3. For no $X \rightarrow A$ in F and proper subset Z of X is $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ equivalent to F.

Intuitively, (2) guarantees that no dependency in F is redundant, and (3) guarantees that no attribute on any left side is redundant. As each right side has only one attribute by (1), surely no attribute on the right is redundant.

Theorem 7.3: Every set of dependencies F is equivalent to a set F' that is minimal.

Proof: By Lemma 7.4, assume no right side in F has more than one attribute. To satisfy condition (2), consider each dependency $X \rightarrow Y$ in F, in some order, and if $F - \{X \rightarrow Y\}$ is equivalent to F, then delete $X \rightarrow Y$ 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:

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

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

Having satisfied (2), we proceed to satisfy (3) by considering each dependency remaining in F, and each attribute in its left side, in some order. If we can eliminate an attribute from a left side and still have an equivalent set of attributes, we do so, until no more attributes can be eliminated from any left side. Again, the order in which attributes are eliminated may affect the result. For example, given

$$\begin{array}{r} AB \rightarrow C \\ A \rightarrow B \\ B \rightarrow A \end{array}$$

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

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:

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

Clearly $CE \rightarrow A$ is redundant, since it is implied by $C \rightarrow A$. $CG \rightarrow B$ is redundant, since $CG \rightarrow D$, $C \rightarrow A$, and $ACD \rightarrow B$ imply $CG \rightarrow B$, as can be checked by computing $(CG)^+$. Then no more dependencies are redundant. However,

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

Fig. 7.2. Two minimal covers.

$ACD \to B$ can be replaced by $CD \to B$, since $C \to A$ is given. Thus one minimal cover for F is shown in Fig. 7.2(a). Another minimal cover, constructed from F by eliminating $CE \to A$, $CG \to D$, and $ACD \to B$, is shown in Fig. 7.2(b). Note that the two minimal covers have different numbers of dependencies. □

7.3 DECOMPOSITION OF RELATION SCHEMES

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. In general, it is the responsibility of the person designing a database (the "database administrator") to decompose an initial set of relation schemes when warranted.

Example 7.7: Let us reconsider the SUPPLIERS relation scheme introduced in Example 3.1, but as a shorthand, let the attributes be S (SNAME), A (SADDRESS), I (ITEM), and P (PRICE). The functional dependencies we shall assume are $S \to A$ and $SI \to P$. 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 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, r is the natural join of its projections onto the R_i's. From our remarks above, it is apparent that the lossless join property is a desirable condition for a decomposition to satisfy, so we shall study the subject of lossless joins in some detail.

Some basic facts about project-join mappings follow in Lemma 7.5. First we introduce some notation. If $\rho = (R_1, R_2, \ldots, R_k)$, 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, $r = m_\rho(r)$. As another useful notational convention, if t is a tuple, we define $t[X]$, where X is a set of attributes, to be the components of t for the attributes of X.‡ For example, we could express $\pi_X(r)$ as $\{\, t[X] \mid t \text{ is in } r \,\}$.

Lemma 7.5: Let R be a relation scheme, $\rho = (R_1, \ldots, R_k)$ a decomposition of R, r a relation for R, and $r_i = \pi_{R_i}(r)$. 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 t be in r. Then for each i, $t_i = t[R_i]$ is in r_i. By definition of the natural join, t is in $m_\rho(r)$, since t agrees with t_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

† Recall Section 5.2 for a definition of the natural join.

‡ Recall that t is a mapping from attributes to values, so $t[X]$ is that mapping restricted to domain X. In practice, we always pick some ordering for the attributes and show tuples, or restricted tuples such as $t[X]$, as lists of values.

show $\pi_{R_i}(s) \subseteq r_i$, suppose for some particular i that t_i is in $\pi_{R_i}(s)$. Then there is some tuple t in s such that $t[R_i] = t_i$. As t is in s, there is some u_j in r_j for each j such that $t[R_j] = u_j$. Thus, in particular, $t[R_i]$ is in r_i. But $t[R_i] = t_i$, so t_i is in r_i, and therefore $\pi_{R_i}(s) \subseteq r_i$. We conclude that $r_i = \pi_{R_i}(s)$.

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 $X \rightarrow Y$ in F, until no more changes can be made to the table. Each time we "consider" $X \rightarrow Y$, we look for rows that agree in all 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 $b_{\ell j}$, make them both b_{ij} or $b_{\ell j}$, arbitrarily.

If after modifying the rows of the table as above, we discover that some row has become $a_1 \cdots a_k$, 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 $S \rightarrow A$ and $SI \rightarrow P$, 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 $S \rightarrow A$, 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 one row 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:

$$\begin{array}{ll} A \rightarrow C & DE \rightarrow C \\ B \rightarrow C & CE \rightarrow A \\ C \rightarrow D & \end{array}$$

The initial table is shown in Fig. 7.3(a). We can apply $A \rightarrow C$ to equate b_{13}, b_{23}, and b_{53}. Then we use $B \rightarrow C$ to equate these symbols with b_{33}; the result is shown in Fig. 7.3(b), where b_{13} has been chosen as the representative symbol. Now use $C \rightarrow D$ to equate a_4, b_{24}, b_{34}, and b_{54}; the resulting symbol is a_4. Then $DE \rightarrow C$ enables us to equate b_{13} with a_3, and $CE \rightarrow A$ lets us equate b_{31}, b_{41}, and a_1. The result is shown in Fig. 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 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 t_i in r, namely the tuple that is row i, such that $t_i[R_i]$ 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 t_i for all i. We conclude that if the final table from Algorithm 7.2 does

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)

Fig. 7.3. Applying Algorithm 7.2.

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 the table as shorthand for the domain relational calculus expression

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

where w_i is the i^{th} row of the initial table. Formula (7.1) defines the function m_ρ, since $m_\rho(r)$ contains an arbitrary tuple $a_1 \cdots a_n$ if and only if for each i, r contains a tuple with a's in the attributes of R_i and arbitrary values in the other attributes.

Since we assume that any relation r for scheme R, to which (7.1) could be applied, satisfies the dependencies F, we can infer that (7.1) is equivalent to a set of similar formulas with some of the a's and/or b's identified. The modifications made to the table by Algorithm 7.2 are such that the table is

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

Fig. 7.4. A general two row table.

always shorthand for some formula whose value on relation r is $m_\rho(r)$ whenever r satisfies F, as can be proved by an easy induction on the number of symbols identified. Since the final table contains a row with all a's, the domain calculus expression for the final table is of the form.

$$\{\, a_1 \cdots a_n \mid R(a_1 \cdots a_n) \wedge \cdots \} \tag{7.2}$$

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 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^+.

Proof: The initial table used in an application of Algorithm 7.2 is shown in Fig. 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 $F = \{A \rightarrow B\}$. Then the decomposition of R into AB and AC has a lossless join, since $AB \cap AC = A$, $AB - AC = B$,† and $A \rightarrow B$ 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 $R_1 - R_2 = A$ nor $R_2 - R_1 = C$. Thus the decomposition AB and BC does not have a lossless join with respect to $F = \{A \rightarrow B\}$, as can be seen by considering the

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

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

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 $X \rightarrow Y$ in F^+ such that $XY \subseteq Z$. (Note that $X \rightarrow Y$ 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.

The reason it is desirable that ρ preserve 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 the lossless join property 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 $CS \rightarrow Z$ and $Z \rightarrow C$. The decomposition of the relation scheme CSZ into SZ and CZ has a lossless join, since

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

However, the projection of $F = \{ CS \rightarrow Z, Z \rightarrow C \}$ onto SZ gives only the trivial dependencies that follow from reflexivity, while the projection onto CZ gives $Z \rightarrow C$ and the trivial dependencies. It can be checked that $Z \rightarrow C$ and trivial dependencies do not imply $CS \rightarrow Z$, so the decomposition does not preserve dependencies.

For example, the join of the two relations in Fig. 7.5(a) and (b) is the relation of Fig. 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 $Z \rightarrow C$. However, their join in Fig. 7.5(c) violates $CS \rightarrow Z$. $\square$

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 $F = \{ A \rightarrow B, C \rightarrow D \}$, $R = ABCD$, and $\rho = (AB, CD)$.

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)

Fig. 7.5. A join violating a functional dependency.

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 the R_i's. Take the union of the resulting sets of dependencies, and test whether this set covers F.

However, in practice, just computing F^+ is a formidable task, since the number of dependencies it contains will be 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 decompostion $\rho = (R_1, \ldots, R_k)$ 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 covers F. To test whether G covers F, we must consider each $X \rightarrow Y$ 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 $Z \cup ((Z \cap R)^+ \cap R)$, the closure being taken with respect to F. This operation adjoins to Z those attributes A such that $(Z \cap R) \rightarrow A$ 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**
        $Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$
```

If Y is a subset of the Z that results from executing the above steps, then $X \rightarrow Y$ is in G. If each $X \rightarrow Y$ 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 $F = \{A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D,\ D \rightarrow A\}$. 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 $D \rightarrow A$, 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 $A \rightarrow B$, but also $B \rightarrow A$. Similarly, we get $C \rightarrow B$ in $\pi_{BC}(F)$ and $D \rightarrow C$ in $\pi_{CD}(F)$, and these three dependencies logically imply $D \rightarrow A$. Thus, we should expect that Algorithm 7.3 will tell us that $D \rightarrow A$ 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 $\{D\} \cup ((\{D\} \cap \{A, B\})^+ \cap \{A, B\})$ is just $\{D\}$. Similarly, the BC-operation does not change Z. However, when we apply the CD-operation we get

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

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 $G \models D \rightarrow A$. 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 $X \rightarrow Y$ 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 $X \rightarrow Y$ 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 therefore it is a dependency in $\pi_{R_i}(F)$ for some i. Let one such dependency be $U \rightarrow V$. 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. □ ✓

7.4 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." These normal forms guarantee that most of the problems of redundancy and anomalies discussed in Section 7.1 do not occur.

Boyce-Codd Normal Form

The strongest of these normal forms is called Boyce-Codd. A relation scheme R with dependencies F is said to be in *Boyce-Codd normal form* if whenever $X \rightarrow A$ holds in R, and A is not in X, then X is a superkey for R; that is, X is or contains a key. Put another way, the only nontrivial dependencies are those in which a key functionally determines one or more other attributes.

✓ *Example 7.12*: The relation scheme CSZ of Example 7.10, with dependencies $CS \rightarrow Z$ and $Z \rightarrow C$, is not in Boyce-Codd normal form (although we shall see it is in third normal form). The reason is that $Z \rightarrow C$ holds (in fact it is a given dependency), but Z is not a key of CSZ, nor does it contain a key.

The MEMBERS and ORDERS relations of Example 3.1 are in Boyce-Codd normal form, since their keys, NAME and ORDER_NO respectively, are the left sides of the only dependencies that were given for their respective relations. We shall see that the SUPPLIERS relation is neither in Boyce-Codd normal form nor third normal form. □ ✓

[illegible]

Third Normal Form

It turns out that in some circumstances, Boyce-Codd normal form 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. Thus third normal form has seen use as a condition that has almost the benefits of Boyce-Codd normal form, as far as elimination of anomalies is concerned, yet that 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

[†] Yes Virginia, there is a first normal form and a second normal form. There's even a fourth normal form. All in good time ···

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 7.10, 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, so A, B, and C are prime; D is nonprime. □

A relation scheme R is in *third normal form* 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.

If $X \rightarrow A$ violates third normal form, then one of two cases can occur. Either

1. X is a proper subset of a key, or
2. X is a proper subset of no key.

In the first case, we say that $X \rightarrow A$ is a *partial dependency*, and in the second case we call it a *transitive dependency*. The term "transitive" comes from the fact that if Y is a key, then $Y \rightarrow X \rightarrow A$ is a nontrivial chain of dependencies. It is nontrivial because we know that X is not a subset of Y, by (2), A is given not to be in X, and A cannot be in Y because A is nonprime. If R has no partial dependencies, although it may have transitive dependencies, we say R is in *second normal form*.†

Example 7.14: The relation scheme $SAIP$ from Example 7.7, with dependencies $SI \rightarrow P$ and $S \rightarrow A$ violates third normal form, and in fact violates second normal form. A is a nonprime attribute, since the only key is SI. Then $S \rightarrow A$ violates the third normal form condition, since S is not a superkey. Note that in this case, the violating dependency, $S \rightarrow A$, not only holds, it is even a given dependency. In general, however, it is sufficient that the violating dependency follow from the given set of dependencies, even if it is not itself a given dependency.

As another example, the relation scheme CSZ from Example 7.12 is in third normal form. Since all of its attributes are prime, the conditions for third normal form hold vacuously.

For an example of a relation scheme in second normal form but not third, consider the attributes S (Store) I (Item) D (Department number), and M

† O.K., we might as well mention "first normal form." That form simply requires that the domain of each attribute consists of indivisible values, not sets or tuples of values from a more elementary domain or domains. We have not considered set-valued domains and so feel free to ignore first normal form. In effect, "relation" is for us synonymous with "first normal form relation" in some works appearing in the literature.

(Manager). The functional dependencies we assume are $SI \rightarrow D$ (each item in each store is sold by at most one department) and $SD \rightarrow M$ (each department in each store has one manager). The only key is SI. Then $SD \rightarrow M$ violates third normal form, since SD is not a superset of a key. Note the application of transitivity implied by the chain $SI \rightarrow SD \rightarrow M$. However, there are no partial dependencies, since no proper subset of the key SI functionally determines D or M. □

Motivation Behind Normal Forms

We may suppose that the functional dependencies $X \rightarrow Y$ not only represent an integrity constraint on relations, but also represent a relationship that the database is intended to store. That is, we regard it important to know, given an assignment of values to the attributes in X, what value for each of the Y attributes is associated with this assignment of X-values. If we have a partial dependency $Y \rightarrow A$, where X is a key and Y a proper subset of X, then in every tuple used to associate an X-value with values for other attributes besides A and those in X, the same association between Y and A must appear. This situation is best seen in the running example of the $SAIP$ scheme, where $S \rightarrow A$ is a partial dependency, and the supplier's address must be repeated once for each item supplied by the supplier. The third normal form condition eliminates this possibility and the resultant redundancy and update anomalies.

If we have a transitive dependency $X \rightarrow Y \rightarrow A$, then we cannot associate a Y-value with an X-value unless there is an A-value associated with the Y value. This situation leads to insertion and deletion anomalies, where we cannot insert an X-to-Y association without a Y-to-A association, and if we delete the A-value associated with a given Y-value, we may lose track of an X-to-Y association. For example, in the relation scheme $SIDM$ with dependencies $SI \rightarrow D$ and $SD \rightarrow M$, mentioned in Example 7.14, we cannot record the department selling hats in Bloomingdales if that department has no manager.

As we saw from Example 7.14, a relation scheme can be in third normal form but not Boyce-Codd normal form. However, every Boyce-Codd normal form relation scheme is in third normal form. The benefits of Boyce-Codd normal form are the same as for third normal form—freedom from insertion and deletion anomalies and redundancies. Note how Boyce-Codd normal form eliminates some anomalies not prevented by third normal form. For instance, in the CSZ example, we cannot record the city to which a zip code belongs unless we know a street address with that zip code.

It is worth mentioning that relations intended to represent an entity set or a many-one mapping between entity sets will be in Boyce-Codd normal form unless there are unexpected relationships among attributes. It is interesting to conjecture that all functional dependencies that satisfy third normal form but violate Boyce-Codd normal form are in a sense irrelevant. That is, they tell us something about the structure of the real world that is of no use to the database designer. For example, $Z \rightarrow C$ in the above example tells us how cities are broken into zip codes, but the information is not really useful, since the apparent application of the CSZ database is not to relate zip codes to cities, but to store zip codes for addresses.

Lossless Join Decomposition into Boyce-Codd Normal Form

We have now been introduced to the properties we desire for relation schemes: Boyce-Codd normal form or, failing that, third normal form. In the last section 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 third normal form 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 Boyce-Codd normal form 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 algorithms, we shall state some properties of natural joins that we shall need.

Lemma 7.6:

a) Suppose R is a relation scheme with functional dependencies F. Let $\rho = (R_1, \ldots, R_k)$ be a decomposition of R with a lossless join with respect to F. For a particular i, let $F_i = \pi_{R_i}(F)$, and let $\sigma = (S_1, \ldots, S_m)$ be a decomposition of R_i whose join is lossless with respect to F_i. Then the decomposition of R into $(R_1, \ldots, R_{i-1}, S_1, \ldots, S_m, R_{i+1}, \ldots, R_k)$ has a lossless join with respect to F.

b) Suppose R, F and ρ are as in (a), and let $\tau = (R_1, \ldots, R_k, R_{k+1}, \ldots, R_n)$ be a decomposition of R into a set of relation schemes that includes those of ρ. Then τ also has a lossless join with respect to F.

Proof: Each of these statements follows by algebraic manipulation from the definition of a lossless join decomposition. We shall leave formal proofs as exercises and only give the intuition here. The reason (a) holds is that if we take relation r for R and project it to relations r_j for each R_j, and then project r_i to relations s_p for each S_p, the lossless join property tells us we can join the s_p's to recover r_i. Then we can join the r_j's to recover r. Since the natural join is an associative operation (another exercise for the reader) 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 the r_j's, for $i \neq j$, and the s_p's.

For part (b), we again appeal to the associativity of the natural join. Observe that if we project relation r for R onto the R_i's, $i = 1, 2, \ldots, n$, then when we take the join of the projections onto $R_1, \ldots, R_k$ we recover r. Since $R_1, \ldots, R_k$ include all the attributes of R, further joins can only produce a subset of what we already have, which is r. But by Lemma 7.5(a), $r \subseteq m_\tau(r)$, so we cannot wind up with less than r. That is, $m_\tau(r) = r$, and τ is a lossless join decomposition. □

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: We iteratively construct a decomposition ρ for R. At all times, ρ will have a lossless join with respect to F. Initially, ρ consists of R alone. If S is a relation scheme in ρ, and S is not in Boyce-Codd normal form, let $X \rightarrow A$ be a dependency that holds in S, where X is not a superkey for S, and A is not in X. Replace S in ρ by S_1 and S_2, where S_1 consists of A and the attributes of X, and S_2 consists of all the attributes of S except for A. S_2 is surely a proper subset of S. S_1 is also a proper subset, or else $X = S - A$, so X is a superkey for S.

By Theorem 7.5, the decomposition of S into S_1 and S_2 has a lossless join with respect to the set of dependencies projected onto S, since $S_1 \cap S_2 = X$, $S_1 - S_2 = A$, and therefore $(S_1 \cap S_2) \rightarrow (S_1 - S_2)$. By Lemma 7.6(a), ρ with S replaced by S_1 and S_2 has a lossless join, if ρ does. As S_1 and S_2 each have fewer attributes then S, and any relation scheme with two or fewer attributes must be in Boyce-Codd normal form, we eventually reach a point where each relation scheme in ρ is in Boyce-Codd normal form. At that time, ρ still has a lossless join, since the initial ρ consisting of R alone does, and each modification of ρ preserves the lossless join property. □

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

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

The only key for $CTHRSG$ is HS.

To decompose this relation scheme into Boyce-Codd normal form, we might first consider the dependency $CS \rightarrow G$, which violates the condition, since CS does not contain a key. Thus, by Algorithm 7.4, we first decompose $CTHRSG$ into CSG and $CTHRS$. For further decompositions we must compute F^+ and project it onto CSG and $CTHRS$.

Note that this process is in general time consuming, as the size of F^+ can be exponential in the size of F. Even in this relatively simple example, F^+, naturally, has all the trivial dependencies that follow by reflexivity and, in addition to those in F, some other nontrivial dependencies like $CH \rightarrow R$, $HS \rightarrow C$, and $HR \rightarrow T$. Once we have F^+, we select those involving only C, S, and G. This is $\pi_{CSG}(F)$. This set has a minimal cover consisting of $CS \rightarrow G$ alone; all other dependencies in the set follow from this dependency by Armstrong's axioms. We also project F^+ onto $CTHRS$. $\pi_{CTHRS}(F)$ has a minimal cover

$$\begin{array}{ll} C \rightarrow T & TH \rightarrow R \\ HR \rightarrow C & HS \rightarrow R \end{array}$$

and the only key for $CTHRS$ is HS.

It is easy to check that CSG is in Boyce-Codd normal form with respect to its projected dependencies. $CTHRS$ must be decomposed further, and we might choose the dependency $C \rightarrow T$ to break it into CT and $CHRS$. Minimal covers for the projected dependencies are $C \rightarrow T$ for CT and $CH \rightarrow R$, $HS \rightarrow R$, and $HR \rightarrow C$ for $CHRS$; HS is the only key of the latter scheme. Observe that $CH \rightarrow R$ is needed in a cover of $CHRS$, although in $CTHRS$ it followed from $C \rightarrow T$ and $TH \rightarrow R$.

CT is in Boyce-Codd normal form, and one more decomposition of $CHRS$, say using $CH \rightarrow R$, puts the entire database scheme into the desired form. In Fig. 7.6 we see the tree of decompositions, with the keys and minimal covers for the sets of projected dependencies also shown.

The final decomposition of $CTHRSG$ is CSG, CT, CHR, and CHS. This is not a bad database design, since its four relation schemes tabulate, respectively,

1. grades for students in courses,
2. the teacher of each course,
3. the hours at which each course meets and the room for each hour, and
4. the schedule of courses and hours for each student.

In fairness it should be noted that not every decomposition produces a database scheme that matches so well our intuition about what information should be tabulated in the database. For example, if at the last decomposition step we had used dependency $HR \rightarrow C$ instead of $CH \rightarrow R$, we would have scheme HRS instead of CHS, and HRS represents the room in which a student can be found at a given hour, rather than the class he is attending. Surely the latter is more fundamental information than the former.

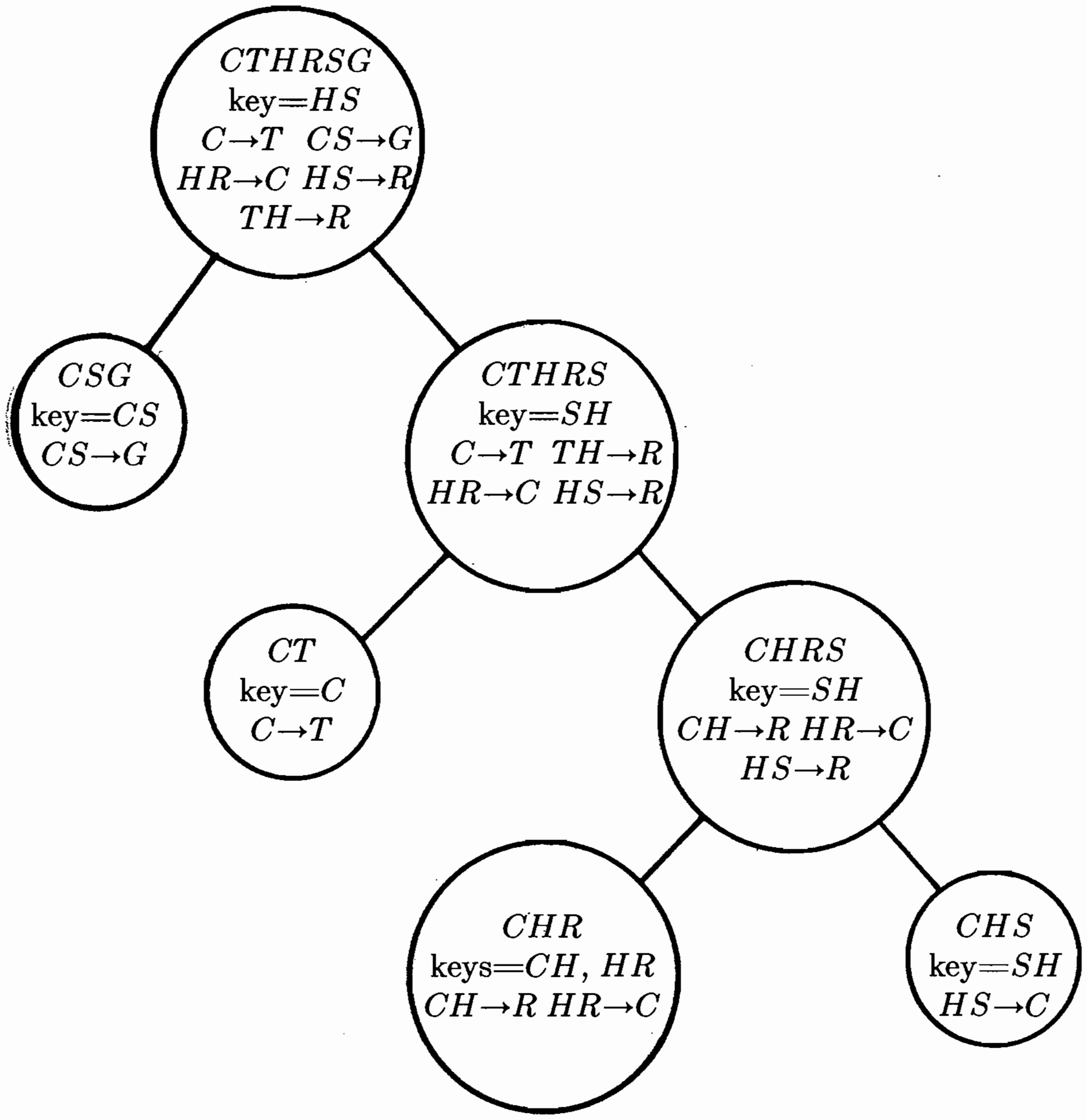

Fig. 7.6. Tree of decomposition.

Another problem with the decomposition of Fig. 7.6 is that the dependency $TH \rightarrow R$ is not preserved by the decomposition. That is, the projection of F onto CSG, CT, CHR, and CHS, which can be represented by the cover

$$\begin{array}{rl} CS \rightarrow G & \quad HR \rightarrow C \\ C \rightarrow T & \quad HS \rightarrow C \\ CH \rightarrow R & \end{array}$$

found by taking the minimal covers in each of the leaves of Fig. 7.6, does not imply $TH \rightarrow R$. 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

does not satisfy $TH \rightarrow R$, yet its projections onto CSG, CT, CHR, and CHS satisfy all the projected dependencies. □

We mentioned that the process of projecting dependencies, where we construct F^+ from F and then select out those with a particular set of attributes, can be exponential in the number of dependencies in F. One might wonder whether Algorithm 7.4 can be made to run in less than exponential time by using another approach to decomposition. Unfortunately, Beeri and Bernstein [1979] proved that it is $\mathcal{NP}$-complete† just to determine whether a relation scheme is in Boyce-Codd normal form. Thus it is extremely unlikely that one will find a substantially better algorithm.

Dependency Preserving Decompositions into Third Normal Form

We saw from Examples 7.12 and 7.14 that it is not always possible to decompose a relation scheme into Boyce-Codd normal form 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 third normal form 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 that attribute can, in principle, form a relation scheme by itself, and we shall eliminate it from R.‡ 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 $X \rightarrow A$ in F. However, if $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$ are in F, we may use scheme $XA_1 \cdots A_n$ instead of XA_i for $1 \leq i \leq n$, and in fact, this substitution is usually preferable. □

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

† $\mathcal{NP}$-completeness of a problem almost certainly implies that it is inherently exponential. See Aho, Hopcroft, and Ullman [1974] or Garey and Johnson [1979] for a description of the theory.

‡ 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 $AB \rightarrow \theta$, to force this association. After completing the design, attribute θ is eliminated.

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

Algorithm 7.5 yields the set of relation schemes CT, CHR, HRT, CGS, 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 $Y \rightarrow B$ in the minimal cover, is in third normal form. Suppose $X \rightarrow A$ violates third normal form 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 $X \rightarrow A$ 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 $X \rightarrow B$, which is $X \rightarrow A$, could replace $Y \rightarrow B$ in the supposed minimal cover, contradicting the assumption that $Y \rightarrow B$ 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 $Z \rightarrow B$ can replace $Y \rightarrow B$ in the supposedly minimal cover, again providing a contradiction. □

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

We have seen that 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 Boyce-Codd normal form (and therefore in third normal form). 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 third normal form. Can we find a decomposition into third normal form 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 third normal form 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 third normal form; the decomposition preserves dependencies and has the lossless join property.

Proof: It is easy to show that any transitive or partial dependency 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 third normal form. 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 $Y \rightarrow A_i$, 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, HRT, CGS, and 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, last mentioned in Example 7.16. □

7.5 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 in the "real world." Suppose we are given a relation scheme R, and X and Y are subsets of R. Intuitively, we say that $X \rightarrow\!\!\!\rightarrow Y$, 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 $R - X - Y$.

C	T	H	R	S	G
CS101	Deadwood, J.	M9	222	Klunk, A.	B+
CS101	Deadwood, J.	W9	333	Klunk, A.	B+
CS101	Deadwood, J.	F9	222	Klunk, A.	B+
CS101	Deadwood, J.	M9	222	Zonker, B.	C
CS101	Deadwood, J.	W9	333	Zonker, B.	C
CS101	Deadwood, J.	F9	222	Zonker, B.	C

Fig. 7.7. A sample relation for scheme $CTHRSG$.

Formally, we say $X \twoheadrightarrow Y$ holds in R if whenever r is a relation for R, and t and s are two tuples in r, with $t[X] = s[X]$ (that is, t and s agree on the attributes of X), then r also contains tuples u and v, where
1. $u[X] = v[X] = t[X] = s[X]$,
2. $u[Y] = t[Y]$ and $u[R-X-Y] = s[R-X-Y]$, and
3. $v[Y] = s[Y]$ and $v[R-X-Y] = t[R-X-Y]$.†

That is, we can exchange the Y values of t and s to obtain two new tuples 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 $CTHRSG$ of the previous section. In Fig. 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.

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, if in the formal definition of a multivalued dependency we let

$t =$ CS101	Deadwood, J.	M9	222	Klunk, A.	B+
$s =$ CS101	Deadwood, J.	W9	333	Zonker, B.	C

then we would expect to be able to exchange (M9, 222) from t with (W9, 333) in s to get the two tuples

$u =$ CS101	Deadwood, J.	M9	222	Zonker, B.	C
$v =$ CS101	Deadwood, J.	W9	333	Klunk, A.	B+

A glance at Fig. 7.7 affirms that u and v are indeed in r.

It should be emphasized that $C \twoheadrightarrow HR$ holds not because it held in the

† Note we could have eliminated clause (3). The existence of tuple v follows from the existence of u when we apply the definition with t and s interchanged.

one relation of Fig. 7.7. It holds because for 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, then we expect from our understanding of the attributes' meanings that the course c also meets at hour h_1 in room r_1 with teacher t_2 and student s_2 with grade g_2.

Note also that $C \rightarrow\!\!\rightarrow H$ does not hold, nor does $C \rightarrow\!\!\rightarrow R$. In proof, consider relation r of Fig. 7.7 with tuples t and s as above. If $C \rightarrow\!\!\rightarrow H$ held, we would expect to find tuple

CS101   Deadwood, J.   M9   333   Zonker, B.   C

in r, which we do not. There are a number of other multivalued dependencies that hold, however, such as $C \rightarrow\!\!\rightarrow SG$ and $HR \rightarrow\!\!\rightarrow SG$. There are also trivial multivalued dependencies like $HR \rightarrow\!\!\rightarrow R$. We shall in fact prove that every functional dependency $X \rightarrow Y$ that holds implies that the multivalued dependency $X \rightarrow\!\!\rightarrow Y$ 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 $X \rightarrow Y$.

A2: *(augmentation for functional dependencies)* If $X \rightarrow Y$ holds, and $Z \subseteq U$, then $XZ \rightarrow YZ$.

A3: *(transitivity for functional dependencies)* $\{X \rightarrow Y,\ Y \rightarrow Z\} \models X \rightarrow Z$.

The next three axioms apply to multivalued dependencies.

A4: *(complementation for multivalued dependencies)*

$$\{X \rightarrow\!\!\rightarrow Y\} \models X \rightarrow\!\!\rightarrow (U - X - Y)$$

A5: *(augmentation for multivalued dependencies)* If $X \rightarrow\!\!\rightarrow Y$ holds, and $V \subseteq W$, then $WX \rightarrow\!\!\rightarrow VY$.

A6: *(transitivity for multivalued dependencies)*

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

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 $X \rightarrow\!\!\rightarrow Y$ whenever $Y \subseteq X$ follows from A1 and the rule (Axiom A7, to be given) that if $X \rightarrow Y$ then $X \rightarrow\!\!\rightarrow Y$. A6 is more restrictive than its counterpart transitivity axiom, A3. The more general statement, that $X \rightarrow\!\!\rightarrow Y$ and $Y \rightarrow\!\!\rightarrow Z$ imply $X \rightarrow\!\!\rightarrow Z$ is false. For instance, we saw in Example 7.18 that