey [1972] and in the opposite direction by Rissanen [1977].

Liu and Demers [1980] provide a more efficient lossless join test for schemes with functional dependencies. Testing lossless joins is equivalent to inferring a join dependency, so the remarks below about inference of generalized dependencies are relevant to lossless-join testing.

Dependency-Preserving Decomposition

Algorithm 7.3, the test for preservation of dependencies, is by Beeri and Honeyman [1981].

The paper by Ginsburg and Zaiddan [1982] points out that when projected, functional dependencies imply certain other dependencies, which happen to be equality-generating, generalized dependencies, but are not themselves functional. As a result, when we discuss projected dependencies, we must be very careful to establish the class of dependencies about which we speak.

Graham and Yannakakis [1984] discuss "independence," a condition on a decomposition that allows satisfaction of dependencies to be checked in the individual relations of a decomposition.

Gottlob [1987] gives an algorithm to compute a cover for $\pi_R(F)$ directly from F; that is, it is not necessary to compute F^+ first. However, the algorithm is not guaranteed to run in polynomial time.

Normal Forms and Decomposition

Third normal form is defined in Codd [1970] and Boyce-Codd normal form in Codd [1972a]. The definitions of first and second normal forms are also found in these papers.

The dependency-preserving decomposition into third normal form, Algorithm 7.5, is from Bernstein [1976], although he uses a "synthetic" approach, designing a scheme without starting with a universal relation. Theorem 7.3, the minimal cover theorem used in Algorithm 7.5, is also from Bernstein [1976]; more restrictive forms of cover are found in Maier [1980, 1983].

The lossless-join decomposition into BCNF given in Algorithm 7.4 is from Tsou and Fischer [1982]. Theorem 7.8, giving a 3NF decomposition with a lossless join and dependency preservation, is from Biskup, Dayal, and Bernstein [1979]. A related result appears in Osborn [1977].

The equivalence problem for decompositions of a given relation was solved by Beeri, Mendelzon, Sagiv, and Ullman [1981]. Ling, Tompa, and Kameda [1981] generalize the notion of third normal form to account for redundancies across several different relation schemes.

Schkolnick and Sorenson [1981] consider the positive and negative consequences of normalizing relation schemes.

Additional Properties of Decompositions

The problem of adequacy of a decomposition has been considered from several points of view. Arora and Carlson [1978] regard the lossless-join and dependency-preservation conditions as a notion of adequacy, while Rissanen [1977] defines a decomposition to have *independent components* if there is a one-to-one correspondence between relations for the universal scheme that satisfy the dependencies, and projections of relations that satisfy the projected dependencies. Maier, Mendelzon, Sadri, and Ullman [1980] show that these notions are equivalent for functional dependencies, but not for multivalued dependencies.

Honeyman [1983] offers an appropriate definition for what it means for a decomposition (database scheme) to satisfy a functional dependency. Graham, Mendelzon, and Vardi [1986] discuss the extension of this question to generalized dependencies.

Recognizing Normalized Relations

Osborn [1979] gives a polynomial-time algorithm to tell whether a given relation scheme R is in BCNF, with respect to a given set of dependencies F over R.[14] In contrast, Jou and Fischer [1983] show that telling whether R is in third normal form with respect to F is $\mathcal{NP}$-complete.

Multivalued Dependencies

Multivalued dependencies were discovered independently by Fagin [1977], Delobel [1978], and Zaniolo [1976] (see also Zaniolo and Melkanoff [1981]), although the earliest manifestation of the concept is in Delobel's thesis in 1973.

The axioms for multivalued dependencies are from Beeri, Fagin, and Howard [1977]. The independence of subsets of these axioms was considered by Mendelzon [1979], while Biskup [1980] shows that if one does not assume a fixed set of attributes, then this set minus the complementation axiom forms a sound and complete set. Lien [1979] develops axioms for multivalued dependencies on the assumption that null values are permitted.

Sagiv et al. [1981] show the equivalence of multivalued dependency theory to a fragment of propositional calculus, thus providing a convenient notation in which to reason about such dependencies.

The dependency basis and Algorithm 7.6 are from Beeri [1980]. Hagihara et al. [1979] give a more efficient test whether a given multivalued dependency is implied by others, and Galil [1982] gives an even faster way to compute the dependency basis.

Embedded multivalued dependencies were considered by Fagin [1977], Delobel [1978] and Tanaka, Kambayashi, and Yajima [1979].

More Normal Forms

Fourth normal form was introduced in Fagin [1977]. In Fagin [1981] we find an "ultimate" normal form theorem; it is possible to decompose relation schemes so

[14] The reader should not be confused between this result and Exercise 7.46(b). The latter indicates that telling whether a relation scheme R is in BCNF given a set of functional dependencies, *defined on a superset of* R, is $\mathcal{NP}$-complete.

that the only dependencies remaining are functional dependencies of a nonkey attribute on a key and constraints that reflect the limited sizes of domains for attributes.

Join Dependencies

Join dependencies were first formalized by Rissanen [1979]. The condition on relations corresponding to a join dependency on their schemes was considered by Nicolas [1978] and Mendelzon and Maier [1979].

A sound and complete axiomatization for a class slightly more general than join dependencies is found in Sciore [1982].

Generalized Dependencies

The notion of generalized dependencies was discovered independently several times; it appears in Beeri and Vardi [1981], Paredaens and Janssens [1981], and Sadri and Ullman [1981].

A somewhat more general class, called implicational dependencies in Fagin [1982] and algebraic dependencies in Yannakakis and Papadimitriou [1980], has also been investigated.

Implications of Generalized Dependencies

The "chase" as an algorithm for inferring dependencies has its roots in the lossless join test of Aho, Beeri, and Ullman [1979]. The term "chase," and its first application to the inference of dependencies, is found in Maier, Mendelzon, and Sagiv [1979]. Its application to generalized dependencies is from Beeri and Vardi [1984b].

The undecidability of implication for generalized tuple-generating dependencies was shown independently by Vardi [1984] and Gurevich and Lewis [1982]. Key results leading to the undecidability proof were contained in earlier papers by Beeri and Vardi [1981] and Chandra, Lewis, and Makowsky [1981].

Axiomatization of Generalized Dependencies

Several sound and complete axiom systems for generalized dependencies are found in Beeri and Vardi [1984a] and Sadri and Ullman [1981]. Yannakakis and Papadimitriou [1980] gives an axiom system for algebraic dependencies.

Inclusion Dependencies

Inclusion dependencies were studied by Casanova, Fagin, and Papadimitriou [1982] and Mitchell [1983]. Kanellakis, Cosmadakis, and Vardi [1983] discuss the important special case of unary inclusion dependencies (see Exercise 4.47), where the domain of a single attribute is declared to be a subset of another single attribute.

Notes on Exercises

Exercise 7.13 (on the order of reductions to produce a minimal cover) is from Maier [1980]. Exercise 7.17 (efficient computation of the closure of a set of attributes) is from Bernstein [1976], although the problem is actually equivalent to the problem of telling whether a context-free grammar generates the empty string.

Exercise 7.32, the soundness and completeness of axioms A1–A8 for functional and multivalued dependencies, is proved in Beeri, Fagin, and Howard [1977]. The algorithm for projecting functional and multivalued dependencies, Exercise 7.37, was proved correct in Aho, Beeri, and Ullman [1979].

Exercise 7.46(a), the $\mathcal{NP}$-completeness of telling whether a relation scheme has a key of given size, is by Lucchesi and Osborn [1978]; part (b), telling whether a relation scheme is in BNCF, is from Beeri and Bernstein [1979], and part (c), inferring a join dependency from multivalued dependencies, is from Fischer and Tsou [1983].

Exercise 7.48 is from Kanellakis, Cosmadakis, and Vardi [1983]; it is the key portion of a polynomial-time algorithm for making inferences of dependencies when given a set of functional dependencies and unary inclusion dependencies.