It might appear that `Movies3` still has redundancy, since the title and year of a movie can appear several times. However, these two attributes form a key for movies, and there is no more succinct way to represent a movie. Moreover, `Movies3` does not offer an opportunity for an update anomaly. For instance, one might suppose that if we changed to 2008 the year in the Carrie Fisher tuple, but not the other two tuples for *Star Wars*, then there would be an update anomaly. However, there is nothing in our assumed FD's that prevents there being a different movie named *Star Wars* in 2008, and Carrie Fisher may star in that one as well. Thus, we do not want to prevent changing the year in one *Star Wars* tuple, nor is such a change necessarily incorrect.

3.3.3 Boyce-Codd Normal Form

The goal of decomposition is to replace a relation by several that do not exhibit anomalies. There is, it turns out, a simple condition under which the anomalies discussed above can be guaranteed not to exist. This condition is called *Boyce-Codd normal form*, or *BCNF*.

- A relation R is in BCNF if and only if: whenever there is a nontrivial FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ for R, it is the case that $\{A_1, A_2, \ldots, A_n\}$ is a superkey for R.

That is, the left side of every nontrivial FD must be a superkey. Recall that a superkey need not be minimal. Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.

Example 3.15: Relation `Movies1`, as in Fig. 3.6, is not in BCNF. To see why, we first need to determine what sets of attributes are keys. We argued in Example 3.2 why {`title`, `year`, `starName`} is a key. Thus, any set of attributes containing these three is a superkey. The same arguments we followed in Example 3.2 can be used to explain why no set of attributes that does not include all three of `title`, `year`, and `starName` could be a superkey. Thus, we assert that {`title`, `year`, `starName`} is the only key for `Movies1`.

However, consider the FD

`title year` $\rightarrow$ `length genre studioName`

which holds in `Movies1` according to our discussion in Example 3.2.

Unfortunately, the left side of the above FD is not a superkey. In particular, we know that `title` and `year` do not functionally determine the sixth attribute, `starName`. Thus, the existence of this FD violates the BCNF condition and tells us `Movies1` is not in BCNF. □

Example 3.16: On the other hand, `Movies2` of Fig. 3.7 is in BCNF. Since

`title year` $\rightarrow$ `length genre studioName`

holds in this relation, and we have argued that neither `title` nor `year` by itself functionally determines any of the other attributes, the only key for `Movies2` is {`title`, `year`}. Moreover, the only nontrivial FD's must have at least `title` and `year` on the left side, and therefore their left sides must be superkeys. Thus, `Movies2` is in BCNF. □

Example 3.17: We claim that any two-attribute relation is in BCNF. We need to examine the possible nontrivial FD's with a single attribute on the right. There are not too many cases to consider, so let us consider them in turn. In what follows, suppose that the attributes are A and B.

1. There are no nontrivial FD's. Then surely the BCNF condition must hold, because only a nontrivial FD can violate this condition. Incidentally, note that $\{A, B\}$ is the only key in this case.
2. $A \rightarrow B$ holds, but $B \rightarrow A$ does not hold. In this case, A is the only key, and each nontrivial FD contains A on the left (in fact the left can only be A). Thus there is no violation of the BCNF condition.
3. $B \rightarrow A$ holds, but $A \rightarrow B$ does not hold. This case is symmetric to case (2).
4. Both $A \rightarrow B$ and $B \rightarrow A$ hold. Then both A and B are keys. Surely any FD has at least one of these on the left, so there can be no BCNF violation.

It is worth noticing from case (4) above that there may be more than one key for a relation. Further, the BCNF condition only requires that *some* key be contained in the left side of any nontrivial FD, not that all keys are contained in the left side. Also observe that a relation with two attributes, each functionally determining the other, is not completely implausible. For example, a company may assign its employees unique employee ID's and also record their Social Security numbers. A relation with attributes `empID` and `ssNo` would have each attribute functionally determining the other. Put another way, each attribute is a key, since we don't expect to find two tuples that agree on either attribute. □

3.3.4 Decomposition into BCNF

By repeatedly choosing suitable decompositions, we can break any relation schema into a collection of subsets of its attributes with the following important properties:

1. These subsets are the schemas of relations in BCNF.
2. The data in the original relation is represented faithfully by the data in the relations that are the result of the decomposition, in a sense to be made precise in Section 3.4.1. Roughly, we need to be able to reconstruct the original relation instance exactly from the decomposed relation instances.

Example 3.17 suggests that perhaps all we have to do is break a relation schema into two-attribute subsets, and the result is surely in BCNF. However, such an arbitrary decomposition will not satisfy condition (2), as we shall see in Section 3.4.1. In fact, we must be more careful and use the violating FD's to guide our decomposition.

The decomposition strategy we shall follow is to look for a nontrivial FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ that violates BCNF; i.e., $\{A_1, A_2, \ldots, A_n\}$ is not a superkey. We shall add to the right side as many attributes as are functionally determined by $\{A_1, A_2, \ldots, A_n\}$. This step is not mandatory, but it often reduces the total amount of work done, and we shall include it in our algorithm. Figure 3.8 illustrates how the attributes are broken into two overlapping relation schemas. One is all the attributes involved in the violating FD, and the other is the left side of the FD plus all the attributes *not* involved in the FD, i.e., all the attributes except those B's that are not A's.

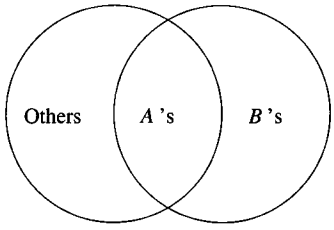

Figure 3.8: Relation schema decomposition based on a BCNF violation

Example 3.18: Consider our running example, the `Movies1` relation of Fig. 3.6. We saw in Example 3.15 that

`title year` $\rightarrow$ `length genre studioName`

is a BCNF violation. In this case, the right side already includes all the attributes functionally determined by `title` and `year`, so we shall use this BCNF violation to decompose `Movies1` into:

1. The schema {`title`, `year`, `length`, `genre`, `studioName`} consisting of all the attributes on either side of the FD.

2. The schema {`title`, `year`, `starName`} consisting of the left side of the FD plus all attributes of `Movies1` that do not appear in either side of the FD (only `starName`, in this case).

Notice that these schemas are the ones selected for relations `Movies2` and `Movies3` in Example 3.14. We observed in Example 3.16 that `Movies2` is in BCNF. `Movies3` is also in BCNF; it has no nontrivial FD's. □

In Example 3.18, one judicious application of the decomposition rule is enough to produce a collection of relations that are in BCNF. In general, that is not the case, as the next example shows.

Example 3.19: Consider a relation with schema

{title, year, studioName, president, presAddr}

That is, each tuple of this relation tells about a movie, its studio, the president of the studio, and the address of the president of the studio. Three FD's that we would assume in this relation are

title year → studioName
studioName → president
president → presAddr

By closing sets of these five attributes, we discover that {title, year} is the only key for this relation. Thus the last two FD's above violate BCNF. Suppose we choose to decompose starting with

studioName → president

First, we add to the right side of this functional dependency any other attributes in the closure of studioName. That closure includes presAddr, so our final choice of FD for the decomposition is:

studioName → president presAddr

The decomposition based on this FD yields the following two relation schemas.

{title, year, studioName}
{studioName, president, presAddr}

If we use Algorithm 3.12 to project FD's, we determine that the FD's for the first relation has a basis:

title year → studioName

while the second has:

studioName → president
president → presAddr

The sole key for the first relation is {title, year}, and it is therefore in BCNF. However, the second has {studioName} for its only key but also has the FD:

president → presAddr

which is a BCNF violation. Thus, we must decompose again, this time using the above FD. The resulting three relation schemas, all in BCNF, are:

{title, year, studioName}
{studioName, president}
{president, presAddr}

□

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in BCNF. We can be sure of ultimate success, because every time we apply the decomposition rule to a relation R, the two resulting schemas each have fewer attributes than that of R. As we saw in Example 3.17, when we get down to two attributes, the relation is sure to be in BCNF; often relations with larger sets of attributes are also in BCNF. The strategy is summarized below.

Algorithm 3.20: BCNF Decomposition Algorithm.

INPUT: A relation R_0 with a set of functional dependencies S_0.

OUTPUT: A decomposition of R_0 into a collection of relations, all of which are in BCNF.

METHOD: The following steps can be applied recursively to any relation R and set of FD's S. Initially, apply them with $R = R_0$ and $S = S_0$.

1. Check whether R is in BCNF. If so, nothing more needs to be done. Return $\{R\}$ as the answer.
2. If there are BCNF violations, let one be $X \to Y$. Use Algorithm 3.7 to compute X^+. Choose $R_1 = X^+$ as one relation schema and let R_2 have attributes X and those attributes of R that are not in X^+.
3. Use Algorithm 3.12 to compute the sets of FD's for R_1 and R_2; let these be S_1 and S_2, respectively.
4. Recursively decompose R_1 and R_2 using this algorithm. Return the union of the results of these decompositions.

□

3.3.5 Exercises for Section 3.3

Exercise 3.3.1: For each of the following relation schemas and sets of FD's:

a) $R(A, B, C, D)$ with FD's $AB \to C$, $C \to D$, and $D \to A$.

b) $R(A, B, C, D)$ with FD's $B \to C$ and $B \to D$.

c) $R(A, B, C, D)$ with FD's $AB \to C$, $BC \to D$, $CD \to A$, and $AD \to B$.

d) $R(A, B, C, D)$ with FD's $A \to B$, $B \to C$, $C \to D$, and $D \to A$.

e) $R(A, B, C, D, E)$ with FD's $AB \to C$, $DE \to C$, and $B \to D$.

f) $R(A, B, C, D, E)$ with FD's $AB \to C$, $C \to D$, $D \to B$, and $D \to E$.

do the following:

i) Indicate all the BCNF violations. Do not forget to consider FD's that are not in the given set, but follow from them. However, it is not necessary to give violations that have more than one attribute on the right side.

ii) Decompose the relations, as necessary, into collections of relations that are in BCNF.

Exercise 3.3.2: We mentioned in Section 3.3.4 that we would exercise our option to expand the right side of an FD that is a BCNF violation if possible. Consider a relation R whose schema is the set of attributes $\{A, B, C, D\}$ with FD's $A \to B$ and $A \to C$. Either is a BCNF violation, because the only key for R is $\{A, D\}$. Suppose we begin by decomposing R according to $A \to B$. Do we ultimately get the same result as if we first expand the BCNF violation to $A \to BC$? Why or why not?

! **Exercise 3.3.3:** Let R be as in Exercise 3.3.2, but let the FD's be $A \to B$ and $B \to C$. Again compare decomposing using $A \to B$ first against decomposing by $A \to BC$ first.

! **Exercise 3.3.4:** Suppose we have a relation schema $R(A, B, C)$ with FD $A \to B$. Suppose also that we decide to decompose this schema into $S(A, B)$ and $T(B, C)$. Give an example of an instance of relation R whose projection onto S and T and subsequent rejoining as in Section 3.4.1 does not yield the same relation instance. That is, $\pi_{A,B}(R) \bowtie \pi_{B,C}(R) \neq R$.

3.4 Decomposition: The Good, Bad, and Ugly

So far, we observed that before we decompose a relation schema into BCNF, it can exhibit anomalies; after we decompose, the resulting relations do not exhibit anomalies. That's the "good." But decomposition can also have some bad, if not downright ugly, consequences. In this section, we shall consider three distinct properties we would like a decomposition to have.

1. *Elimination of Anomalies* by decomposition as in Section 3.3.
2. *Recoverability of Information.* Can we recover the original relation from the tuples in its decomposition?
3. *Preservation of Dependencies.* If we check the projected FD's in the relations of the decomposition, can we can be sure that when we reconstruct the original relation from the decomposition by joining, the result will satisfy the original FD's?

It turns out that the BCNF decomposition of Algorithm 3.20 gives us (1) and (2), but does not necessarily give us all three. In Section 3.5 we shall see another way to pick a decomposition that gives us (2) and (3) but does not necessarily give us (1). In fact, there is no way to get all three at once.

3.4.1 Recovering Information from a Decomposition

Since we learned that every two-attribute relation is in BCNF, why did we have to go through the trouble of Algorithm 3.20? Why not just take any relation R and decompose it into relations, each of whose schemas is a pair of R's attributes? The answer is that the data in the decomposed relations, even if their tuples were each the projection of a relation instance of R, might not allow us to join the relations of the decomposition and get the instance of R back. If we do get R back, then we say the decomposition has a *lossless join.*

However, if we decompose using Algorithm 3.20, where all decompositions are motivated by a BCNF-violating FD, then the projections of the original tuples can be joined again to produce all and only the original tuples. We shall consider why here. Then, in Section 3.4.2 we shall give an algorithm called the "chase," for testing whether the projection of a relation onto any decomposition allows us to recover the relation by rejoining.

To simplify the situation, consider a relation $R(A, B, C)$ and an FD $B \to C$ that is a BCNF violation. The decomposition based on the FD $B \to C$ separates the attributes into relations $R_1(A, B)$ and $R_2(B, C)$.

Let t be a tuple of R. We may write $t = (a, b, c)$, where a, b, and c are the components of t for attributes A, B, and C, respectively. Tuple t projects as (a, b) in $R_1(A, B) = \pi_{A,B}(R)$ and as (b, c) in $R_2(B, C) = \pi_{B,C}(R)$. When we compute the natural join $R_1 \bowtie R_2$, these two projected tuples join, because they agree on the common B component (they both have b there). They give us $t = (a, b, c)$, the tuple we started with, in the join. That is, regardless of what tuple t we started with, we can always join its projections to get t back.

However, getting back those tuples we started with is not enough to assure that the original relation R is truly represented by the decomposition. Consider what happens if there are two tuples of R, say $t = (a, b, c)$ and $v = (d, b, e)$. When we project t onto $R_1(A, B)$ we get $u = (a, b)$, and when we project v onto $R_2(B, C)$ we get $w = (b, e)$. These tuples also match in the natural join, and the resulting tuple is $x = (a, b, e)$. Is it possible that x is a bogus tuple? That is, could (a, b, e) not be a tuple of R?

Since we assume the FD $B \to C$ for relation R, the answer is "no." Recall that this FD says any two tuples of R that agree in their B components must also agree in their C components. Since t and v agree in their B components, they also agree on their C components. That means $c = e$; i.e., the two values we supposed were different are really the same. Thus, tuple (a, b, e) of R is really (a, b, c); that is, $x = t$.

Since t is in R, it must be that x is in R. Put another way, as long as FD $B \to C$ holds, the joining of two projected tuples cannot produce a bogus tuple.

Rather, every tuple produced by the natural join is guaranteed to be a tuple of R.

This argument works in general. We assumed A, B, and C were each single attributes, but the same argument would apply if they were any sets of attributes X, Y and Z. That is, if $Y \rightarrow Z$ holds in R, whose attributes are $X \cup Y \cup Z$, then $R = \pi_{X \cup Y}(R) \bowtie \pi_{Y \cup Z}(R)$.

We may conclude:

- If we decompose a relation according to Algorithm 3.20, then the original relation can be recovered exactly by the natural join.

To see why, we argued above that at any one step of the recursive decomposition, a relation is equal to the join of its projections onto the two components. If those components are decomposed further, they can also be recovered by the natural join from their decomposed relations. Thus, an easy induction on the number of binary decomposition steps says that the original relation is always the natural join of whatever relations it is decomposed into. We can also prove that the natural join is associative and commutative, so the order in which we perform the natural join of the decomposition components does not matter.

The FD $Y \rightarrow Z$, or its symmetric FD $Y \rightarrow X$, is essential. Without one of these FD's, we might not be able to recover the original relation. Here is an example.

Example 3.21: Suppose we have the relation $R(A, B, C)$ as above, but neither of the FD's $B \rightarrow A$ nor $B \rightarrow C$ holds. Then R might consist of the two tuples

A	B	C
1	2	3
4	2	5

The projections of R onto the relations with schemas $\{A, B\}$ and $\{B, C\}$ are $R_1 = \pi_{AB}(R) =$

A	B
1	2
4	2

and $R_2 = \pi_{BC}(R) =$

B	C
2	3
2	5

respectively. Since all four tuples share the same B-value, 2, each tuple of one relation joins with both tuples of the other relation. When we try to reconstruct R by the natural join of the projected relations, we get $R_3 = R_1 \bowtie R_2 =$

Is Join the Only Way to Recover?

We have assumed that the only possible way we could reconstruct a relation from its projections is to use the natural join. However, might there be some other algorithm to reconstruct the original relation that would work even in cases where the natural join fails? There is in fact no such other way. In Example 3.21, the relations R and R_3 are different instances, yet have exactly the same projections onto $\{A, B\}$ and $\{B, C\}$, namely the instances we called R_1 and R_2, respectively. Thus, given R_1 and R_2, no algorithm whatsoever can tell whether the original instance was R or R_3.

Moreover, this example is not unusual. Given any decomposition of a relation with attributes $X \cup Y \cup Z$ into relations with schemas $X \cup Y$ and $Y \cup Z$, where neither $Y \to X$ nor $Y \to Z$ holds, we can construct an example similar to Example 3.21 where the original instance cannot be determined from its projections.

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

That is, we get "too much"; we get two bogus tuples, $(1, 2, 5)$ and $(4, 2, 3)$, that were not in the original relation R. □

3.4.2 The Chase Test for Lossless Join

In Section 3.4.1 we argued why a particular decomposition, that of $R(A, B, C)$ into $\{A, B\}$ and $\{B, C\}$, with a particular FD, $B \to C$, had a lossless join. Now, consider a more general situation. We have decomposed relation R into relations with sets of attributes $S_1, S_2, \ldots, S_k$. We have a given set of FD's F that hold in R. Is it true that if we project R onto the relations of the decomposition, then we can recover R by taking the natural join of all these relations? That is, is it true that $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples t such that for all $i = 1, 2, \ldots, k$, t projected onto the set of attributes S_i is a tuple in $\pi_{S_i}(R)$.

- Any tuple t in R is surely in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$. The reason is that the projection of t onto S_i is surely in $\pi_{S_i}(R)$ for each i, and therefore by our first point above, t is in the result of the join.

- As a consequence, $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$ when the FD's in F hold for R if and only if every tuple in the join is also in R. That is, the membership test is all we need to verify that the decomposition has a lossless join.

The *chase* test for a lossless join is just an organized way to see whether a tuple t in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ can be proved, using the FD's in F, also to be a tuple in R. If t is in the join, then there must be tuples in R, say $t_1, t_2, \ldots, t_k$, such that t is the join of the projections of each t_i onto the set of attributes S_i, for $i = 1, 2, \ldots, k$. We therefore know that t_i agrees with t on the attributes of S_i, but t_i has unknown values in its components not in S_i.

We draw a picture of what we know, called a *tableau*. Assuming R has attributes $A, B, \ldots$ we use $a, b, \ldots$ for the components of t. For t_i, we use the same letter as t in the components that are in S_i, but we subscript the letter with i if the component is not in i. In that way, t_i will agree with t for the attributes of S_i, but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

Example 3.22: Suppose we have relation $R(A, B, C, D)$, which we have decomposed into relations with sets of attributes $S_1 = \{A, D\}$, $S_2 = \{A, C\}$, and $S_3 = \{B, C, D\}$. Then the tableau for this decomposition is shown in Fig. 3.9.

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

Figure 3.9: Tableau for the decomposition of R into $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$

The first row corresponds to set of attributes A and D. Notice that the components for attributes A and D are the unsubscripted letters a and d. However, for the other attributes, b and c, we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple (a, b_1, c_1, d) represents a tuple of R that contributes to $t = (a, b, c, d)$ by being projected onto $\{A, D\}$ and then joined with other tuples. Since the B- and C-components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.

Similarly, the second row has the unsubscripted letters in attributes A and C, while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for $\{B, C, D\}$ and subscript 3 on a. Since

each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters. □

Remember that our goal is to use the given set of FD's F to prove that t is really in R. In order to do so, we "chase" the tableau by applying the FD's in F to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as t (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple t in the join of the projections was actually a tuple of R.

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurences.

Example 3.23: Let us continue with the decomposition of Example 3.22, and suppose the given FD's are $A \to B$, $B \to C$, and $CD \to A$. Start with the tableau of Fig. 3.9. Since the first two rows agree in their A-components, the FD $A \to B$ tells us they must also agree in their B-components. That is, $b_1 = b_2$. We can replace either one with the other, since they are both subscripted. Let us replace b_2 by b_1. Then the resulting tableau is:

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Now, we see that the first two rows have equal B-values, and so we may use the FD $B \to C$ to deduce that their C-components, c_1 and c, are the same. Since c is unsubscripted, we replace c_1 by c, leaving:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

Next, we observe that the first and third rows agree in both columns C and D. Thus, we may apply the FD $CD \to A$ to deduce that these rows also have the same A-value; that is, $a = a_3$. We replace a_3 by a, giving us:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

At this point, we see that the last row has become equal to t, that is, (a, b, c, d). We have proved that if R satisfies the FD's $A \to B$, $B \to C$, and $CD \to A$, then whenever we project onto $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$ and rejoin, what we get must have been in R. In particular, what we get is the same as the tuple of R that we projected onto $\{B, C, D\}$. □

3.4.3 Why the Chase Works

There are two issues to address:

1. When the chase results in a row that matches the tuple t (i.e., the tableau is shown to have a row with all unsubscripted variables), why must the join be lossless?
2. When, after applying FD's whenever we can, we still find no row of all unsubscripted variables, why must the join not be lossless?

Question (1) is easy to answer. The chase process itself is a proof that one of the projected tuples from R must in fact be the tuple t that is produced by the join. We also know that every tuple in R is sure to come back if we project and join. Thus, the chase has proved that the result of projection and join is exactly R.

For the second question, suppose that we eventually derive a tableau without an unsubscripted row, and that this tableau does not allow us to apply any of the FD's to equate any symbols. Then think of the tableau as an instance of the relation R. It obviously satisfies the given FD's, because none can be applied to equate symbols. We know that the ith row has unsubscripted symbols in the attributes of S_i, the ith relation of the decomposition. Thus, when we project this relation onto the S_i's and take the natural join, we get the tuple with all unsubscripted variables. This tuple is not in R, so we conclude that the join is not lossless.

Example 3.24: Consider the relation $R(A, B, C, D)$ with the FD $B \to AD$ and the proposed decomposition $\{A, B\}$, $\{B, C\}$, and $\{C, D\}$. Here is the initial tableau:

A	B	C	D
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

When we apply the lone FD, we deduce that $a = a_2$ and $d_1 = d_2$. Thus, the final tableau is:

A	B	C	D
a	b	c_1	d_1
a	b	c	d_1
a_3	b_3	c	d

No more changes can be made because of the given FD's, and there is no row that is fully unsubscripted. Thus, this decomposition does not have a lossless join. We can verify that fact by treating the above tableau as a relation with three tuples. When we project onto $\{A, B\}$, we get $\{(a, b)\}$, $(a_3, b_3)\}$. The projection onto $\{B, C\}$ is $\{(b, c_1)$, (b, c), $(b_3, c)\}$, and the projection onto $\{C, D\}$ is (c_1, d_1), (c, d_1), $(c, d)\}$. If we join the first two projections, we get $\{(a, b, c_1)$, (a, b, c), $(a_3, b_3, c)\}$. Joining this relation with the third projection gives $\{(a, b, c_1, d_1)$, (a, b, c, d_1), (a, b, c, d), (a_3, b_3, c, d_1), $(a_3, b_3, c, d)\}$. Notice that this join has two more tuples than R, and in particular it has the tuple (a, b, c, d), as it must. □

3.4.4 Dependency Preservation

We mentioned that it is not possible, in some cases, to decompose a relation into BCNF relations that have both the lossless-join and dependency-preservation properties. Below is an example where we need to make a tradeoff between preserving dependencies and BNCF.

Example 3.25: Suppose we have a relation `Bookings` with attributes:

1. `title`, the name of a movie.
2. `theater`, the name of a theater where the movie is being shown.
3. `city`, the city where the theater is located.

The intent behind a tuple (m, t, c) is that the movie with title m is currently being shown at theater t in city c.

We might reasonably assert the following FD's:

`theater` → `city`
`title city` → `theater`

The first says that a theater is located in one city. The second is not obvious but is based on the common practice of not booking a movie into two theaters in the same city. We shall assert this FD if only for the sake of the example.

Let us first find the keys. No single attribute is a key. For example, `title` is not a key because a movie can play in several theaters at once and in several cities at once.[2] Also, `theater` is not a key, because although `theater` functionally determines `city`, there are multiscreen theaters that show many movies at once. Thus, `theater` does not determine `title`. Finally, `city` is not a key because cities usually have more than one theater and more than one movie playing.

[2] In this example we assume that there are not two "current" movies with the same title, even though we have previously recognized that there could be two movies with the same title made in different years.

On the other hand, two of the three sets of two attributes are keys. Clearly {title, city} is a key because of the given FD that says these attributes functionally determine theater.

It is also true that {theater, title} is a key, because its closure includes city due to the given FD theater → city. The remaining pair of attributes, city and theater, do not functionally determine title, because of multiscreen theaters, and are therefore not a key. We conclude that the only two keys are

{title, city}
{theater, title}

Now we immediately see a BCNF violation. We were given functional dependency theater → city, but its left side, theater, is not a superkey. We are therefore tempted to decompose, using this BCNF-violating FD, into the two relation schemas:

{theater, city}
{theater, title}

There is a problem with this decomposition, concerning the FD

title city→theater

There could be current relations for the decomposed schemas that satisfy the FD theater → city (which can be checked in the relation {theater, city}) but that, when joined, yield a relation not satisfying title city→theater. For instance, the two relations

theater	*city*
Guild	Menlo Park
Park	Menlo Park

and

theater	*title*
Guild	Antz
Park	Antz

are permissible according to the FD's that apply to each of the above relations, but when we join them we get two tuples

theater	*city*	*title*
Guild	Menlo Park	Antz
Park	Menlo Park	Antz

that violate the FD title city → theater. □

3.4.5 Exercises for Section 3.4

Exercise 3.4.1: Let $R(A, B, C, D, E)$ be decomposed into relations with the following three sets of attributes: $\{A, B, C\}$, $\{B, C, D\}$, and $\{A, C, E\}$. For each of the following sets of FD's, use the chase test to tell whether the decomposition of R is lossless. For those that are not lossless, give an example of an instance of R that returns more than R when projected onto the decomposed relations and rejoined.

a) $B \to E$ and $CE \to A$.

b) $AC \to E$ and $BC \to D$.

c) $A \to D$, $D \to E$, and $B \to D$.

d) $A \to D$, $CD \to E$, and $E \to D$.

! **Exercise 3.4.2:** For each of the sets of FD's in Exercise 3.4.1, are dependencies preserved by the decomposition?

3.5 Third Normal Form

The solution to the problem illustrated by Example 3.25 is to relax our BCNF requirement slightly, in order to allow the occasional relation schema that cannot be decomposed into BCNF relations without our losing the ability to check the FD's. This relaxed condition is called "third normal form." In this section we shall give the requirements for third normal form, and then show how to do a decomposition in a manner quite different from Algorithm 3.20, in order to obtain relations in third normal form that have both the lossless-join and dependency-preservation properties.

3.5.1 Definition of Third Normal Form

A relation R is in *third normal form* (3NF) if:

- Whenever $A_1 A_2 \cdots A_n \to B_1 B_2 \cdots B_m$ is a nontrivial FD, either

  $$\{A_1, A_2, \ldots, A_n\}$$

  is a superkey, or those of $B_1, B_2, \ldots, B_m$ that are not among the A's, are each a member of some key (not necessarily the same key).

An attribute that is a member of some key is often said to be *prime*. Thus, the 3NF condition can be stated as "for each nontrivial FD, either the left side is a superkey, or the right side consists of prime attributes only."

Note that the difference between this 3NF condition and the BCNF condition is the clause "is a member of some key (i.e., prime)." This clause "excuses" an FD like `theater` $\to$ `city` in Example 3.25, because the right side, `city`, is prime.

Other Normal Forms

If there is a "third normal form," what happened to the first two "normal forms"? They indeed were defined, but today there is little use for them. *First normal form* is simply the condition that every component of every tuple is an atomic value. *Second normal form* is a less restrictive verison of 3NF. There is also a "fourth normal form" that we shall meet in Section 3.6.

3.5.2 The Synthesis Algorithm for 3NF Schemas

We can now explain and justify how we decompose a relation R into a set of relations such that:

a) The relations of the decomposition are all in 3NF.

b) The decomposition has a lossless join.

c) The decomposition has the dependency-preservation property.

Algorithm 3.26: Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.

INPUT: A relation R and a set F of functional dependencies that hold for R.

OUTPUT: A decomposition of R into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.

METHOD: Perform the following steps:

1. Find a minimal basis for F, say G.
2. For each functional dependency $X \to A$ in G, use XA as the schema of one of the relations in the decomposition.
3. If none of the relation schemas from Step 2 is a superkey for R, add another relation whose schema is a key for R.

□

Example 3.27: Consider the relation $R(A, B, C, D, E)$ with FD's $AB \to C$, $C \to B$, and $A \to D$. To start, notice that the given FD's are their own minimal basis. To check, we need to do a bit of work. First, we need to verify that we cannot eliminate any of the given dependencies. That is, we show, using Algorithm 3.7, that no two of the FD's imply the third. For example, we must take the closure of $\{A, B\}$, the left side of the first FD, using only the

second and third FD's, $C \to B$ and $A \to D$. This closure includes D but not C, so we conclude that the first FD $AB \to C$ is not implied by the second and third FD's. We get a similar conclusion if we try to drop the second or third FD.

We must also verify that we cannot eliminate any attributes from a left side. In this simple case, the only possibility is that we could eliminate A or B from the first FD. For example, if we eliminate A, we would be left with $B \to C$. We must show that $B \to C$ is not implied by the three original FD's, $AB \to C$, $C \to B$, and $A \to D$. With these FD's, the closure of $\{B\}$ is just B, so $B \to C$ does not follow. A similar conclusion is drawn if we try to drop B from $AB \to C$. Thus, we have our minimal basis.

We start the 3NF synthesis by taking the attributes of each FD as a relation schema. That is, we get relations $S_1(A, B, C)$, $S_2(B, C)$, and $S_3(A, D)$. It is never necessary to use a relation whose schema is a proper subset of another relation's schema, so we can drop S_2.

We must also consider whether we need to add a relation whose schema is a key. In this example, R has two keys: $\{A, B, E\}$ and $\{A, C, E\}$, as you can verify. Neither of these keys is a subset of the schemas chosen so far. Thus, we must add one of them, say $S_4(A, B, E)$. The final decomposition of R is thus $S_1(A, B, C)$, $S_3(A, D)$, and $S_4(A, B, E)$. □

3.5.3 Why the 3NF Synthesis Algorithm Works

We need to show three things: that the lossless-join and dependency-preservation properties hold, and that all the relations of the decomposition are in 3NF.

1. *Lossless Join.* Start with a relation of the decomposition whose set of attributes K is a superkey. Consider the sequence of FD's that are used in Algorithm 3.7 to expand K to become K^+. Since K is a superkey, we know K^+ is all the attributes. The same sequence of FD applications on the tableau cause the subscripted symbols in the row corresponding to K to be equated to unsubscripted symbols in the same order as the attributes were added to the closure. Thus, the chase test concludes that the decomposition is lossless.

2. *Dependency Preservation.* Each FD of the minimal basis has all its attributes in some relation of the decomposition. Thus, each dependency can be checked in the decomposed relations.

3. *Third Normal Form.* If we have to add a relation whose schema is a key, then this relation is surely in 3NF. The reason is that all attributes of this relation are prime, and thus no violation of 3NF could be present in this relation. For the relations whose schemas are derived from the FD's of a minimal basis, the proof that they are in 3NF is beyond the scope of this book. The argument involves showing that a 3NF violation implies that the basis is not minimal.

3.5.4 Exercises for Section 3.5

Exercise 3.5.1: For each of the relation schemas and sets of FD's of Exercise 3.3.1:

i) Indicate all the 3NF violations.

ii) Decompose the relations, as necessary, into collections of relations that are in 3NF.

Exercise 3.5.2: Consider the relation `Courses`(C, T, H, R, S, G), whose attributes may be thought of informally as course, teacher, hour, room, student, and grade. Let the set of FD's for `Courses` be $C \to T$, $HR \to C$, $HT \to R$, $HS \to R$, and $CS \to G$. Intuitively, the first says that a course has a unique teacher, and the second says that only one course can meet in a given room at a given hour. The third says that a teacher can be in only one room at a given hour, and the fourth says the same about students. The last says that students get only one grade in a course.

a) What are all the keys for `Courses`?

b) Verify that the given FD's are their own minimal basis.

c) Use the 3NF synthesis algorithm to find a lossless-join, dependency-preserving decomposition of R into 3NF relations. Are any of the relations not in BCNF?

Exercise 3.5.3: Consider a relation `Stocks`(B, O, I, S, Q, D), whose attributes may be thought of informally as broker, office (of the broker), investor, stock, quantity (of the stock owned by the investor), and dividend (of the stock). Let the set of FD's for `Stocks` be $S \to D$, $I \to B$, $IS \to Q$, and $B \to O$. Repeat Exercise 3.5.2 for the relation `Stocks`.

Exercise 3.5.4: Verify, using the chase, that the decomposition of Example 3.27 has a lossless join.

!! **Exercise 3.5.5:** Suppose we modified Algorithm 3.20 (BNCF decomposition) so that instead of decomposing a relation R whenever R was not in BCNF, we only decomposed R if it was not in 3NF. Provide a counterexample to show that this modified algorithm would not necessarily produce a 3NF decomposition with dependency preservation.

3.6 Multivalued Dependencies

A "multivalued dependency" is an assertion that two attributes or sets of attributes are independent of one another. This condition is, as we shall see, a generalization of the notion of a functional dependency, in the sense that

every FD implies the corresponding multivalued dependency. However, there are some situations involving independence of attribute sets that cannot be explained as FD's. In this section we shall explore the cause of multivalued dependencies and see how they can be used in database schema design.

3.6.1 Attribute Independence and Its Consequent Redundancy

There are occasional situations where we design a relation schema and find it is in BCNF, yet the relation has a kind of redundancy that is not related to FD's. The most common source of redundancy in BCNF schemas is an attempt to put two or more set-valued properties of the key into a single relation.

Example 3.28: In this example, we shall suppose that stars may have several addresses, which we break into street and city components. The set of addresses is one of the set-valued properties this relation will store. The second set-valued property of stars that we shall put into this relation is the set of titles and years of movies in which the star appeared. Then Fig. 3.10 is a typical instance of this relation.

name	*street*	*city*	*title*	*year*
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

Figure 3.10: Sets of addresses independent from movies

We focus in Fig. 3.10 on Carrie Fisher's two hypothetical addresses and her three best-known movies. There is no reason to associate an address with one movie and not another. Thus, the only way to express the fact that addresses and movies are independent properties of stars is to have each address appear with each movie. But when we repeat address and movie facts in all combinations, there is obvious redundancy. For instance, Fig. 3.10 repeats each of Carrie Fisher's addresses three times (once for each of her movies) and each movie twice (once for each address).

Yet there is no BCNF violation in the relation suggested by Fig. 3.10. There are, in fact, no nontrivial FD's at all. For example, attribute `city` is not functionally determined by the other four attributes. There might be a star with two homes that had the same street address in different cities. Then there would be two tuples that agreed in all attributes but `city` and disagreed in `city`. Thus,

$$\texttt{name street title year} \rightarrow \texttt{city}$$

is not an FD for our relation. We leave it to the reader to check that none of the five attributes is functionally determined by the other four. Since there are no nontrivial FD's, it follows that all five attributes form the only key and that there are no BCNF violations. □

3.6.2 Definition of Multivalued Dependencies

A *multivalued dependency* (abbreviated MVD) is a statement about some relation R that when you fix the values for one set of attributes, then the values in certain other attributes are independent of the values of all the other attributes in the relation. More precisely, we say the MVD

$$A_1 A_2 \cdots A_n \twoheadrightarrow B_1 B_2 \cdots B_m$$

holds for a relation R if when we restrict ourselves to the tuples of R that have particular values for each of the attributes among the A's, then the set of values we find among the B's is independent of the set of values we find among the attributes of R that are not among the A's or B's. Still more precisely, we say this MVD holds if

> For each pair of tuples t and u of relation R that agree on all the A's, we can find in R some tuple v that agrees:
>
> 1. With both t and u on the A's,
> 2. With t on the B's, and
> 3. With u on all attributes of R that are not among the A's or B's.

Note that we can use this rule with t and u interchanged, to infer the existence of a fourth tuple w that agrees with u on the B's and with t on the other attributes. As a consequence, for any fixed values of the A's, the associated values of the B's and the other attributes appear in all possible combinations in different tuples. Figure 3.11 suggests how v relates to t and u when an MVD holds. However, the A's and B's to not have to appear consecutively.

In general, we may assume that the A's and B's (left side and right side) of an MVD are disjoint. However, as with FD's, it is permissible to add some of the A's to the right side if we wish.

Example 3.29: In Example 3.28 we encountered an MVD that in our notation is expressed:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

That is, for each star's name, the set of addresses appears in conjunction with each of the star's movies. For an example of how the formal definition of this MVD applies, consider the first and fourth tuples from Fig. 3.10:

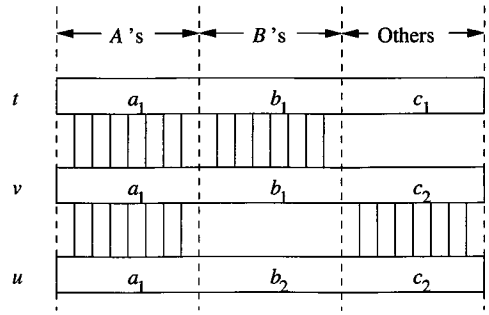

Figure 3.11: A multivalued dependency guarantees that v exists

name	*street*	*city*	*title*	*year*
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980

If we let the first tuple be t and the second be u, then the MVD asserts that we must also find in R the tuple that has name C. Fisher, a street and city that agree with the first tuple, and other attributes (title and year) that agree with the second tuple. There is indeed such a tuple; it is the third tuple of Fig. 3.10.

Similarly, we could let t be the second tuple above and u be the first. Then the MVD tells us that there is a tuple of R that agrees with the second in attributes name, street, and city and with the first in name, title, and year. This tuple also exists; it is the second tuple of Fig. 3.10. □

3.6.3 Reasoning About Multivalued Dependencies

There are a number of rules about MVD's that are similar to the rules we learned for FD's in Section 3.2. For example, MVD's obey

- *Trivial MVD's.* The MVD

$$A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$$

holds in any relation if $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$.

- The *transitive rule*, which says that if $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ and $B_1B_2\cdots B_m \twoheadrightarrow C_1C_2\cdots C_k$ hold for some relation, then so does

$$A_1A_2\cdots A_n \twoheadrightarrow C_1C_2\cdots C_k$$

Any C's that are also A's must be deleted from the right side.

On the other hand, MVD's do not obey the splitting part of the splitting/combining rule, as the following example shows.

Example 3.30: Consider again Fig. 3.10, where we observed the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

If the splitting rule applied to MVD's, we would expect

$$\texttt{name} \twoheadrightarrow \texttt{street}$$

also to be true. This MVD says that each star's street addresses are independent of the other attributes, including `city`. However, that statement is false. Consider, for instance, the first two tuples of Fig. 3.10. The hypothetical MVD would allow us to infer that the tuples with the streets interchanged:

name	*street*	*city*	*title*	*year*
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple St.	Malibu	Star Wars	1977

were in the relation. But these are not true tuples, because, for instance, the home on 5 Locust Ln. is in Malibu, not Hollywood. □

However, there are several new rules dealing with MVD's that we can learn.

- *FD Promotion.* Every FD is an MVD. That is, if

  $$A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$$

  then $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$.

  To see why, suppose R is some relation for which the FD

  $$A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$$

  holds, and suppose t and u are tuples of R that agree on the A's. To show that the MVD $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ holds, we have to show that R also contains a tuple v that agrees with t and u on the A's, with t on the B's, and with u on all other attributes. But v can be u. Surely u agrees with t and u on the A's, because we started by assuming that these two tuples agree on the A's. The FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ assures us that u agrees with t on the B's. And of course u agrees with itself on the other attributes. Thus, whenever an FD holds, the corresponding MVD holds.

- *Complementation Rule.* If $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ is an MVD for relation R, then R also satisfies $A_1A_2\cdots A_n \twoheadrightarrow C_1C_2\cdots C_k$, where the C's are all attributes of R not among the A's and B's.

That is, swapping the B's between two tuples that agree in the A's has the same effect as swapping the C's.

Example 3.31: Again consider the relation of Fig. 3.10, for which we asserted the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

The complementation rule says that

$$\texttt{name} \twoheadrightarrow \texttt{title year}$$

must also hold in this relation, because `title` and `year` are the attributes not mentioned in the first MVD. The second MVD intuitively means that each star has a set of movies starred in, which are independent of the star's addresses. □

An MVD whose right side is a subset of the left side is trivial — it holds in every relation. However, an interesting consequence of the complementation rule is that there are some other MVD's that are trivial, but that look distinctly nontrivial.

- *More Trivial MVD's.* If all the attributes of relation R are

  $$\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$$

  then $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$ holds in R.

To see why these additional trivial MVD's hold, notice that if we take two tuples that agree in $A_1, A_2, \ldots, A_n$ and swap their components in attributes $B_1, B_2, \ldots, B_m$, we get the same two tuples back, although in the opposite order.

3.6.4 Fourth Normal Form

The redundancy that we found in Section 3.6.1 to be caused by MVD's can be eliminated if we use these dependencies for decomposition. In this section we shall introduce a new normal form, called "fourth normal form." In this normal form, all nontrivial MVD's are eliminated, as are all FD's that violate BCNF. As a result, the decomposed relations have neither the redundancy from FD's that we discussed in Section 3.3.1 nor the redundancy from MVD's that we discussed in Section 3.6.1.

The "fourth normal form" condition is essentially the BCNF condition, but applied to MVD's instead of FD's. Formally:

- A relation R is in *fourth normal form* (4NF) if whenever

  $$A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$$

is a nontrivial MVD, $\{A_1, A_2, \ldots, A_n\}$ is a superkey.

That is, if a relation is in 4NF, then every nontrivial MVD is really an FD with a superkey on the left. Note that the notions of keys and superkeys depend on FD's only; adding MVD's does not change the definition of "key."

Example 3.32: The relation of Fig. 3.10 violates the 4NF condition. For example,

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is a nontrivial MVD, yet name by itself is not a superkey. In fact, the only key for this relation is all the attributes. □

Fourth normal form is truly a generalization of BCNF. Recall from Section 3.6.3 that every FD is also an MVD. Thus, every BCNF violation is also a 4NF violation. Put another way, every relation that is in 4NF is therefore in BCNF.

However, there are some relations that are in BCNF but not 4NF. Figure 3.10 is a good example. The only key for this relation is all five attributes, and there are no nontrivial FD's. Thus it is surely in BCNF. However, as we observed in Example 3.32, it is not in 4NF.

3.6.5 Decomposition into Fourth Normal Form

The 4NF decomposition algorithm is quite analogous to the BCNF decomposition algorithm.

Algorithm 3.33: Decomposition into Fourth Normal Form.

INPUT: A relation R_0 with a set of functional and multivalued dependencies S_0.

OUTPUT: A decomposition of R_0 into relations all of which are in 4NF. The decomposition has the lossless-join property.

METHOD: Do the following steps, with $R = R_0$ and $S = S_0$:

1. Find a 4NF violation in R, say $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$, where

   $$\{A_1, A_2, \ldots, A_n\}$$

   is not a superkey. Note this MVD could be a true MVD in S, or it could be derived from the corresponding FD $A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m$ in S, since every FD is an MVD. If there is none, return; R by itself is a suitable decomposition.

2. If there is such a 4NF violation, break the schema for the relation R that has the 4NF violation into two schemas:

(a) R_1, whose schema is A's and the B's.

(b) R_2, whose schema is the A's and all attributes of R that are not among the A's or B's.

3. Find the FD's and MVD's that hold in R_1 and R_2 (Section 3.7 explains how to do this task in general, but often this "projection" of dependencies is straightforward). Recursively decompose R_1 and R_2 with respect to their projected dependencies.

□

Example 3.34: Let us continue Example 3.32. We observed that

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

was a 4NF violation. The decomposition rule above tells us to replace the five-attribute schema by one schema that has only the three attributes in the above MVD and another schema that consists of the left side, `name`, plus the attributes that do not appear in the MVD. These attributes are `title` and `year`, so the following two schemas

$$\{\texttt{name, street, city}\}$$
$$\{\texttt{name, title, year}\}$$

are the result of the decomposition. In each schema there are no nontrivial multivalued (or functional) dependencies, so they are in 4NF. Note that in the relation with schema {`name, street, city`}, the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is trivial since it involves all attributes. Likewise, in the relation with schema {`name, title, year`}, the MVD:

$$\texttt{name} \twoheadrightarrow \texttt{title year}$$

is trivial. Should one or both schemas of the decomposition not be in 4NF, we would have had to decompose the non-4NF schema(s). □

As for the BCNF decomposition, each decomposition step leaves us with schemas that have strictly fewer attributes than we started with, so eventually we get to schemas that need not be decomposed further; that is, they are in 4NF. Moreover, the argument justifying the decomposition that we gave in Section 3.4.1 carries over to MVD's as well. When we decompose a relation because of an MVD $A_1A_2\cdots A_n \twoheadrightarrow B_1B_2\cdots B_m$, this dependency is enough to justify the claim that we can reconstruct the original relation from the relations of the decomposition.

We shall, in Section 3.7, give an algorithm by which we can verify that the MVD used to justify a 4NF decomposition also proves that the decomposition has a lossless join. Also in that section, we shall show how it is possible, although time-consuming, to perform the projection of MVD's onto the decomposed relations. This projection is required if we are to decide whether or not further decomposition is necessary.

3.6.6 Relationships Among Normal Forms

As we have mentioned, 4NF implies BCNF, which in turn implies 3NF. Thus, the sets of relation schemas (including dependencies) satisfying the three normal forms are related as in Fig. 3.12. That is, if a relation with certain dependencies is in 4NF, it is also in BCNF and 3NF. Also, if a relation with certain dependencies is in BCNF, then it is in 3NF.

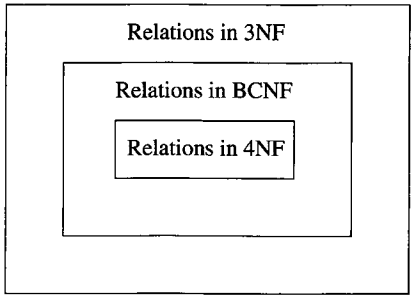

Figure 3.12: 4NF implies BCNF implies 3NF

Another way to compare the normal forms is by the guarantees they make about the set of relations that result from a decomposition into that normal form. These observations are summarized in the table of Fig. 3.13. That is, BCNF (and therefore 4NF) eliminates the redundancy and other anomalies that are caused by FD's, while only 4NF eliminates the additional redundancy that is caused by the presence of MVD's that are not FD's. Often, 3NF is enough to eliminate this redundancy, but there are examples where it is not. BCNF does not guarantee preservation of FD's, and none of the normal forms guarantee preservation of MVD's, although in typical cases the dependencies are preserved.

Property	3NF	BCNF	4NF
Eliminates redundancy due to FD's	No	Yes	Yes
Eliminates redundancy due to MVD's	No	No	Yes
Preserves FD's	Yes	No	No
Preserves MVD's	No	No	No

Figure 3.13: Properties of normal forms and their decompositions

3.6.7 Exercises for Section 3.6

Exercise 3.6.1: Suppose we have a relation $R(A, B, C)$ with an MVD $A \twoheadrightarrow$

B. If we know that the tuples (a, b_1, c_1), (a, b_2, c_2), and (a, b_3, c_3) are in the current instance of R, what other tuples do we know must also be in R?

Exercise 3.6.2: Suppose we have a relation in which we want to record for each person their name, Social Security number, and birthdate. Also, for each child of the person, the name, Social Security number, and birthdate of the child, and for each automobile the person owns, its serial number and make. To be more precise, this relation has all tuples

$$(n, s, b, cn, cs, cb, as, am)$$

such that

1. n is the name of the person with Social Security number s.
2. b is n's birthdate.
3. cn is the name of one of n's children.
4. cs is cn's Social Security number.
5. cb is cn's birthdate.
6. as is the serial number of one of n's automobiles.
7. am is the make of the automobile with serial number as.

For this relation:

a) Tell the functional and multivalued dependencies we would expect to hold.

b) Suggest a decomposition of the relation into 4NF.

Exercise 3.6.3: For each of the following relation schemas and dependencies

a) $R(A, B, C, D)$ with MVD's $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$.

b) $R(A, B, C, D)$ with MVD's $A \twoheadrightarrow B$ and $B \twoheadrightarrow CD$.

c) $R(A, B, C, D)$ with MVD $AB \twoheadrightarrow C$ and FD $B \to D$.

d) $R(A, B, C, D, E)$ with MVD's $A \twoheadrightarrow B$ and $AB \twoheadrightarrow C$ and FD's $A \to D$ and $AB \to E$.

do the following:

i) Find all the 4NF violations.

ii) Decompose the relations into a collection of relation schemas in 4NF.

Exercise 3.6.4: Give informal arguments why we would not expect any of the five attributes in Example 3.28 to be functionally determined by the other four.

3.7 An Algorithm for Discovering MVD's

Reasoning about MVD's, or combinations of MVD's and FD's, is rather more difficult than reasoning about FD's alone. For FD's, we have Algorithm 3.7 to decide whether or not an FD follows from some given FD's. In this section, we shall first show that the closure algorithm is really the same as the chase algorithm we studied in Section 3.4.2. The ideas behind the chase can be extended to incorporate MVD's as well as FD's. Once we have that tool in place, we can solve all the problems we need to solve about MVD's and FD's, such as finding whether an MVD follows from given dependencies or projecting MVD's and FD's onto the relations of a decomposition.

3.7.1 The Closure and the Chase

In Section 3.2.4 we saw how to take a set of attributes X and compute its closure X^+ of all attributes that functionally depend on X. In that manner, we can test whether an FD $X \to Y$ follows from a given set of FD's F, by closing X with respect to F and seeing whether $Y \subseteq X^+$. We could see the closure as a variant of the chase, in which the starting tableau and the goal condition are different from what we used in Section 3.4.2.

Suppose we start with a tableau that consists of two rows. These rows agree in the attributes of X and disagree in all other attributes. If we apply the FD's in F to chase this tableau, we shall equate the symbols in exactly those columns that are in $X^+ - X$. Thus, a chase-based test for whether $X \to Y$ follows from F can be summarized as:

1. Start with a tableau having two rows that agree only on X.
2. Chase the tableau using the FD's of F.
3. If the final tableau agrees in all columns of Y, then $X \to Y$ holds; otherwise it does not.

Example 3.35: Let us repeat Example 3.8, where we had a relation

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

with FD's $AB \to C$, $BC \to AD$, $D \to E$, and $CF \to B$. We want to test whether $AB \to D$ holds. Start with the tableau:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_2	d_2	e_2	f_2

We can apply $AB \to C$ to infer $c_1 = c_2$; say both become c_1. The resulting tableau is:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_1	d_2	e_2	f_2

Next, apply $BC \rightarrow AD$ to infer that $d_1 = d_2$, and apply $D \rightarrow E$ to infer $e_1 = e_2$. At this point, the tableau is:

A	B	C	D	E	F
a	b	c_1	d_1	e_1	f_1
a	b	c_1	d_1	e_1	f_2

and we can go no further. Since the two tuples now agree in the D column, we know that $AB \rightarrow D$ does follow from the given FD's. □

3.7.2 Extending the Chase to MVD's

The method of inferring an FD using the chase can be applied to infer MVD's as well. When we try to infer an FD, we are asking whether we can conclude that two possibly unequal values must indeed be the same. When we apply an FD $X \rightarrow Y$, we search for pairs of rows in the tableau that agree on all the columns of X, and we force the symbols in each column of Y to be equal.

However, MVD's do not tell us to conclude symbols are equal. Rather, $X \twoheadrightarrow Y$ tells us that if we find two rows of the tableau that agree in X, then we can form two new tuples by swapping all their components in the attributes of Y; the resulting two tuples must also be in the relation, and therefore in the tableau. Likewise, if we want to infer some MVD $X \twoheadrightarrow Y$ from given FD's and MVD's, we start with a tableau consisting of two tuples that agree in X and disagree in all attributes not in the set X. We apply the given FD's to equate symbols, and we apply the given MVD's to swap the values in certain attributes between two existing rows of the tableau in order to add new rows to the tableau. If we ever discover that one of the original tuples, with its components for Y replaced by those of the other original tuple, is in the tableau, then we have inferred the MVD.

There is a point of caution to be observed in this more complex chase process. Since symbols may get equated and replaced by other symbols, we may not recognize that we have created one of the desired tuples, because some of the original symbols may be replaced by others. The simplest way to avoid a problem is to define the target tuple initially, and never change its symbols. That is, let the target row be one with an unsubscripted letter in each component. Let the two initial rows of the tableau for the test of $X \twoheadrightarrow Y$ have the unsubscripted letters in X. Let the first row also have unsubscripted letters in Y, and let the second row have the unsubscripted letters in all attributes not in X or Y. Fill in the other positions of the two rows with new symbols that each occur only once. When we equate subscripted and unsubscripted symbols, always replace a subscripted one by the unsubscripted one, as we did in Section 3.4.2. Then, when applying the chase, we have only to ask whether the all-unsubscripted-letters row ever appears in the tableau.

Example 3.36: Suppose we have a relation $R(A, B, C, D)$ with given dependencies $A \to B$ and $B \twoheadrightarrow C$. We wish to prove that $A \twoheadrightarrow C$ holds in R. Start with the two-row tableau that represents $A \twoheadrightarrow C$:

A	B	C	D
a	b_1	c	d_1
a	b	c_2	d

Notice that our target row is (a, b, c, d). Both rows of the tableau have the unsubscripted letter in the column for A. The first row has the unsubscripted letter in C, and the second row has unsubscripted letters in the remaining columns.

We first apply the FD $A \to B$ to infer that $b = b_1$. We must therefore replace the subscripted b_1 by the unsubscripted b. The tableau becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d

Next, we apply the MVD $B \twoheadrightarrow C$, since the two rows now agree in the B column. We swap the C columns to get two more rows which we add to the tableau, which becomes:

A	B	C	D
a	b	c	d_1
a	b	c_2	d
a	b	c_2	d_1
a	b	c	d

We have now a row with all unsubscripted symbols, which proves that $A \twoheadrightarrow C$ holds in relation R. Notice how the tableau manipulations really give a proof that $A \twoheadrightarrow C$ holds. This proof is: "Given two tuples of R that agree in A, they must also agree in B because $A \to B$. Since they agree in B, we can swap their C components by $B \twoheadrightarrow C$, and the resulting tuples will be in R. Thus, if two tuples of R agree in A, the tuples that result when we swap their C's are also in R; i.e., $A \twoheadrightarrow C$." □

Example 3.37: There is a surprising rule for FD's and MVD's that says whenever there is an MVD $X \twoheadrightarrow Y$, and any FD whose right side is a (not necessarily proper) subset of Y, say Z, then $X \to Z$. We shall use the chase process to prove a simple example of this rule. Let us be given relation $R(A, B, C, D)$ with MVD $A \twoheadrightarrow BC$ and FD $D \to C$. We claim that $A \to C$.

Since we are trying to prove an FD, we don't have to worry about a target tuple of unsubscripted letters. We can start with any two tuples that agree in A and disagree in every other column. such as:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_2	d_2

Our goal is to prove that $c_1 = c_2$.

The only thing we can do to start is to apply the MVD $A \twoheadrightarrow BC$, since the two rows agree on A, but no other columns. When we swap the B and C columns of these two rows, we get two new rows to add:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_2	d_2
a	b_2	c_2	d_1
a	b_1	c_1	d_2

Now, we have pairs of rows that agree in D, so we can apply the FD $D \rightarrow C$. For instance, the first and third rows have the same D-value d_1, so we can apply the FD and conclude $c_1 = c_2$. That is our goal, so we have proved $A \rightarrow C$. The new tableau is:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_1	d_2
a	b_2	c_1	d_1
a	b_1	c_1	d_2

It happens that no further changes are possible, using the given dependencies. However, that doesn't matter, since we already proved what we need. □

3.7.3 Why the Chase Works for MVD's

The arguments are essentially the same as we have given before. Each step of the chase, whether it equates symbols or generates new rows, is a true observation about tuples of the given relation R that is justified by the FD or MVD that we apply in that step. Thus, a positive conclusion of the chase is always a proof that the concluded FD or MVD holds in R.

When the chase ends in failure — the goal row (for an MVD) or the desired equality of symbols (for an FD) is not produced — then the final tableau is a counterexample. It satisfies the given dependencies, or else we would not be finished making changes. However, it does not satisfy the dependency we were trying to prove.

There is one other issue that did not come up when we performed the chase using only FD's. Since the chase with MVD's adds rows to the tableau, how do we know we ever terminate the chase? Could we keep adding rows forever, never reaching our goal, but not sure that after a few more steps we would achieve that goal? Fortunately, that cannot happen. The reason is that we

never create any new symbols. We start out with at most two symbols in each of k columns, and all rows we create will have one of these two symbols in its component for that column. Thus, we cannot ever have more than 2^k rows in our tableau, if k is the number of columns. The chase with MVD's can take exponential time, but it cannot run forever.

3.7.4 Projecting MVD's

Recall that our reason for wanting to infer MVD's was to perform a cascade of decompositions leading to 4NF relations. To do that task, we need to be able to project the given dependencies onto the schemas of the two relations that we get in the first step of the decomposition. Only then can we know whether they are in 4NF or need to be decomposed further.

In the worst case, we have to test every possible FD and MVD for each of the decomposed relations. The chase test is applied on the full set of attributes of the original relation. However, the goal for an MVD is to produce a row of the tableau that has unsubscripted letters in all the attributes of one of the relations of the decomposition; that row may have any letters in the other attributes. The goal for an FD is the same: equality of the symbols in a given column.

Example 3.38: Suppose we have a relation $R(A, B, C, D, E)$ that we decompose, and let one of the relations of the decomposition be $S(A, B, C)$. Suppose that the MVD $A \twoheadrightarrow CD$ holds in R. Does this MVD imply any dependency in S? We claim that $A \twoheadrightarrow C$ holds in S, as does $A \twoheadrightarrow B$ (by the complementation rule). Let us verify that $A \twoheadrightarrow C$ holds in S. We start with the tableau:

A	B	C	D	E
a	b_1	c	d_1	e_1
a	b	c_2	d	e

Use the MVD of R, $A \twoheadrightarrow CD$ to swap the C and D components of these two rows to get two new rows:

A	B	C	D	E
a	b_1	c	d_1	e_1
a	b	c_2	d	e
a	b_1	c_2	d	e_1
a	b	c	d_1	e

Notice that the last row has unsubscripted symbols in all the attributes of S, that is, A, B, and C. That is enough to conclude that $A \twoheadrightarrow C$ holds in S. □

Often, our search for FD's and MVD's in the projected relations does not have to be completely exhaustive. Here are some simplifications.

1. It is surely not necessary to check the trivial FD's and MVD's.
2. For FD's, we can restrict ourselves to looking for FD's with a singleton right side, because of the combining rule for FD's.
3. An FD or MVD whose left side does not contain the left side of any given dependency surely cannot hold, since there is no way for its chase test to get started. That is, the two rows with which you start the test are unchanged by the given dependencies.

3.7.5 Exercises for Section 3.7

Exercise 3.7.1: Use the chase test to tell whether each of the following dependencies hold in a relation $R(A, B, C, D, E)$ with the dependencies $A \twoheadrightarrow BC$, $B \rightarrow D$, and $C \twoheadrightarrow E$.

a) $A \rightarrow D$.

b) $A \twoheadrightarrow D$.

c) $A \rightarrow E$.

d) $A \twoheadrightarrow E$.

! **Exercise 3.7.2:** If we project the relation R of Exercise 3.7.1 onto $S(A, C, E)$, what nontrivial FD's and MVD's hold in S?

! **Exercise 3.7.3:** Show the following rules for MVD's. In each case, you can set up the proof as a chase test, but you must think a little more generally than in the examples, since the set of attributes are arbitrary sets X, Y, Z, and the other unnamed attributes of the relation in which these dependencies hold.

a) The *Union Rule.* If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y \cup Z)$.

b) The *Intersection Rule.* If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y \cap Z)$.

c) The *Difference Rule.* If X, Y, and Z are sets of attributes, $X \twoheadrightarrow Y$, and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow (Y - Z)$.

d) *Removing attributes shared by left and right side.* If $X \twoheadrightarrow Y$ holds, then $X \twoheadrightarrow (Y - X)$ holds.

! **Exercise 3.7.4:** Give counterexample relations to show why the following rules for MVD's do *not* hold. *Hint*: apply the chase test and see what happens.

a) If $A \twoheadrightarrow BC$, then $A \twoheadrightarrow B$.

b) If $A \twoheadrightarrow B$, then $A \rightarrow B$.

c) If $AB \twoheadrightarrow C$, then $A \twoheadrightarrow C$.

3.8 Summary of Chapter 3

- ✦ *Functional Dependencies*: A functional dependency is a statement that two tuples of a relation that agree on some particular set of attributes must also agree on some other particular set of attributes.
- ✦ *Keys of a Relation*: A superkey for a relation is a set of attributes that functionally determines all the attributes of the relation. A key is a superkey, no proper subset of which is also a superkey.
- ✦ *Reasoning About Functional Dependencies*: There are many rules that let us infer that one FD $X \rightarrow A$ holds in any relation instance that satisfies some other given set of FD's. To verify that $X \rightarrow A$ holds, compute the closure of X, using the given FD's to expand X until it includes A.
- ✦ *Minimal Basis for a set of FD's*: For any set of FD's, there is at least one minimal basis, which is a set of FD's equivalent to the original (each set implies the other set), with singleton right sides, no FD that can be eliminated while preserving equivalence, and no attribute in a left side that can be eliminated while preserving equivalence.
- ✦ *Boyce-Codd Normal Form*: A relation is in BCNF if the only nontrivial FD's say that some superkey functionally determines one or more of the other attributes. A major benefit of BCNF is that it eliminates redundancy caused by the existence of FD's.
- ✦ *Lossless-Join Decomposition*: A useful property of a decomposition is that the original relation can be recovered exactly by taking the natural join of the relations in the decomposition. Any decomposition gives us back at least the tuples with which we start, but a carelessly chosen decomposition can give tuples in the join that were not in the original relation.
- ✦ *Dependency-Preserving Decomposition*: Another desirable property of a decomposition is that we can check all the functional dependencies that hold in the original relation by checking FD's in the decomposed relations.
- ✦ *Third Normal Form*: Sometimes decomposition into BCNF can lose the dependency-preservation property. A relaxed form of BCNF, called 3NF, allows an FD $X \rightarrow A$ even if X is not a superkey, provided A is a member of some key. 3NF does not guarantee to eliminate all redundancy due to FD's, but often does so.
- ✦ *The Chase*: We can test whether a decomposition has the lossless-join property by setting up a tableau — a set of rows that represent tuples of the original relation. We chase a tableau by applying the given functional dependencies to infer that certain pairs of symbols must be the same. The decomposition is lossless with respect to a given set of FD's if and only if the chase leads to a row identical to the tuple whose membership in the join of the projected relations we assumed.

- *Synthesis Algorithm for 3NF*: If we take a minimal basis for a given set of FD's, turn each of these FD's into a relation, and add a key for the relation, if necessary, the result is a decomposition into 3NF that has the lossless-join and dependency-preservation properties.

- *Multivalued Dependencies*: A multivalued dependency is a statement that two sets of attributes in a relation have sets of values that appear in all possible combinations.

- *Fourth Normal Form*: MVD's can also cause redundancy in a relation. 4NF is like BCNF, but also forbids nontrivial MVD's whose left side is not a superkey. It is possible to decompose a relation into 4NF without losing information.

- *Reasoning About MVD's*: We can infer MVD's and FD's from a given set of MVD's and FD's by a chase process. We start with a two-row tableau that represent the dependency we are trying to prove. FD's are applied by equating symbols, and MVD's are applied by adding rows to the tableau that have the appropriate components interchanged.

3.9 References for Chapter 3

Third normal form was described in [6]. This paper introduces the idea of functional dependencies, as well as the basic relational concept. Boyce-Codd normal form is in a later paper [7].

Multivalued dependencies and fourth normal form were defined by Fagin in [9]. However, the idea of multivalued dependencies also appears independently in [8] and [11].

Armstrong was the first to study rules for inferring FD's [2]. The rules for FD's that we have covered here (including what we call "Armstrong's axioms") and rules for inferring MVD's as well, come from [3].

The technique for testing an FD by computing the closure for a set of attributes is from [4], as is the fact that a minimal basis provides a 3NF decomposition. The fact that this decomposition provides the lossless-join and dependency-preservation propoerties is from [5].

The tableau test for the lossless-join property and the chase are from [1]. More information and the history of the idea is found in [10].

1. A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Transactions on Database Systems* 4:3, pp. 297-314, 1979.

2. W. W. Armstrong, "Dependency structures of database relationships," *Proceedings of the 1974 IFIP Congress*, pp. 580–583.

3. C. Beeri, R. Fagin, and J. H. Howard, "A complete axiomatization for functional and multivalued dependencies," *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 47–61, 1977.

4. P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Transactions on Database Systems* **1**:4, pp. 277–298, 1976.

5. J. Biskup, U. Dayal, and P. A. Bernstein, "Synthesizing independent database schemas," *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 143–152, 1979.

6. E. F. Codd, "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp. 377–387, 1970.

7. E. F. Codd, "Further normalization of the data base relational model," in *Database Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

8. C. Delobel, "Normalization and hierarchical dependencies in the relational data model," *ACM Transactions on Database Systems* **3**:3, pp. 201–222, 1978.

9. R. Fagin, "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems* **2**:3, pp. 262–278, 1977.

10. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

11. C. Zaniolo and M. A. Melkanoff, "On the design of relational database schemata," *ACM Transactions on Database Systems* **6**:1, pp. 1–47, 1981.

Chapter 4

High-Level Database Models

Let us consider the process whereby a new database, such as our movie database, is created. Figure 4.1 suggests the process. We begin with a design phase, in which we address and answer questions about what information will be stored, how information elements will be related to one another, what constraints such as keys or referential integrity may be assumed, and so on. This phase may last for a long time, while options are evaluated and opinions are reconciled. We show this phase in Fig. 4.1 as the conversion of ideas to a high-level design.

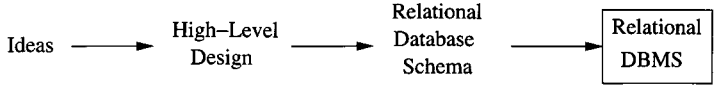

Figure 4.1: The database modeling and implementation process

Since the great majority of commercial database systems use the relational model, we might suppose that the design phase should use this model too. However, in practice it is often easier to start with a higher-level model and then convert the design to the relational model. The primary reason for doing so is that the relational model has only one concept — the relation — rather than several complementary concepts that more closely model real-world situations. Simplicity of concepts in the relational model is a great strength of the model, especially when it comes to efficient implementation of database operations. Yet that strength becomes a weakness when we do a preliminary design, which is why it often is helpful to begin by using a high-level design model.

There are several options for the notation in which the design is expressed. The first, and oldest, method is the "entity-relationship diagram," and here is where we shall start in Section 4.1. A more recent trend is the use of UML ("Unified Modeling Language"), a notation that was originally designed for

describing object-oriented software projects, but which has been adapted to describe database schemas as well. We shall see this model in Section 4.7. Finally, in Section 4.9, we shall consider ODL ("Object Description Language"), which was created to describe databases as collections of classes and their objects.

The next phase shown in Fig. 4.1 is the conversion of our high-level design to a relational design. This phase occurs only when we are confident of the high-level design. Whichever of the high-level models we use, there is a fairly mechanical way of converting the high-level design into a relational database schema, which then runs on a conventional DBMS. Sections 4.5 and 4.6 discuss conversion of E/R diagrams to relational database schemas. Section 4.8 does the same for UML, and Section 4.10 serves for ODL.

4.1 The Entity/Relationship Model

In the *entity-relationship model* (or *E/R model*). the structure of data is represented graphically, as an "entity-relationship diagram," using three principal element types:

1. Entity sets,
2. Attributes, and
3. Relationships.

We shall cover each in turn.

4.1.1 Entity Sets

An *entity* is an abstract object of some sort, and a collection of similar entities forms an *entity set*. An entity in some ways resembles an "object" in the sense of object-oriented programming. Likewise, an entity set bears some resemblance to a class of objects. However, the E/R model is a static concept, involving the structure of data and not the operations on data. Thus, one would not expect to find methods associated with an entity set as one would with a class.

Example 4.1: Let us consider the design of our running movie-database example. Each movie is an entity, and the set of all movies constitutes an entity set. Likewise, the stars are entities, and the set of stars is an entity set. A studio is another kind of entity, and the set of studios is a third entity set that will appear in our examples. □

4.1.2 Attributes

Entity sets have associated *attributes*, which are properties of the entities in that set. For instance, the entity set *Movies* might be given attributes such as *title* and *length*. It should not surprise you if the attributes for the entity

E/R Model Variations

In some versions of the E/R model, the type of an attribute can be either:

1. A primitive type, as in the version presented here.
2. A "struct," as in C, or tuple with a fixed number of primitive components.
3. A set of values of one type: either primitive or a "struct" type.

For example, the type of an attribute in such a model could be a set of pairs, each pair consisting of an integer and a string.

set *Movies* resemble the attributes of the relation `Movies` in our example. It is common for entity sets to be implemented as relations, although not every relation in our final relational design will come from an entity set.

In our version of the E/R model, we shall assume that attributes are of primitive types, such as strings, integers, or reals. There are other variations of this model in which attributes can have some limited structure; see the box on "E/R Model Variations."

4.1.3 Relationships

Relationships are connections among two or more entity sets. For instance, if *Movies* and *Stars* are two entity sets, we could have a relationship *Stars-in* that connects movies and stars. The intent is that a movie entity m is related to a star entity s by the relationship *Stars-in* if s appears in movie m. While binary relationships, those between two entity sets, are by far the most common type of relationship, the E/R model allows relationships to involve any number of entity sets. We shall defer discussion of these multiway relationships until Section 4.1.7.

4.1.4 Entity-Relationship Diagrams

An *E/R diagram* is a graph representing entity sets, attributes, and relationships. Elements of each of these kinds are represented by nodes of the graph, and we use a special shape of node to indicate the kind, as follows:

- Entity sets are represented by rectangles.
- Attributes are represented by ovals.
- Relationships are represented by diamonds.

Edges connect an entity set to its attributes and also connect a relationship to its entity sets.

Example 4.2: In Fig. 4.2 is an E/R diagram that represents a simple database about movies. The entity sets are *Movies*, *Stars*, and *Studios*.

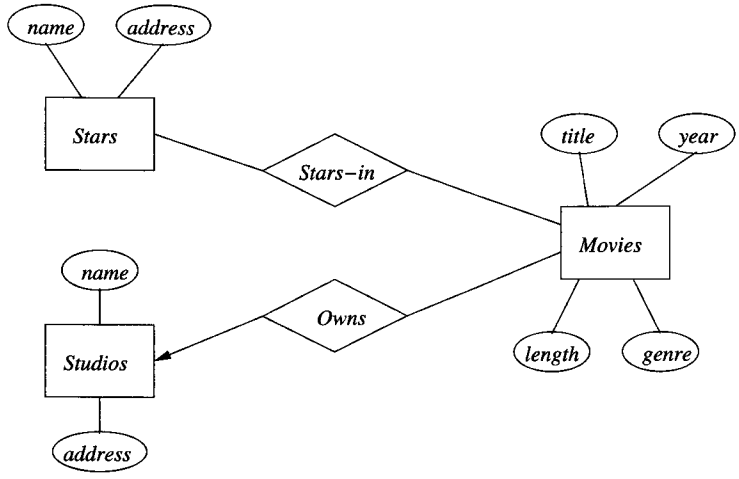

Figure 4.2: An entity-relationship diagram for the movie database

The *Movies* entity set has four of our usual attributes: *title*, *year*, *length*, and *genre*. The other two entity sets *Stars* and *Studios* happen to have the same two attributes: *name* and *address*, each with an obvious meaning. We also see two relationships in the diagram:

1. *Stars-in* is a relationship connecting each movie to the stars of that movie. This relationship consequently also connects stars to the movies in which they appeared.

2. *Owns* connects each movie to the studio that owns the movie. The arrow pointing to entity set *Studios* in Fig. 4.2 indicates that each movie is owned by at most one studio. We shall discuss uniqueness constraints such as this one in Section 4.1.6.

□

4.1.5 Instances of an E/R Diagram

E/R diagrams are a notation for describing schemas of databases. We may imagine that a database described by an E/R diagram contains particular data, an "instance" of the database. Since the database is not implemented in the E/R model, only designed, the instance never exists in the sense that a relation's

instances exist in a DBMS. However, it is often useful to visualize the database being designed as if it existed.

For each entity set, the database instance will have a particular finite set of entities. Each of these entities has particular values for each attribute. A relationship R that connects n entity sets $E_1, E_2, \ldots, E_n$ may be imagined to have an "instance" that consists of a finite set of tuples $(e_1, e_2, \ldots, e_n)$, where each e_i is chosen from the entities that are in the current instance of entity set E_i. We regard each of these tuples as "connected" by relationship R.

This set of tuples is called the *relationship set* for R. It is often helpful to visualize a relationship set as a table or relation. However, the "tuples" of a relationship set are not really tuples of a relation, since their components are entities rather than primitive types such as strings or integers. The columns of the table are headed by the names of the entity sets involved in the relationship, and each list of connected entities occupies one row of the table. As we shall see, however, when we convert relationships to relations, the resulting relation is not the same as the relationship set.

Example 4.3: An instance of the *Stars-in* relationship could be visualized as a table with pairs such as:

Movies	*Stars*
Basic Instinct	Sharon Stone
Total Recall	Arnold Schwarzenegger
Total Recall	Sharon Stone

The members of the relationship set are the rows of the table. For instance, (Basic Instinct, Sharon Stone) is a tuple in the relationship set for the current instance of relationship *Stars-in*. □

4.1.6 Multiplicity of Binary E/R Relationships

In general, a binary relationship can connect any member of one of its entity sets to any number of members of the other entity set. However, it is common for there to be a restriction on the "multiplicity" of a relationship. Suppose R is a relationship connecting entity sets E and F. Then:

- If each member of E can be connected by R to at most one member of F, then we say that R is *many-one* from E to F. Note that in a many-one relationship from E to F, each entity in F can be connected to many members of E. Similarly, if instead a member of F can be connected by R to at most one member of E, then we say R is many-one from F to E (or equivalently, one-many from E to F).

- If R is both many-one from E to F and many-one from F to E, then we say that R is *one-one*. In a one-one relationship an entity of either entity set can be connected to at most one entity of the other set.

- If R is neither many-one from E to F or from F to E, then we say R is *many-many.*

As we mentioned in Example 4.2, arrows can be used to indicate the multiplicity of a relationship in an E/R diagram. If a relationship is many-one from entity set E to entity set F, then we place an arrow entering F. The arrow indicates that each entity in set E is related to at most one entity in set F. Unless there is also an arrow on the edge to E, an entity in F may be related to many entities in E.

Example 4.4: A one-one relationship between entity sets E and F is represented by arrows pointing to both E and F. For instance, Fig. 4.3 shows two entity sets, *Studios* and *Presidents*, and the relationship *Runs* between them (attributes are omitted). We assume that a president can run only one studio and a studio has only one president, so this relationship is one-one, as indicated by the two arrows, one entering each entity set.

Figure 4.3: A one-one relationship

Remember that the arrow means "at most one"; it does not guarantee existence of an entity of the set pointed to. Thus, in Fig. 4.3, we would expect that a "president" is surely associated with some studio; how could they be a "president" otherwise? However, a studio might not have a president at some particular time, so the arrow from *Runs* to *Presidents* truly means "at most one" and not "exactly one." We shall discuss the distinction further in Section 4.3.3. □

4.1.7 Multiway Relationships

The E/R model makes it convenient to define relationships involving more than two entity sets. In practice, ternary (three-way) or higher-degree relationships are rare, but they occasionally are necessary to reflect the true state of affairs. A multiway relationship in an E/R diagram is represented by lines from the relationship diamond to each of the involved entity sets.

Example 4.5: In Fig. 4.4 is a relationship *Contracts* that involves a studio, a star, and a movie. This relationship represents that a studio has contracted with a particular star to act in a particular movie. In general, the value of an E/R relationship can be thought of as a relationship set of tuples whose components are the entities participating in the relationship, as we discussed in Section 4.1.5. Thus, relationship *Contracts* can be described by triples of the form (studio, star, movie).

Implications Among Relationship Types

We should be aware that a many-one relationship is a special case of a many-many relationship, and a one-one relationship is a special case of a many-one relationship. Thus, any useful property of many-one relationships holds for one-one relationships too. For example, a data structure for representing many-one relationships will work for one-one relationships, although it might not be suitable for many-many relationships.

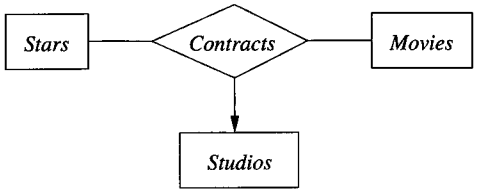

Figure 4.4: A three-way relationship

In multiway relationships, an arrow pointing to an entity set E means that if we select one entity from each of the other entity sets in the relationship, those entities are related to at most one entity in E. (Note that this rule generalizes the notation used for many-one, binary relationships.) Informally, we may think of a functional dependency with E on the right and all the other entity sets of the relationship on the left.

In Fig. 4.4 we have an arrow pointing to entity set *Studios*, indicating that for a particular star and movie, there is only one studio with which the star has contracted for that movie. However, there are no arrows pointing to entity sets *Stars* or *Movies*. A studio may contract with several stars for a movie, and a star may contract with one studio for more than one movie. □

4.1.8 Roles in Relationships

It is possible that one entity set appears two or more times in a single relationship. If so, we draw as many lines from the relationship to the entity set as the entity set appears in the relationship. Each line to the entity set represents a different *role* that the entity set plays in the relationship. We therefore label the edges between the entity set and relationship by names, which we call "roles."

Example 4.6: In Fig. 4.5 is a relationship *Sequel-of* between the entity set *Movies* and itself. Each relationship is between two movies, one of which is the sequel of the other. To differentiate the two movies in a relationship, one line is labeled by the role *Original* and one by the role *Sequel*, indicating the

Limits on Arrow Notation in Multiway Relationships

There are not enough choices of arrow or no-arrow on the lines attached to a relationship with three or more participants. Thus, we cannot describe every possible situation with arrows. For instance, in Fig. 4.4, the studio is really a function of the movie alone, not the star and movie jointly, since only one studio produces a movie. However, our notation does not distinguish this situation from the case of a three-way relationship where the entity set pointed to by the arrow is truly a function of both other entity sets. To handle all possible situations, we would have to give a set of functional dependencies involving the entity sets of the relationship.

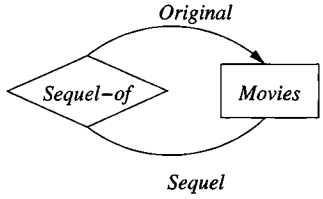

Figure 4.5: A relationship with roles

original movie and its sequel, respectively. We assume that a movie may have many sequels, but for each sequel there is only one original movie. Thus, the relationship is many-one from *Sequel* movies to *Original* movies, as indicated by the arrow in the E/R diagram of Fig. 4.5. □

Example 4.7: As a final example that includes both a multiway relationship and an entity set with multiple roles, in Fig. 4.6 is a more complex version of the *Contracts* relationship introduced earlier in Example 4.5. Now, relationship *Contracts* involves two studios, a star, and a movie. The intent is that one studio, having a certain star under contract (in general, not for a particular movie), may further contract with a second studio to allow that star to act in a particular movie. Thus, the relationship is described by 4-tuples of the form (studio1, studio2, star, movie), meaning that studio2 contracts with studio1 for the use of studio1's star by studio2 for the movie.

We see in Fig. 4.6 arrows pointing to *Studios* in both of its roles, as "owner" of the star and as producer of the movie. However, there are not arrows pointing to *Stars* or *Movies*. The rationale is as follows. Given a star, a movie, and a studio producing the movie, there can be only one studio that "owns" the star. (We assume a star is under contract to exactly one studio.) Similarly, only one studio produces a given movie, so given a star, a movie, and the star's studio, we can determine a unique producing studio. Note that in both

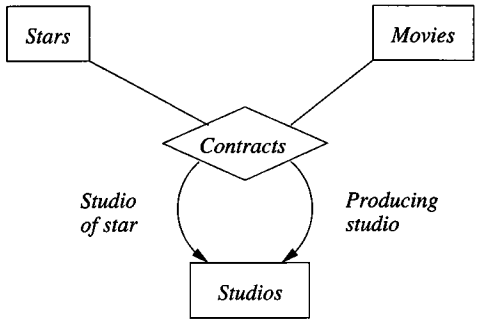

Figure 4.6: A four-way relationship

cases we actually needed only one of the other entities to determine the unique entity—for example, we need only know the movie to determine the unique producing studio—but this fact does not change the multiplicity specification for the multiway relationship.

There are no arrows pointing to *Stars* or *Movies*. Given a star, the star's studio, and a producing studio, there could be several different contracts allowing the star to act in several movies. Thus, the other three components in a relationship 4-tuple do not necessarily determine a unique movie. Similarly, a producing studio might contract with some other studio to use more than one of their stars in one movie. Thus, a star is not determined by the three other components of the relationship. □

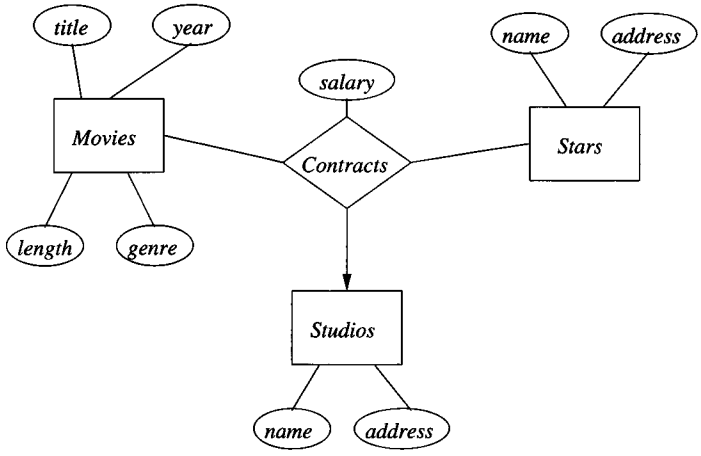

Figure 4.7: A relationship with an attribute

4.1.9 Attributes on Relationships

Sometimes it is convenient, or even essential, to associate attributes with a relationship, rather than with any one of the entity sets that the relationship connects. For example, consider the relationship of Fig. 4.4, which represents contracts between a star and studio for a movie.[1] We might wish to record the salary associated with this contract. However, we cannot associate it with the star; a star might get different salaries for different movies. Similarly, it does not make sense to associate the salary with a studio (they may pay different salaries to different stars) or with a movie (different stars in a movie may receive different salaries).

However, we can associate a unique salary with the (star, movie, studio) triple in the relationship set for the *Contracts* relationship. In Fig. 4.7 we see Fig. 4.4 fleshed out with attributes. The relationship has attribute *salary*, while the entity sets have the same attributes that we showed for them in Fig. 4.2.

In general, we may place one or more attributes on any relationship. The values of these attributes are functionally determined by the entire tuple in the relationship set for that relation. In some cases, the attributes can be determined by a subset of the entity sets involved in the relation, but presumably not by any single entity set (or it would make more sense to place the attribute on that entity set). For instance, in Fig. 4.7, the salary is really determined by the movie and star entities, since the studio entity is itself determined by the movie entity.

It is never necessary to place attributes on relationships. We can instead invent a new entity set, whose entities have the attributes ascribed to the relationship. If we then include this entity set in the relationship, we can omit the attributes on the relationship itself. However, attributes on a relationship are a useful convention, which we shall continue to use where appropriate.

Example 4.8: Let us revise the E/R diagram of Fig. 4.7, which has the salary attribute on the *Contracts* relationship. Instead, we create an entity set *Salaries*, with attribute *salary*. *Salaries* becomes the fourth entity set of relationship *Contracts*. The whole diagram is shown in Fig. 4.8.

Notice that there is an arrow into the *Salaries* entity set in Fig. 4.8. That arrow is appropriate, since we know that the salary is determined by all the other entity sets involved in the relationship. In general, when we do a conversion from attributes on a relationship to an additional entity set, we place an arrow into that entity set. □

4.1.10 Converting Multiway Relationships to Binary

There are some data models, such as UML (Section 4.7) and ODL (Section 4.9), that limit relationships to be binary. Thus, while the E/R model does not

[1] Here, we have reverted to the earlier notion of three-way contracts in Example 4.5, not the four-way relationship of Example 4.7.

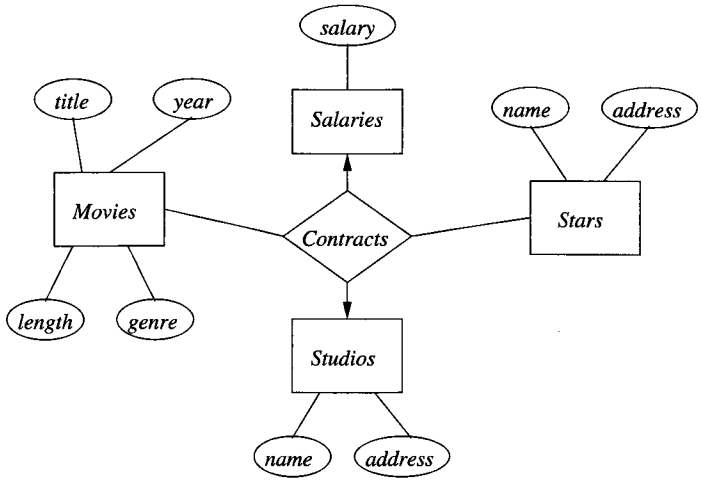

Figure 4.8: Moving the attribute to an entity set

require binary relationships, it is useful to observe that any relationship connecting more than two entity sets can be converted to a collection of binary, many-one relationships. To do so, introduce a new entity set whose entities we may think of as tuples of the relationship set for the multiway relationship. We call this entity set a *connecting* entity set. We then introduce many-one relationships from the connecting entity set to each of the entity sets that provide components of tuples in the original, multiway relationship. If an entity set plays more than one role, then it is the target of one relationship for each role.

Example 4.9: The four-way *Contracts* relationship in Fig. 4.6 can be replaced by an entity set that we may also call *Contracts*. As seen in Fig. 4.9, it participates in four relationships. If the relationship set for the relationship *Contracts* has a 4-tuple (studio1, studio2, star, movie) then the entity set *Contracts* has an entity e. This entity is linked by relationship *Star-of* to the entity *star* in entity set *Stars*. It is linked by relationship *Movie-of* to the entity *movie* in *Movies*. It is linked to entities *studio1* and *studio2* of *Studios* by relationships *Studio-of-star* and *Producing-studio*, respectively.

Note that we have assumed there are no attributes of entity set *Contracts*, although the other entity sets in Fig. 4.9 have unseen attributes. However, it is possible to add attributes, such as the date of signing, to entity set *Contracts*. □

4.1.11 Subclasses in the E/R Model

Often, an entity set contains certain entities that have special properties not associated with all members of the set. If so, we find it useful to define certain

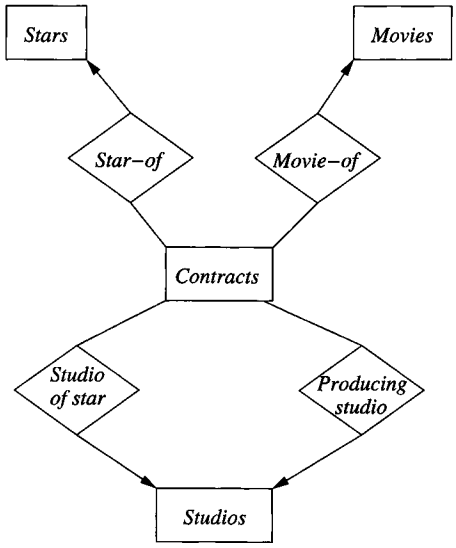

Figure 4.9: Replacing a multiway relationship by an entity set and binary relationships

special-case entity sets, or *subclasses*, each with its own special attributes and/or relationships. We connect an entity set to its subclasses using a relationship called *isa* (i.e., "an A is a B" expresses an "isa" relationship from entity set A to entity set B).

An isa relationship is a special kind of relationship, and to emphasize that it is unlike other relationships, we use a special notation: a triangle. One side of the triangle is attached to the subclass, and the opposite point is connected to the superclass. Every isa relationship is one-one, although we shall not draw the two arrows that are associated with other one-one relationships.

Example 4.10: Among the special kinds of movies we might store in our example database are cartoons and murder mysteries. For each of these special movie types, we could define a subclass of the entity set *Movies*. For instance, let us postulate two subclasses: *Cartoons* and *Murder-Mysteries*. A cartoon has, in addition to the attributes and relationships of *Movies*, an additional relationship called *Voices* that gives us a set of stars who speak, but do not appear in the movie. Movies that are not cartoons do not have such stars. Murder-mysteries have an additional attribute *weapon*. The connections among the three entity sets *Movies*, *Cartoons*, and *Murder-Mysteries* is shown in Fig. 4.10. □

While, in principle, a collection of entity sets connected by *isa* relationships could have any structure, we shall limit isa-structures to trees, in which there

Parallel Relationships Can Be Different

Figure 4.9 illustrates a subtle point about relationships. There are two different relationships, *Studio-of-Star* and *Producing-Studio*, that each connect entity sets *Contracts* and *Studios*. We should not presume that these relationships therefore have the same relationship sets. In fact, in this case, it is unlikely that both relationships would ever relate the same contract to the same studios, since a studio would then be contracting with itself.

More generally, there is nothing wrong with an E/R diagram having several relationships that connect the same entity sets. In the database, the instances of these relationships will normally be different, reflecting the different meanings of the relationships.

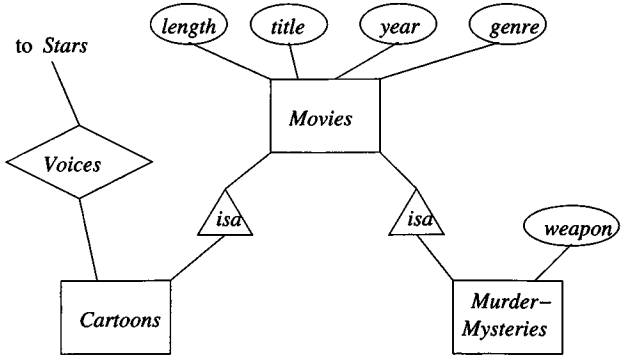

Figure 4.10: Isa relationships in an E/R diagram

is one *root* entity set (e.g., *Movies* in Fig. 4.10) that is the most general, with progressively more specialized entity sets extending below the root in a tree.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, as long as those components are in a subtree including the root. That is, if an entity e has a component c in entity set E, and the parent of E in the tree is F, then entity e also has a component d in F. Further, c and d must be paired in the relationship set for the *isa* relationship from E to F. The entity e has whatever attributes any of its components has, and it participates in whatever relationships any of its components participate in.

Example 4.11: The typical movie, being neither a cartoon nor a murder-mystery, will have a component only in the root entity set *Movies* in Fig. 4.10. These entities have only the four attributes of *Movies* (and the two relationships

The E/R View of Subclasses

There is a significant resemblance between "isa" in the E/R model and subclasses in object-oriented languages. In a sense, "isa" relates a subclass to its superclass. However, there is also a fundamental difference between the conventional E/R view and the object-oriented approach: entities are allowed to have representatives in a tree of entity sets, while objects are assumed to exist in exactly one class or subclass.

The difference becomes apparent when we consider how the movie *Roger Rabbit* was handled in Example 4.11. In an object-oriented approach, we would need for this movie a fourth entity set, "cartoon-murder-mystery," which inherited all the attributes and relationships of *Movies*, *Cartoons*, and *Murder-Mysteries*. However, in the E/R model, the effect of this fourth subclass is obtained by putting components of the movie *Roger Rabbit* in both the *Cartoons* and *Murder-Mysteries* entity sets.

of *Movies* — *Stars-in* and *Owns* — that are not shown in Fig. 4.10).

A cartoon that is not a murder-mystery will have two components, one in *Movies* and one in *Cartoons*. Its entity will therefore have not only the four attributes of *Movies*, but the relationship *Voices*. Likewise, a murder-mystery will have two components for its entity, one in *Movies* and one in *Murder-Mysteries* and thus will have five attributes, including *weapon*.

Finally, a movie like *Roger Rabbit*, which is both a cartoon and a murder-mystery, will have components in all three of the entity sets *Movies*, *Cartoons*, and *Murder-Mysteries*. The three components are connected into one entity by the *isa* relationships. Together, these components give the *Roger Rabbit* entity all four attributes of *Movies* plus the attribute *weapon* of entity set *Murder-Mysteries* and the relationship *Voices* of entity set *Cartoons*. □

4.1.12 Exercises for Section 4.1

Exercise 4.1.1: Design a database for a bank, including information about customers and their accounts. Information about a customer includes their name, address, phone, and Social Security number. Accounts have numbers, types (e.g., savings, checking) and balances. Also record the customer(s) who own an account. Draw the E/R diagram for this database. Be sure to include arrows where appropriate, to indicate the multiplicity of a relationship.

Exercise 4.1.2: Modify your solution to Exercise 4.1.1 as follows:

a) Change your diagram so an account can have only one customer.

b) Further change your diagram so a customer can have only one account.

! c) Change your original diagram of Exercise 4.1.1 so that a customer can have a set of addresses (which are street-city-state triples) and a set of phones. Remember that we do not allow attributes to have nonprimitive types, such as sets, in the E/R model.

! d) Further modify your diagram so that customers can have a set of addresses, and at each address there is a set of phones.

Exercise 4.1.3: Give an E/R diagram for a database recording information about teams, players, and their fans, including:

1. For each team, its name, its players, its team captain (one of its players), and the colors of its uniform.
2. For each player, his/her name.
3. For each fan, his/her name, favorite teams, favorite players, and favorite color.

Remember that a set of colors is not a suitable attribute type for teams. How can you get around this restriction?

Exercise 4.1.4: Suppose we wish to add to the schema of Exercise 4.1.3 a relationship *Led-by* among two players and a team. The intention is that this relationship set consists of triples (player1, player2, team) such that player 1 played on the team at a time when some other player 2 was the team captain.

a) Draw the modification to the E/R diagram.

b) Replace your ternary relationship with a new entity set and binary relationships.

! c) Are your new binary relationships the same as any of the previously existing relationships? Note that we assume the two players are different, i.e., the team captain is not self-led.

Exercise 4.1.5: Modify Exercise 4.1.3 to record for each player the history of teams on which they have played, including the start date and ending date (if they were traded) for each such team.

! **Exercise 4.1.6:** Design a genealogy database with one entity set: *People*. The information to record about persons includes their name (an attribute), their mother, father, and children.

! **Exercise 4.1.7:** Modify your "people" database design of Exercise 4.1.6 to include the following special types of people:

1. Females.

2. Males.

3. People who are parents.

You may wish to distinguish certain other kinds of people as well, so relationships connect appropriate subclasses of people.

Exercise 4.1.8: An alternative way to represent the information of Exercise 4.1.6 is to have a ternary relationship *Family* with the intent that in the relationship set for *Family*, triple (person, mother, father) is a person, their mother, and their father; all three are in the *People* entity set, of course.

a) Draw this diagram, placing arrows on edges where appropriate.

b) Replace the ternary relationship *Family* by an entity set and binary relationships. Again place arrows to indicate the multiplicity of relationships.

Exercise 4.1.9: Design a database suitable for a university registrar. This database should include information about students, departments, professors, courses, which students are enrolled in which courses, which professors are teaching which courses, student grades, TA's for a course (TA's are students), which courses a department offers, and any other information you deem appropriate. Note that this question is more free-form than the questions above, and you need to make some decisions about multiplicities of relationships, appropriate types, and even what information needs to be represented.

! **Exercise 4.1.10:** Informally, we can say that two E/R diagrams "have the same information" if, given a real-world situation, the instances of these two diagrams that reflect this situation can be computed from one another. Consider the E/R diagram of Fig. 4.6. This four-way relationship can be decomposed into a three-way relationship and a binary relationship by taking advantage of the fact that for each movie, there is a unique studio that produces that movie. Give an E/R diagram without a four-way relationship that has the same information as Fig. 4.6.

4.2 Design Principles

We have yet to learn many of the details of the E/R model, but we have enough to begin study of the crucial issue of what constitutes a good design and what should be avoided. In this section, we offer some useful design principles.

4.2.1 Faithfulness

First and foremost, the design should be faithful to the specifications of the application. That is, entity sets and their attributes should reflect reality. You can't attach an attribute *number-of-cylinders* to *Stars*, although that attribute

would make sense for an entity set *Automobiles*. Whatever relationships are asserted should make sense given what we know about the part of the real world being modeled.

Example 4.12: If we define a relationship *Stars-in* between *Stars* and *Movies*, it should be a many-many relationship. The reason is that an observation of the real world tells us that stars can appear in more than one movie, and movies can have more than one star. It is incorrect to declare the relationship *Stars-in* to be many-one in either direction or to be one-one. □

Example 4.13: On the other hand, sometimes it is less obvious what the real world requires us to do in our E/R design. Consider, for instance, entity sets *Courses* and *Instructors*, with a relationship *Teaches* between them. Is *Teaches* many-one from *Courses* to *Instructors*? The answer lies in the policy and intentions of the organization creating the database. It is possible that the school has a policy that there can be only one instructor for any course. Even if several instructors may "team-teach" a course, the school may require that exactly one of them be listed in the database as the instructor responsible for the course. In either of these cases, we would make *Teaches* a many-one relationship from *Courses* to *Instructors*.

Alternatively, the school may use teams of instructors regularly and wish its database to allow several instructors to be associated with a course. Or, the intent of the *Teaches* relationship may not be to reflect the current teacher of a course, but rather those who have ever taught the course, or those who are capable of teaching the course; we cannot tell simply from the name of the relationship. In either of these cases, it would be proper to make *Teaches* be many-many. □

4.2.2 Avoiding Redundancy

We should be careful to say everything once only. The problems we discussed in Section 3.3 regarding redundancy and anomalies are typical of problems that can arise in E/R designs. However, in the E/R model, there are several new mechanisms whereby redundancy and other anomalies can arise.

For instance, we have used a relationship *Owns* between movies and studios. We might also choose to have an attribute *studioName* of entity set *Movies*. While there is nothing illegal about doing so, it is dangerous for several reasons.

1. Doing so leads to repetition of a fact, with the result that extra space is required to represent the data, once we convert the E/R design to a relational (or other type of) concrete implementation.

2. There is an update-anomaly potential, since we might change the relationship but not the attribute, or vice-versa.

We shall say more about avoiding anomalies in Sections 4.2.4 and 4.2.5.

4.2.3 Simplicity Counts

Avoid introducing more elements into your design than is absolutely necessary.

Example 4.14: Suppose that instead of a relationship between *Movies* and *Studios* we postulated the existence of "movie-holdings," the ownership of a single movie. We might then create another entity set *Holdings*. A one-one relationship *Represents* could be established between each movie and the unique holding that represents the movie. A many-one relationship from *Holdings* to *Studios* completes the picture shown in Fig. 4.11.

Figure 4.11: A poor design with an unnecessary entity set

Technically, the structure of Fig. 4.11 truly represents the real world, since it is possible to go from a movie to its unique owning studio via *Holdings*. However, *Holdings* serves no useful purpose, and we are better off without it. It makes programs that use the movie-studio relationship more complicated, wastes space, and encourages errors. □

4.2.4 Choosing the Right Relationships

Entity sets can be connected in various ways by relationships. However, adding to our design every possible relationship is not often a good idea. Doing so can lead to redundancy, update anomalies, and deletion anomalies, where the connected pairs or sets of entities for one relationship can be deduced from one or more other relationships. We shall illustrate the problem and what to do about it with two examples. In the first example, several relationships could represent the same information; in the second, one relationship could be deduced from several others.

Example 4.15: Let us review Fig. 4.7, where we connected movies, stars, and studios with a three-way relationship *Contracts*. We omitted from that figure the two binary relationships *Stars-in* and *Owns* from Fig. 4.2. Do we also need these relationships, between *Movies* and *Stars*, and between *Movies* and *Studios*, respectively? The answer is: "we don't know; it depends on our assumptions regarding the three relationships in question."

It might be possible to deduce the relationship *Stars-in* from *Contracts*. If a star can appear in a movie only if there is a contract involving that star, that movie, and the owning studio for the movie, then there truly is no need for relationship *Stars-in*. We could figure out all the star-movie pairs by looking at the star-movie-studio triples in the relationship set for *Contracts* and taking only the star and movie components, i.e., projecting *Contracts* onto *Stars-in*.

However, if a star can work on a movie without there being a contract — or what is more likely, without there being a contract that we know about in our database — then there could be star-movie pairs in *Stars-in* that are not part of star-movie-studio triples in *Contracts*. In that case, we need to retain the *Stars-in* relationship.

A similar observation applies to relationship *Owns*. If for every movie, there is at least one contract involving that movie, its owning studio, and some star for that movie, then we can dispense with *Owns*. However, if there is the possibility that a studio owns a movie, yet has no stars under contract for that movie, or no such contract is known to our database, then we must retain *Owns*.

In summary, we cannot tell you whether a given relationship will be redundant. You must find out from those who wish the database implemented what to expect. Only then can you make a rational decision about whether or not to include relationships such as *Stars-in* or *Owns*. □

Example 4.16: Now, consider Fig. 4.2 again. In this diagram, there is no relationship between stars and studios. Yet we can use the two relationships *Stars-in* and *Owns* to build a connection by the process of composing those two relationships. That is, a star is connected to some movies by *Stars-in*, and those movies are connected to studios by *Owns*. Thus, we could say that a star is connected to the studios that own movies in which the star has appeared.

Would it make sense to have a relationship *Works-for*, as suggested in Fig. 4.12, between *Stars* and *Studios* too? Again, we cannot tell without knowing more. First, what would the meaning of this relationship be? If it is to mean "the star appeared in at least one movie of this studio," then probably there is no good reason to include it in the diagram. We could deduce this information from *Stars-in* and *Owns* instead.

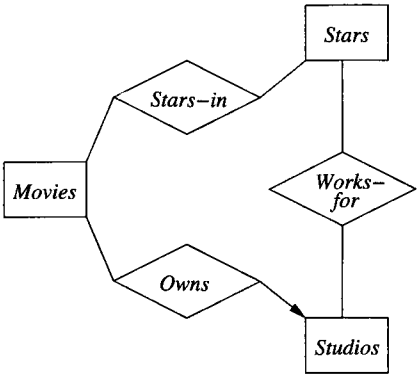

Figure 4.12: Adding a relationship between *Stars* and *Studios*

However, perhaps we have other information about stars working for studios that is not implied by the connection through a movie. In that case, a

relationship connecting stars directly to studios might be useful and would not be redundant. Alternatively, we might use a relationship between stars and studios to mean something entirely different. For example, it might represent the fact that the star is under contract to the studio, in a manner unrelated to any movie. As we suggested in Example 4.7, it is possible for a star to be under contract to one studio and yet work on a movie owned by another studio. In this case, the information found in the new *Works-for* relation would be independent of the *Stars-in* and *Owns* relationships, and would surely be nonredundant. □

4.2.5 Picking the Right Kind of Element

Sometimes we have options regarding the type of design element used to represent a real-world concept. Many of these choices are between using attributes and using entity set/relationship combinations. In general, an attribute is simpler to implement than either an entity set or a relationship. However, making everything an attribute will usually get us into trouble.

Example 4.17: Let us consider a specific problem. In Fig. 4.2, were we wise to make studios an entity set? Should we instead have made the name and address of the studio be attributes of movies and eliminated the *Studio* entity set? One problem with doing so is that we repeat the address of the studio for each movie. We can also have an update anomaly if we change the address for one movie but not another with the same studio, and we can have a deletion anomaly if we delete the last movie owned by a given studio.

On the other hand, if we did not record addresses of studios, then there is no harm in making the studio name an attribute of movies. We have no anomalies in this case. Saying the name of a studio for each movie is not true redundancy, since we must represent the owner of each movie somehow, and saying the name of the studio is a reasonable way to do so. □

We can abstract what we have observed in Example 4.17 to give the conditions under which we prefer to use an attribute instead of an entity set. Suppose E is an entity set. Here are conditions that E must obey in order for us to replace E by an attribute or attributes of several other entity sets.

1. All relationships in which E is involved must have arrows entering E. That is, E must be the "one" in many-one relationships, or its generalization for the case of multiway relationships.

2. If E has more than one attribute, then no attribute depends on the other attributes, the way *address* depends on *name* for *Studios*. That is, the only key for E is all its attributes.

3. No relationship involves E more than once.

If these conditions are met, then we can replace entity set E as follows:

a) If there is a many-one relationship R from some entity set F to E, then remove R and make the attributes of E be attributes of F, suitably renamed if they conflict with attribute names for F. In effect, each F-entity takes, as attributes, the name of the unique, related E-entity.[2] For instance, *Movies* entities could take their studio name as an attribute, should we dispense with studio addresses.

b) If there is a multiway relationship R with an arrow to E, make the attributes of E be attributes of R and delete the arc from R to E. An example of this transformation is replacing Fig. 4.8, where there is an entity set *Salaries* with a number as its lone attribute, by its original diagram in Fig. 4.7.

Example 4.18: Let us consider a point where there is a tradeoff between using a multiway relationship and using a connecting entity set with several binary relationships. We saw a four-way relationship *Contracts* among a star, a movie, and two studios in Fig. 4.6. In Fig. 4.9, we mechanically converted it to an entity set *Contracts*. Does it matter which we choose?

As the problem was stated, either is appropriate. However, should we change the problem just slightly, then we are almost forced to choose a connecting entity set. Let us suppose that contracts involve one star, one movie, but any set of studios. This situation is more complex than the one in Fig. 4.6, where we had two studios playing two roles. In this case, we can have any number of studios involved, perhaps one to do production, one for special effects, one for distribution, and so on. Thus, we cannot assign roles for studios.

It appears that a relationship set for the relationship *Contracts* must contain triples of the form (star, movie, set-of-studios), and the relationship *Contracts* itself involves not only the usual *Stars* and *Movies* entity sets, but a new entity set whose entities are *sets of* studios. While this approach is possible, it seems unnatural to think of sets of studios as basic entities, and we do not recommend it.

A better approach is to think of contracts as an entity set. As in Fig. 4.9, a contract entity connects a star, a movie and a set of studios, but now there must be no limit on the number of studios. Thus, the relationship between contracts and studios is many-many, rather than many-one as it would be if contracts were a true "connecting" entity set. Figure 4.13 sketches the E/R diagram. Note that a contract is related to a single star and to a single movie, but to any number of studios. □

4.2.6 Exercises for Section 4.2

Exercise 4.2.1: In Fig. 4.14 is an E/R diagram for a bank database involving customers and accounts. Since customers may have several accounts, and

[2] In a situation where an F-entity is not related to any E-entity, the new attributes of F would be given special "null" values to indicate the absence of a related E-entity. A similar arrangement would be used for the new attributes of R in case (b).

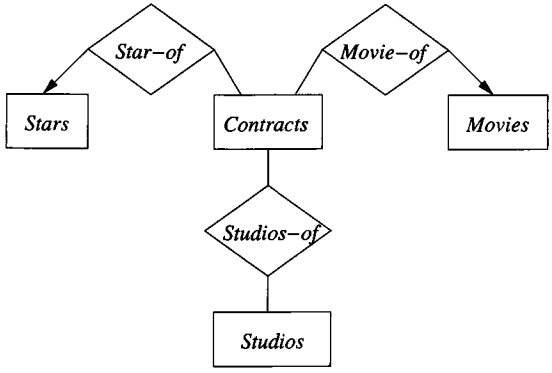

Figure 4.13: Contracts connecting a star, a movie, and a set of studios

accounts may be held jointly by several customers, we associate with each customer an "account set," and accounts are members of one or more account sets. Assuming the meaning of the various relationships and attributes are as expected given their names, criticize the design. What design rules are violated? Why? What modifications would you suggest?

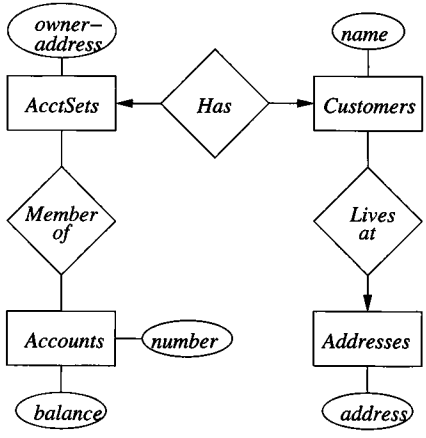

Figure 4.14: A poor design for a bank database

Exercise 4.2.2: Under what circumstances (regarding the unseen attributes of *Studios* and *Presidents*) would you recommend combining the two entity sets and relationship in Fig. 4.3 into a single entity set and attributes?

Exercise 4.2.3: Suppose we delete the attribute *address* from *Studios* in Fig. 4.7. Show how we could then replace an entity set by an attribute. Where

would that attribute appear?

Exercise 4.2.4: Give choices of attributes for the following entity sets in Fig. 4.13 that will allow the entity set to be replaced by an attribute:

a) *Stars.*

b) *Movies.*

! c) *Studios.*

!! **Exercise 4.2.5:** In this and following exercises we shall consider two design options in the E/R model for describing births. At a birth, there is one baby (twins would be represented by two births), one mother, any number of nurses, and any number of doctors. Suppose, therefore, that we have entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors*. Suppose we also use a relationship *Births*, which connects these four entity sets, as suggested in Fig. 4.15. Note that a tuple of the relationship set for *Births* has the form (baby, mother, nurse, doctor). If there is more than one nurse and/or doctor attending a birth, then there will be several tuples with the same baby and mother, one for each combination of nurse and doctor.

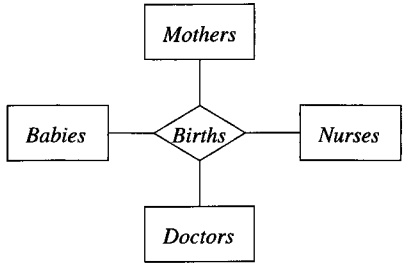

Figure 4.15: Representing births by a multiway relationship

There are certain assumptions that we might wish to incorporate into our design. For each, tell how to add arrows or other elements to the E/R diagram in order to express the assumption.

a) For every baby, there is a unique mother.

b) For every combination of a baby, nurse, and doctor, there is a unique mother.

c) For every combination of a baby and a mother there is a unique doctor.

! **Exercise 4.2.6:** Another approach to the problem of Exercise 4.2.5 is to connect the four entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors* by an entity set

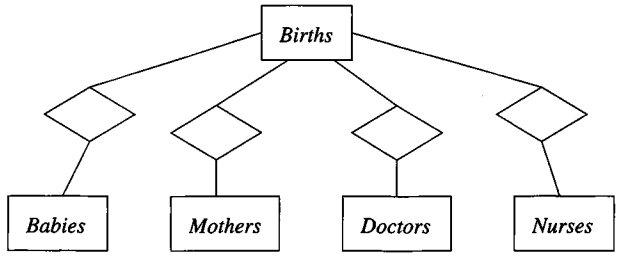

Figure 4.16: Representing births by an entity set

Births, with four relationships, one between *Births* and each of the other entity sets, as suggested in Fig. 4.16. Use arrows (indicating that certain of these relationships are many-one) to represent the following conditions:

a) Every baby is the result of a unique birth, and every birth is of a unique baby.

b) In addition to (a), every baby has a unique mother.

c) In addition to (a) and (b), for every birth there is a unique doctor.

In each case, what design flaws do you see?

!! **Exercise 4.2.7:** Suppose we change our viewpoint to allow a birth to involve more than one baby born to one mother. How would you represent the fact that every baby still has a unique mother using the approaches of Exercises 4.2.5 and 4.2.6?

4.3 Constraints in the E/R Model

The E/R model has several ways to express the common kinds of constraints on the data that will populate the database being designed. Like the relational model, there is a way to express the idea that an attribute or attributes are a key for an entity set. We have already seen how an arrow connecting a relationship to an entity set serves as a "functional dependency." There is also a way to express a referential-integrity constraint, where an entity in one set is required to have an entity in another set to which it is related.

4.3.1 Keys in the E/R Model

A *key* for an entity set E is a set K of one or more attributes such that, given any two distinct entities e_1 and e_2 in E, e_1 and e_2 cannot have identical values for each of the attributes in the key K. If K consists of more than one attribute, then it is possible for e_1 and e_2 to agree in some of these attributes, but never in all attributes. Some important points to remember are:

- Every entity set must have a key, although in some cases — isa-hierarchies and "weak" entity sets (see Section 4.4), the key actually belongs to another entity set.

- There can be more than one possible key for an entity set. However, it is customary to pick one key as the "primary key," and to act as if that were the only key.

- When an entity set is involved in an isa-hierarchy, we require that the root entity set have all the attributes needed for a key, and that the key for each entity is found from its component in the root entity set, regardless of how many entity sets in the hierarchy have components for the entity.

In our running movies example, we have used *title* and *year* as the key for *Movies*, counting on the observation that it is unlikely that two movies with the same title would be released in one year. We also decided that it was safe to use *name* as a key for *MovieStar*, believing that no real star would ever want to use the name of another star.

4.3.2 Representing Keys in the E/R Model

In our E/R-diagram notation, we underline the attributes belonging to a key for an entity set. For example, Fig. 4.17 reproduces our E/R diagram for movies, stars, and studios from Fig. 4.2, but with key attributes underlined. Attribute *name* is the key for *Stars*. Likewise, *Studios* has a key consisting of only its own attribute *name*.

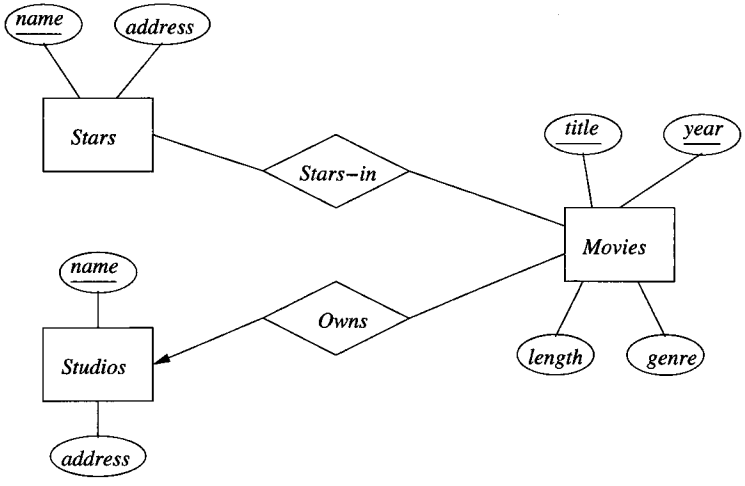

Figure 4.17: E/R diagram; keys are indicated by underlines

The attributes *title* and *year* together form the key for *Movies*. Note that when several attributes are underlined, as in Fig. 4.17, then they are each members of the key. There is no notation for representing the situation where there are several keys for an entity set; we underline only the primary key. You should also be aware that in some unusual situations, the attributes forming the key for an entity set do not all belong to the entity set itself. We shall defer this matter, called "weak entity sets," until Section 4.4.

4.3.3 Referential Integrity

Recall our discussion of referential-integrity constraints in Section 2.5.2. These constraints say that a value appearing in one context must also appear in another. For example, let us consider the many-one relationship *Owns* from *Movies* to *Studios* in Fig. 4.2. The many-one requirement simply says that no movie can be owned by more than one studio. It does *not* say that a movie must surely be owned by a studio, or that the owning studio must be present in the *Studios* entity set, as stored in our database. An appropriate referential integrity constraint on relationship *Owns* is that for each movie, the owning studio (the entity "referenced" by the relationship for this movie) must exist in our database.

The arrow notation in E/R diagrams is able to indicate whether a relationship is expected to support referential integrity in one or more directions. Suppose R is a relationship from entity set E to entity set F. A rounded arrowhead pointing to F indicates not only that the relationship is many-one from E to F, but that the entity of set F related to a given entity of set E is required to exist. The same idea applies when R is a relationship among more than two entity sets.

Example 4.19: Figure 4.18 shows some appropriate referential integrity constraints among the entity sets *Movies*, *Studios*, and *Presidents*. These entity sets and relationships were first introduced in Figs. 4.2 and 4.3. We see a rounded arrow entering *Studios* from relationship *Owns*. That arrow expresses the referential integrity constraint that every movie must be owned by one studio, and this studio is present in the *Studios* entity set.

Figure 4.18: E/R diagram showing referential integrity constraints

Similarly, we see a rounded arrow entering *Studios* from *Runs*. That arrow expresses the referential integrity constraint that every president runs a studio that exists in the *Studios* entity set.

Note that the arrow to *Presidents* from *Runs* remains a pointed arrow. That choice reflects a reasonable assumption about the relationship between studios

and their presidents. If a studio ceases to exist, its president can no longer be called a president, so we would expect the president of the studio to be deleted from the entity set *Presidents*. Hence there is a rounded arrow to *Studios*. On the other hand, if a president were fired or resigned, the studio would continue to exist. Thus, we place an ordinary, pointed arrow to *Presidents*, indicating that each studio has at most one president, but might have no president at some time. □

4.3.4 Degree Constraints

In the E/R model, we can attach a bounding number to the edges that connect a relationship to an entity set, indicating limits on the number of entities that can be connected to any one entity of the related entity set. For example, we could choose to place a constraint on the degree of a relationship, such as that a movie entity cannot be connected by relationship *Stars-in* to more than 10 star entities.

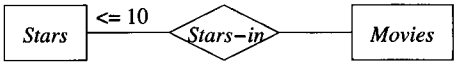

Figure 4.19: Representing a constraint on the number of stars per movie

Figure 4.19 shows how we can represent this constraint. As another example, we can think of the arrow as a synonym for the constraint "≤ 1," and we can think of the rounded arrow of Fig. 4.18 as standing for the constraint "$= 1$."

4.3.5 Exercises for Section 4.3

Exercise 4.3.1: For your E/R diagrams of:

a) Exercise 4.1.1.

b) Exercise 4.1.3.

c) Exercise 4.1.6.

(*i*) Select and specify keys, and (*ii*) Indicate appropriate referential integrity constraints.

! **Exercise 4.3.2:** We may think of relationships in the E/R model as having keys, just as entity sets do. Let R be a relationship among the entity sets $E_1, E_2, \ldots, E_n$. Then a *key* for R is a set K of attributes chosen from the attributes of $E_1, E_2, \ldots, E_n$ such that if $(e_1, e_2, \ldots, e_n)$ and $(f_1, f_2, \ldots, f_n)$ are two different tuples in the relationship set for R, then it is not possible that these tuples agree in all the attributes of K. Now, suppose $n = 2$; that is, R is a binary relationship. Also, for each i, let K_i be a set of attributes that is a key for entity set E_i. In terms of E_1 and E_2, give a smallest possible key for R under the assumption that:

a) R is many-many.

b) R is many-one from E_1 to E_2.

c) R is many-one from E_2 to E_1.

d) R is one-one.

!! **Exercise 4.3.3:** Consider again the problem of Exercise 4.3.2, but with n allowed to be any number, not just 2. Using only the information about which arcs from R to the E_i's have arrows, show how to find a smallest possible key K for R in terms of the K_i's.

4.4 Weak Entity Sets

It is possible for an entity set's key to be composed of attributes, some or all of which belong to another entity set. Such an entity set is called a *weak entity set.*

4.4.1 Causes of Weak Entity Sets

There are two principal reasons we need weak entity sets. First, sometimes entity sets fall into a hierarchy based on classifications unrelated to the "isa hierarchy" of Section 4.1.11. If entities of set E are subunits of entities in set F, then it is possible that the names of E-entities are not unique until we take into account the name of the F-entity to which the E entity is subordinate. Several examples will illustrate the problem.

Example 4.20: A movie studio might have several film crews. The crews might be designated by a given studio as crew 1, crew 2, and so on. However, other studios might use the same designations for crews, so the attribute *number* is not a key for crews. Rather, to name a crew uniquely, we need to give both the name of the studio to which it belongs and the number of the crew. The situation is suggested by Fig. 4.20. The double-rectangle indicates a weak entity set, and the double-diamond indicates a many-one relationship that helps provide the key for the weak entity set. The notation will be explained further in Section 4.4.3. The key for weak entity set *Crews* is its own *number* attribute and the *name* attribute of the unique studio to which the crew is related by the many-one *Unit-of* relationship. □

Example 4.21: A species is designated by its genus and species names. For example, humans are of the species *Homo sapiens*; *Homo* is the genus name and *sapiens* the species name. In general, a genus consists of several species, each of which has a name beginning with the genus name and continuing with the species name. Unfortunately, species names, by themselves, are not unique.

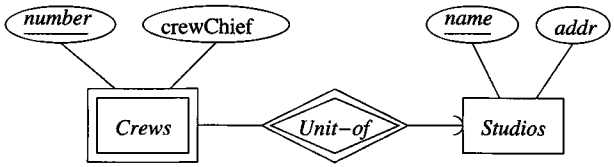

Figure 4.20: A weak entity set for crews, and its connections

Two or more genera may have species with the same species name. Thus, to designate a species uniquely we need both the species name and the name of the genus to which the species is related by the *Belongs-to* relationship, as suggested in Fig. 4.21. *Species* is a weak entity set whose key comes partially from its genus. □

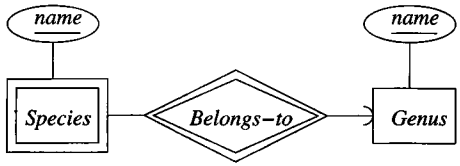

Figure 4.21: Another weak entity set, for species

The second common source of weak entity sets is the connecting entity sets that we introduced in Section 4.1.10 as a way to eliminate a multiway relationship.[3] These entity sets often have no attributes of their own. Their key is formed from the attributes that are the key attributes for the entity sets they connect.

Example 4.22: In Fig. 4.22 we see a connecting entity set *Contracts* that replaces the ternary relationship *Contracts* of Example 4.5. *Contracts* has an attribute *salary*, but this attribute does not contribute to the key. Rather, the key for a contract consists of the name of the studio and the star involved, plus the title and year of the movie involved. □

4.4.2 Requirements for Weak Entity Sets

We cannot obtain key attributes for a weak entity set indiscriminately. Rather, if E is a weak entity set then its key consists of:

1. Zero or more of its own attributes, and

[3]Remember that there is no particular requirement in the E/R model that multiway relationships be eliminated, although this requirement exists in some other database design models.

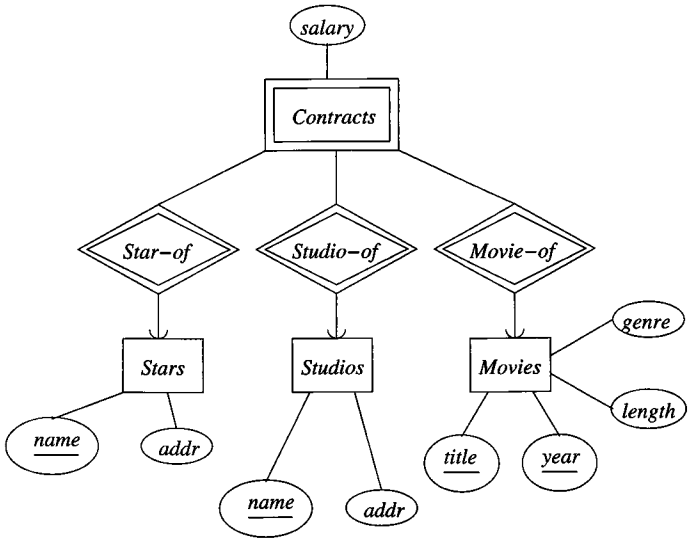

Figure 4.22: Connecting entity sets are weak

2. Key attributes from entity sets that are reached by certain many-one relationships from E to other entity sets. These many-one relationships are called *supporting relationships* for E, and the entity sets reached from E are *supporting entity sets.*

In order for R, a many-one relationship from E to some entity set F, to be a supporting relationship for E, the following conditions must be obeyed:

a) R must be a binary, many-one relationship[4] from E to F.

b) R must have referential integrity from E to F. That is, for every E-entity, there must be exactly one existing F-entity related to it by R. Put another way, a rounded arrow from R to F must be justified.

c) The attributes that F supplies for the key of E must be key attributes of F.

d) However, if F is itself weak, then some or all of the key attributes of F supplied to E will be key attributes of one or more entity sets G to which F is connected by a supporting relationship. Recursively, if G is weak, some key attributes of G will be supplied from elsewhere, and so on.

[4]Remember that a one-one relationship is a special case of a many-one relationship. When we say a relationship must be many-one, we always include one-one relationships as well.

e) If there are several different supporting relationships from E to the same entity set F, then each relationship is used to supply a copy of the key attributes of F to help form the key of E. Note that an entity e from E may be related to different entities in F through different supporting relationships from E. Thus, the keys of several different entities from F may appear in the key values identifying a particular entity e from E.

The intuitive reason why these conditions are needed is as follows. Consider an entity in a weak entity set, say a crew in Example 4.20. Each crew is unique, abstractly. In principle we can tell one crew from another, even if they have the same number but belong to different studios. It is only the data about crews that makes it hard to distinguish crews, because the number alone is not sufficient. The only way we can associate additional information with a crew is if there is some deterministic process leading to additional values that make the designation of a crew unique. But the only unique values associated with an abstract crew entity are:

1. Values of attributes of the *Crews* entity set, and
2. Values obtained by following a relationship from a crew entity to a unique entity of some other entity set, where that other entity has a unique associated value of some kind. That is, the relationship followed must be many-one to the other entity set F, and the associated value must be part of a key for F.

4.4.3 Weak Entity Set Notation

We shall adopt the following conventions to indicate that an entity set is weak and to declare its key attributes.

1. If an entity set is weak, it will be shown as a rectangle with a double border. Examples of this convention are *Crews* in Fig. 4.20 and *Contracts* in Fig. 4.22.
2. Its supporting many-one relationships will be shown as diamonds with a double border. Examples of this convention are *Unit-of* in Fig. 4.20 and all three relationships in Fig. 4.22.
3. If an entity set supplies any attributes for its own key, then those attributes will be underlined. An example is in Fig. 4.20, where the number of a crew participates in its own key, although it is not the complete key for *Crews*.

We can summarize these conventions with the following rule:

- Whenever we use an entity set E with a double border, it is weak. The key for E is whatever attributes of E are underlined plus the key attributes of those entity sets to which E is connected by many-one relationships with a double border.

We should remember that the double-diamond is used only for supporting relationships. It is possible for there to be many-one relationships from a weak entity set that are not supporting relationships, and therefore do not get a double diamond.

Example 4.23: In Fig. 4.22, the relationship *Studio-of* need not be a supporting relationship for *Contracts*. The reason is that each movie has a unique owning studio, determined by the (not shown) many-one relationship from *Movies* to *Studios*. Thus, if we are told the name of a star and a movie, there is at most one contract with any studio for the work of that star in that movie. In terms of our notation, it would be appropriate to use an ordinary single diamond, rather than the double diamond, for *Studio-of* in Fig. 4.22. □

4.4.4 Exercises for Section 4.4

Exercise 4.4.1: One way to represent students and the grades they get in courses is to use entity sets corresponding to students, to courses, and to "enrollments." Enrollment entities form a "connecting" entity set between students and courses and can be used to represent not only the fact that a student is taking a certain course, but the grade of the student in the course. Draw an E/R diagram for this situation, indicating weak entity sets and the keys for the entity sets. Is the grade part of the key for enrollments?

Exercise 4.4.2: Modify your solution to Exercise 4.4.1 so that we can record grades of the student for each of several assignments within a course. Again, indicate weak entity sets and keys.

Exercise 4.4.3: For your E/R diagrams of Exercise 4.2.6(a)–(c), indicate weak entity sets, supporting relationships, and keys.

Exercise 4.4.4: Draw E/R diagrams for the following situations involving weak entity sets. In each case indicate keys for entity sets.

a) Entity sets *Courses* and *Departments*. A course is given by a unique department, but its only attribute is its number. Different departments can offer courses with the same number. Each department has a unique name.

! b) Entity sets *Leagues*, *Teams*, and *Players*. League names are unique. No league has two teams with the same name. No team has two players with the same number. However, there can be players with the same number on different teams, and there can be teams with the same name in different leagues.

4.5 From E/R Diagrams to Relational Designs

To a first approximation, converting an E/R design to a relational database schema is straightforward:

- Turn each entity set into a relation with the same set of attributes, and
- Replace a relationship by a relation whose attributes are the keys for the connected entity sets.

While these two rules cover much of the ground, there are also several special situations that we need to deal with, including:

1. Weak entity sets cannot be translated straightforwardly to relations.
2. "Isa" relationships and subclasses require careful treatment.
3. Sometimes, we do well to combine two relations, especially the relation for an entity set E and the relation that comes from a many-one relationship from E to some other entity set.

4.5.1 From Entity Sets to Relations

Let us first consider entity sets that are not weak. We shall take up the modifications needed to accommodate weak entity sets in Section 4.5.4. For each non-weak entity set, we shall create a relation of the same name and with the same set of attributes. This relation will not have any indication of the relationships in which the entity set participates; we'll handle relationships with separate relations, as discussed in Section 4.5.2.

Example 4.24: Consider the three entity sets *Movies*, *Stars* and *Studios* from Fig. 4.17, which we reproduce here as Fig. 4.23. The attributes for the *Movies* entity set are *title*, *year*, *length*, and *genre*. As a result, this relation `Movies` looks just like the relation `Movies` of Fig. 2.1 with which we began Section 2.2.

Next, consider the entity set *Stars* from Fig. 4.23. There are two attributes, *name* and *address*. Thus, we would expect the corresponding `Stars` relation to have schema `Stars(name, address)` and for

name	*address*
`Carrie Fisher`	`123 Maple St., Hollywood`
`Mark Hamill`	`456 Oak Rd., Brentwood`
`Harrison Ford`	`789 Palm Dr., Beverly Hills`

to be a typical instance. □

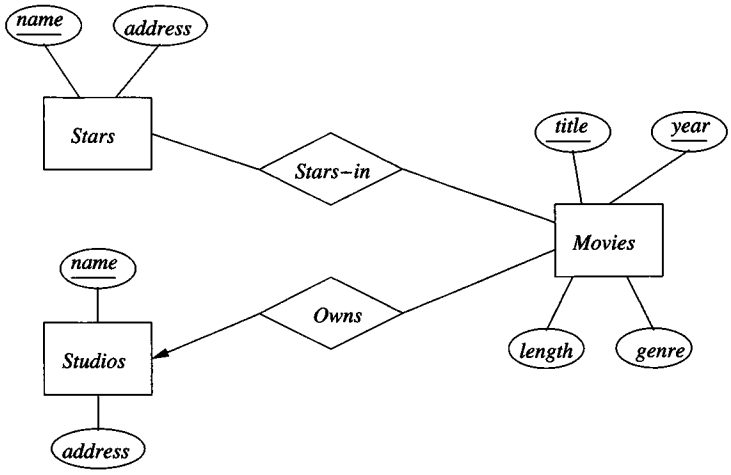

Figure 4.23: E/R diagram for the movie database

4.5.2 From E/R Relationships to Relations

Relationships in the E/R model are also represented by relations. The relation for a given relationship R has the following attributes:

1. For each entity set involved in relationship R, we take its key attribute or attributes as part of the schema of the relation for R.

2. If the relationship has attributes, then these are also attributes of relation R.

If one entity set is involved several times in a relationship, in different roles, then its key attributes each appear as many times as there are roles. We must rename the attributes to avoid name duplication. More generally, should the same attribute name appear twice or more among the attributes of R itself and the keys of the entity sets involved in relationship R, then we need to rename to avoid duplication.

Example 4.25: Consider the relationship *Owns* of Fig. 4.23. This relationship connects entity sets *Movies* and *Studios*. Thus, for the schema of relation `Owns` we use the key for *Movies*, which is *title* and *year*, and the key of *Studios*, which is *name*. That is, the schema for relation `Owns` is:

```
Owns(title, year, studioName)
```

A sample instance of this relation is:

title	*year*	*studioName*
Star Wars	1977	Fox
Gone With the Wind	1939	MGM
Wayne's World	1992	Paramount

We have chosen the attribute `studioName` for clarity; it corresponds to the attribute *name* of *Studios*. □

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

Figure 4.24: A relation for relationship *Stars-In*

Example 4.26: Similarly, the relationship *Stars-In* of Fig. 4.23 can be transformed into a relation with the attributes `title` and `year` (the key for *Movies*) and attribute `starName`, which is the key for entity set *Stars*. Figure 4.24 shows a sample relation `Stars-In`. □

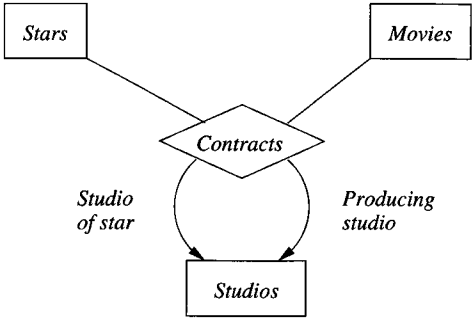

Figure 4.25: The relationship *Contracts*

Example 4.27: Multiway relationships are also easy to convert to relations. Consider the four-way relationship *Contracts* of Fig. 4.6, reproduced here as Fig. 4.25, involving a star, a movie, and two studios — the first holding the star's contract and the second contracting for that star's services in that movie. We represent this relationship by a relation `Contracts` whose schema consists of the attributes from the keys of the following four entity sets:

1. The key `starName` for the star.
2. The key consisting of attributes `title` and `year` for the movie.
3. The key `studioOfStar` indicating the name of the first studio; recall we assume the studio name is a key for the entity set `Studios`.
4. The key `producingStudio` indicating the name of the studio that will produce the movie using that star.

That is, the relation schema is:

```
Contracts(starName, title, year, studioOfStar, producingStudio)
```

Notice that we have been inventive in choosing attribute names for our relation schema, avoiding "name" for any attribute, since it would be unobvious whether that referred to a star's name or studio's name, and in the latter case, which studio role. Also, were there attributes attached to entity set *Contracts*, such as *salary*, these attributes would be added to the schema of relation `Contracts`. □

4.5.3 Combining Relations

Sometimes, the relations that we get from converting entity sets and relationships to relations are not the best possible choice of relations for the given data. One common situation occurs when there is an entity set E with a many-one relationship R from E to F. The relations from E and R will each have the key for E in their relation schema. In addition, the relation for E will have in its schema the attributes of E that are not in the key, and the relation for R will have the key attributes of F and any attributes of R itself. Because R is many-one, all these attributes are functionally determined by the key for E, and we can combine them into one relation with a schema consisting of:

1. All attributes of E.
2. The key attributes of F.
3. Any attributes belonging to relationship R.

For an entity e of E that is not related to any entity of F, the attributes of types (2) and (3) will have null values in the tuple for e.

Example 4.28: In our running movie example, *Owns* is a many-one relationship from *Movies* to *Studios*, which we converted to a relation in Example 4.25. The relation obtained from entity set *Movies* was discussed in Example 4.24. We can combine these relations by taking all their attributes and forming one relation schema. If we do, the relation looks like that in Fig. 4.26. □

title	*year*	*length*	*genre*	*studioName*
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	239	drama	MGM
Wayne's World	1992	95	comedy	Paramount

Figure 4.26: Combining relation Movies with relation Owns

Whether or not we choose to combine relations in this manner is a matter of judgement. However, there are some advantages to having all the attributes that are dependent on the key of entity set E together in one relation, even if there are a number of many-one relationships from E to other entity sets. For example, it is often more efficient to answer queries involving attributes of one relation than to answer queries involving attributes of several relations. In fact, some design systems based on the E/R model combine these relations automatically.

On the other hand, one might wonder if it made sense to combine the relation for E with the relation of a relationship R that involved E but was not many-one from E to some other entity set. Doing so is risky, because it often leads to redundancy, as the next example shows.

Example 4.29: To get a sense of what can go wrong, suppose we combined the relation of Fig. 4.26 with the relation that we get for the many-many relationship *Stars-in*; recall this relation was suggested by Fig. 4.24. Then the combined relation would look like Fig. 3.2, which we reproduce here as Fig. 4.27. As we discussed in Section 3.3.1, this relation has anomalies that we need to remove by the process of normalization. □

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 4.27: The relation Movies with star information

4.5.4 Handling Weak Entity Sets

When a weak entity set appears in an E/R diagram, we need to do three things differently.

1. The relation for the weak entity set W itself must include not only the attributes of W but also the key attributes of the supporting entity sets. The supporting entity sets are easily recognized because they are reached by supporting (double-diamond) relationships from W.

2. The relation for any relationship in which the weak entity set W appears must use as a key for W all of its key attributes, including those of other entity sets that contribute to W's key.

3. However, a supporting relationship R, from the weak entity set W to a supporting entity set, need not be converted to a relation at all. The justification is that, as discussed in Section 4.5.3, the attributes of many-one relationship R's relation will either *be* attributes of the relation for W, or (in the case of attributes on R) can be added to the schema for W's relation.

Of course, when introducing additional attributes to build the key of a weak entity set, we must be careful not to use the same name twice. If necessary, we rename some or all of these attributes.

Example 4.30: Let us consider the weak entity set *Crews* from Fig. 4.20, which we reproduce here as Fig. 4.28. From this diagram we get three relations, whose schemas are:

```
Studios(name, addr)
Crews(number, studioName, crewChief)
Unit-of(number, studioName, name)
```

The first relation, `Studios`, is constructed in a straightforward manner from the entity set of the same name. The second, `Crews`, comes from the weak entity set *Crews*. The attributes of this relation are the key attributes of *Crews* and the one nonkey attribute of *Crews*, which is *crewChief*. We have chosen `studioName` as the attribute in relation `Crews` that corresponds to the attribute *name* in the entity set *Studios*.

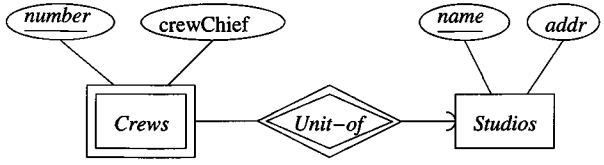

Figure 4.28: The crews example of a weak entity set

The third relation, `Unit-of`, comes from the relationship of the same name. As always, we represent an E/R relationship in the relational model by a relation whose schema has the key attributes of the related entity sets. In this case,

`Unit-of` has attributes `number` and `studioName`, the key for weak entity set *Crews*, and attribute `name`, the key for entity set *Studios*. However, notice that since *Unit-of* is a many-one relationship, the studio `studioName` is surely the same as the studio `name`.

For instance, suppose Disney crew #3 is one of the crews of the Disney studio. Then the relationship set for E/R relationship *Unit-of* includes the pair

(Disney-crew-#3, Disney)

This pair gives rise to the tuple

(3, Disney, Disney)

for the relation `Unit-of`.

Notice that, as must be the case, the components of this tuple for attributes `studioName` and `name` are identical. As a consequence, we can "merge" the attributes `studioName` and `name` of `Unit-of`, giving us the simpler schema:

```
Unit-of(number, name)
```

However, now we can dispense with the relation `Unit-of` altogether, since its attributes are now a subset of the attributes of relation `Crews`. □

The phenomenon observed in Example 4.30 — that a supporting relationship needs no relation — is universal for weak entity sets. The following is a modified rule for converting to relations entity sets that are weak.

- If W is a weak entity set, construct for W a relation whose schema consists of:
  1. All attributes of W.
  2. All attributes of supporting relationships for W.
  3. For each supporting relationship for W, say a many-one relationship from W to entity set E, all the *key* attributes of E.

  Rename attributes, if necessary, to avoid name conflicts.
- Do *not* construct a relation for any supporting relationship for W.

4.5.5 Exercises for Section 4.5

Exercise 4.5.1: Convert the E/R diagram of Fig. 4.29 to a relational database schema.

! **Exercise 4.5.2:** There is another E/R diagram that could describe the weak entity set *Bookings* in Fig. 4.29. Notice that a booking can be identified uniquely by the flight number, day of the flight, the row, and the seat; the customer is not then necessary to help identify the booking.

Relations With Subset Schemas

You might imagine from Example 4.30 that whenever one relation R has a set of attributes that is a subset of the attributes of another relation S, we can eliminate R. That is not exactly true. R might hold information that doesn't appear in S because the additional attributes of S do not allow us to extend a tuple from R to S.

For instance, the Internal Revenue Service tries to maintain a relation `People(name, ss#)` of potential taxpayers and their social-security numbers, even if the person had no income and did not file a tax return. They might also maintain a relation `TaxPayers(name, ss#, amount)` indicating the amount of tax paid by each person who filed a return in the current year. The schema of `People` is a subset of the schema of `TaxPayers`, yet there may be value in remembering the social-security number of those who are mentioned in `People` but not in `Taxpayers`.

In fact, even identical sets of attributes may have different semantics, so it is not possible to merge their tuples. An example would be two relations `Stars(name, addr)` and `Studios(name, addr)`. Although the schemas look alike, we cannot turn star tuples into studio tuples, or vice-versa.

On the other hand, when the two relations come from the weak-entity-set construction, then there can be no such additional value to the relation with the smaller set of attributes. The reason is that the tuples of the relation that comes from the supporting relationship correspond one-for-one with the tuples of the relation that comes from the weak entity set. Thus, we routinely eliminate the former relation.

a) Revise the diagram of Fig. 4.29 to reflect this new viewpoint.

b) Convert your diagram from (a) into relations. Do you get the same database schema as in Exercise 4.5.1?

Exercise 4.5.3: The E/R diagram of Fig. 4.30 represents ships. Ships are said to be *sisters* if they were designed from the same plans. Convert this diagram to a relational database schema.

Exercise 4.5.4: Convert the following E/R diagrams to relational database schemas.

a) Figure 4.22.

b) Your answer to Exercise 4.4.1.

c) Your answer to Exercise 4.4.4(a).

d) Your answer to Exercise 4.4.4(b).

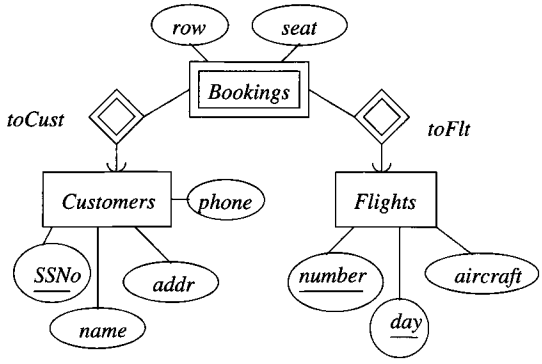

Figure 4.29: An E/R diagram about airlines

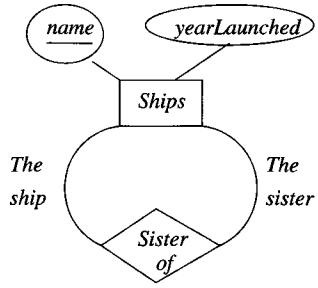

Figure 4.30: An E/R diagram about sister ships

4.6 Converting Subclass Structures to Relations

When we have an isa-hierarchy of entity sets, we are presented with several choices of strategy for conversion to relations. Recall we assume that:

- There is a root entity set for the hierarchy,
- This entity set has a key that serves to identify every entity represented by the hierarchy, and
- A given entity may have *components* that belong to the entity sets of any subtree of the hierarchy, as long as that subtree includes the root.

The principal conversion strategies are:

1. *Follow the E/R viewpoint.* For each entity set E in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to E.

2. *Treat entities as objects belonging to a single class.* For each possible subtree that includes the root, create one relation, whose schema includes all the attributes of all the entity sets in the subtree.

3. *Use null values.* Create one relation with all the attributes of all the entity sets in the hierarchy. Each entity is represented by one tuple, and that tuple has a null value for whatever attributes the entity does not have.

We shall consider each approach in turn.

4.6.1 E/R-Style Conversion

Our first approach is to create a relation for each entity set, as usual. If the entity set E is not the root of the hierarchy, then the relation for E will include the key attributes at the root, to identify the entity represented by each tuple, plus all the attributes of E. In addition, if E is involved in a relationship, then we use these key attributes to identify entities of E in the relation corresponding to that relationship.

Note, however, that although we spoke of "isa" as a relationship, it is unlike other relationships, in that it connects components of a single entity, not distinct entities. Thus, we do not create a relation for "isa."

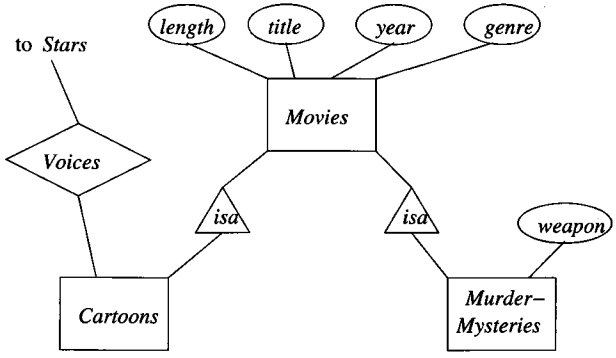

Figure 4.31: The movie hierarchy

Example 4.31: Consider the hierarchy of Fig. 4.10, which we reproduce here as Fig. 4.31. The relations needed to represent the entity sets in this hierarchy are:

1. `Movies(title, year, length, genre)`. This relation was discussed in Example 4.24, and every movie is represented by a tuple here.

2. `MurderMysteries(title, year, weapon)`. The first two attributes are the key for all movies, and the last is the lone attribute for the corresponding entity set. Those movies that are murder mysteries have a tuple here as well as in `Movies`.

3. `Cartoons(title, year)`. This relation is the set of cartoons. It has no attributes other than the key for movies, since the extra information about cartoons is contained in the relationship *Voices*. Movies that are cartoons have a tuple here as well as in `Movies`.

Note that the fourth kind of movie — those that are both cartoons and murder mysteries — have tuples in all three relations.

In addition, we shall need the relation `Voices(title, year, starName)` that corresponds to the relationship *Voices* between *Stars* and *Cartoons*. The last attribute is the key for *Stars* and the first two form the key for *Cartoons*.

For instance, the movie *Roger Rabbit* would have tuples in all four relations. Its basic information would be in `Movies`, the murder weapon would appear in `MurderMysteries`, and the stars that provided voices for the movie would appear in `Voices`.

Notice that the relation `Cartoons` has a schema that is a subset of the schema for the relation `Voices`. In many situations, we would be content to eliminate a relation such as `Cartoons`, since it appears not to contain any information beyond what is in `Voices`. However, there may be silent cartoons in our database. Those cartoons would have no voices, and we would therefore lose information should we eliminate relation `Cartoons`. □

4.6.2 An Object-Oriented Approach

An alternative strategy for converting isa-hierarchies to relations is to enumerate all the possible subtrees of the hierarchy. For each, create one relation that represents entities having components in exactly those subtrees. The schema for this relation has all the attributes of any entity set in the subtree. We refer to this approach as "object-oriented," since it is motivated by the assumption that entities are "objects" that belong to one and only one class.

Example 4.32: Consider the hierarchy of Fig. 4.31. There are four possible subtrees including the root:

1. *Movies* alone.
2. *Movies* and *Cartoons* only.
3. *Movies* and *Murder-Mysteries* only.
4. All three entity sets.

We must construct relations for all four "classes." Since only Murder-Mysteries contributes an attribute that is unique to its entities, there is actually some repetition, and these four relations are:

```
Movies(title, year, length, genre)
MoviesC(title, year, length, genre)
MoviesMM(title, year, length, genre, weapon)
MoviesCMM(title, year, length, genre, weapon)
```

If *Cartoons* had attributes unique to that entity set, then all four relations would have different sets of attributes. As that is not the case here, we could combine `Movies` with `MoviesC` (i.e., create one relation for non-murder-mysteries) and combine `MoviesMM` with `MoviesCMM` (i.e., create one relation for all murder mysteries), although doing so loses some information — which movies are cartoons.

We also need to consider how to handle the relationship *Voices* from *Cartoons* to *Stars*. If *Voices* were many-one from *Cartoons*, then we could add a `voice` attribute to `MoviesC` and `MoviesCMM`, which would represent the *Voices* relationship and would have the side-effect of making all four relations different. However, *Voices* is many-many, so we need to create a separate relation for this relationship. As always, its schema has the key attributes from the entity sets connected; in this case

```
Voices(title, year, starName)
```

would be an appropriate schema.

One might consider whether it was necessary to create two such relations, one connecting cartoons that are not murder mysteries to their voices, and the other for cartoons that *are* murder mysteries. However, there does not appear to be any benefit to doing so in this case. □

4.6.3 Using Null Values to Combine Relations

There is one more approach to representing information about a hierarchy of entity sets. If we are allowed to use `NULL` (the null value as in SQL) as a value in tuples, we can handle a hierarchy of entity sets with a single relation. This relation has all the attributes belonging to any entity set of the hierarchy. An entity is then represented by a single tuple. This tuple has `NULL` in each attribute that is not defined for that entity.

Example 4.33: If we applied this approach to the diagram of Fig. 4.31, we would create a single relation whose schema is:

```
Movie(title, year, length, genre, weapon)
```

Those movies that are not murder mysteries would have `NULL` in the `weapon` component of their tuple. It would also be necessary to have a relation `Voices` to connect those movies that are cartoons to the stars performing the voices, as in Example 4.32. □

4.6.4 Comparison of Approaches

Each of the three approaches, which we shall refer to as "straight-E/R," "object-oriented," and "nulls," respectively, have advantages and disadvantages. Here is a list of the principal issues.

1. It can be expensive to answer queries involving several relations, so we would prefer to find all the attributes we needed to answer a query in one relation. The nulls approach uses only one relation for all the attributes, so it has an advantage in this regard. The other two approaches have advantages for different kinds of queries. For instance:

   (a) A query like "what films of 2008 were longer than 150 minutes?" can be answered directly from the relation `Movies` in the straight-E/R approach of Example 4.31. However, in the object-oriented approach of Example 4.32, we need to examine `Movies`, `MoviesC`, `MoviesMM`, and `MoviesCMM`, since a long movie may be in any of these four relations.

   (b) On the other hand, a query like "what weapons were used in cartoons of over 150 minutes in length?" gives us trouble in the straight-E/R approach. We must access `Movies` to find those movies of over 150 minutes. We must access `Cartoons` to verify that a movie is a cartoon, and we must access `MurderMysteries` to find the murder weapon. In the object-oriented approach, we have only to access the relation `MoviesCMM`, where all the information we need will be found.

2. We would like not to use too many relations. Here again, the nulls method shines, since it requires only one relation. However, there is a difference between the other two methods, since in the straight-E/R approach, we use only one relation per entity set in the hierarchy. In the object-oriented approach, if we have a root and n children ($n + 1$ entity sets in all), then there are 2^n different classes of entities, and we need that many relations.

3. We would like to minimize space and avoid repeating information. Since the object-oriented method uses only one tuple per entity, and that tuple has components for only those attributes that make sense for the entity, this approach offers the minimum possible space usage. The nulls approach also has only one tuple per entity, but these tuples are "long"; i.e., they have components for all attributes, whether or not they are appropriate for a given entity. If there are many entity sets in the hierarchy, and there are many attributes among those entity sets, then a large fraction of the space could be wasted in the nulls approach. The straight-E/R method has several tuples for each entity, but only the key attributes are repeated. Thus, this method could use either more or less space than the nulls method.

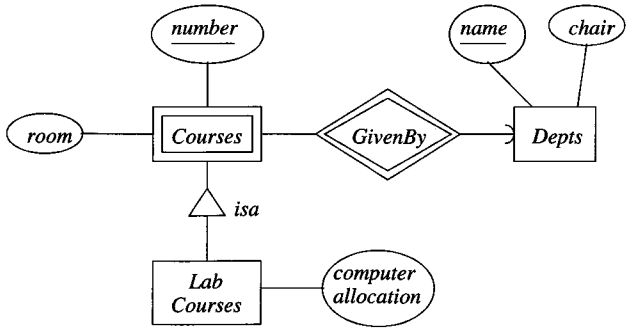

Figure 4.32: E/R diagram for Exercise 4.6.1

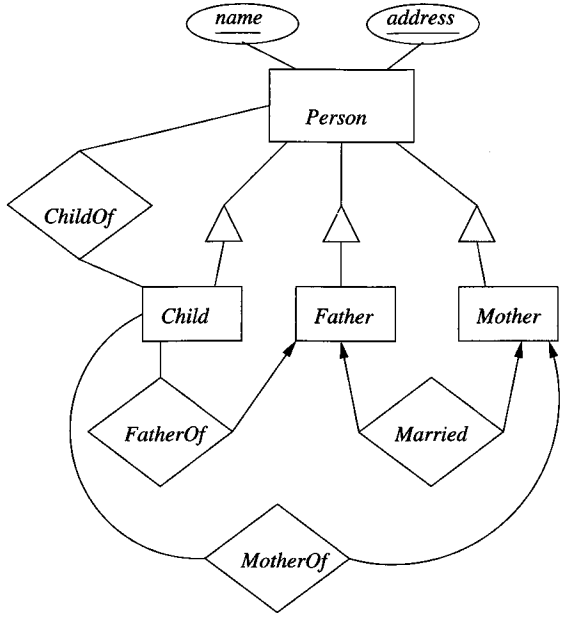

Figure 4.33: E/R diagram for Exercise 4.6.2

4.6.5 Exercises for Section 4.6

Exercise 4.6.1: Convert the E/R diagram of Fig. 4.32 to a relational database schema, using each of the following approaches:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! **Exercise 4.6.2:** Convert the E/R diagram of Fig. 4.33 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

Exercise 4.6.3: Convert your E/R design from Exercise 4.1.7 to a relational database schema, using:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

! **Exercise 4.6.4:** Suppose that we have an isa-hierarchy involving e entity sets. Each entity set has a attributes, and k of those at the root form the key for all these entity sets. Give formulas for (i) the minimum and maximum number of relations used, and (ii) the minimum and maximum number of components that the tuple(s) for a single entity have all together, when the method of conversion to relations is:

a) The straight-E/R method.

b) The object-oriented method.

c) The nulls method.

4.7 Unified Modeling Language

UML (*Unified Modeling Language*) was developed originally as a graphical notation for describing software designs in an object-oriented style. It has been extended, with some modifications, to be a popular notation for describing database designs, and it is this portion of UML that we shall study here. UML offers much the same capabilities as the E/R model, with the exception of multiway relationships. UML also offers the ability to treat entity sets as true classes, with methods as well as data. Figure 4.34 summarizes the common concepts, with different terminology, used by E/R and UML.

UML	E/R Model
Class	Entity set
Association	Binary relationship
Association Class	Attributes on a relationship
Subclass	Isa hierarchy
Aggregation	Many-one relationship
Composition	Many-one relationship with referential integrity

Figure 4.34: Comparison between UML and E/R terminology

4.7.1 UML Classes

A class in UML is similar to an entity set in the E/R model. The notation for a class is rather different, however. Figure 4.35 shows the class that corresponds to the E/R entity set *Movies* from our running example of this chapter.

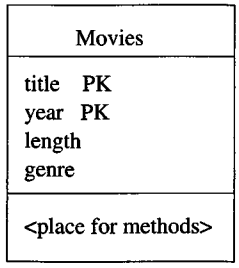

Figure 4.35: The *Movies* class in UML

The box for a class is divided into three parts. At the top is the name of the class. The middle has the attributes, which are like instance variables of a class. In our *Movies* class, we use the attributes *title, year, length*, and *genre*.

The bottom portion is for methods. Neither the E/R model nor the relational model provides methods. However, they are an important concept, and one that actually appears in modern relational systems, called "object-relational" DBMS's (see Section 10.3).

Example 4.34: We might have added an instance method *lengthInHours*(). The UML specification doesn't tell anything more about a method than the types of any arguments and the type of its return-value. Perhaps this method returns *length*/60.0, but we cannot know from the design. □

In this section, we shall not use methods in our design. Thus, in the future, UML class boxes will have only two sections, for the class name and the attributes.

4.7.2 Keys for UML classes

As for entity sets, we can declare one key for a UML class. To do so, we follow each attribute in the key by the letters PK, standing for "primary key." There is no convenient way to stipulate that several attributes or sets of attributes are each keys.

Example 4.35: In Fig. 4.35, we have made our standard assumption that *title* and *year* together form the key for *Movies*. Notice that PK appears on the lines for these attributes and not for the others. □

4.7.3 Associations

A binary relationship between classes is called an *association*. There is no analog of multiway relationships in UML. Rather, a multiway relationship has to be broken into binary relationships, which as we suggested in Section 4.1.10, can always be done. The interpretation of an association is exactly what we described for relationships in Section 4.1.5 on relationship sets. The association is a set of pairs of objects, one from each of the classes it connects.

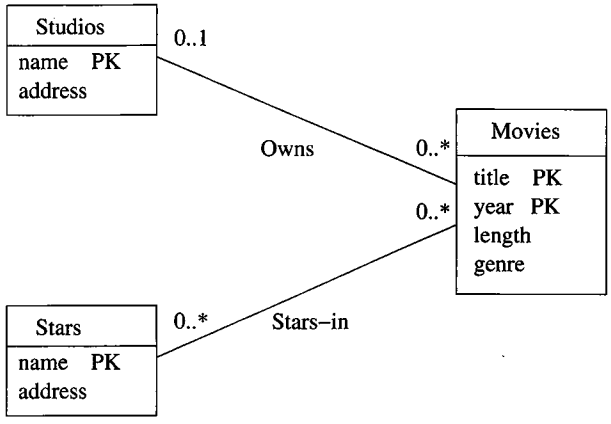

Figure 4.36: Movies, stars, and studios in UML

We draw a UML association between two classes simply by drawing a line between them, and giving the line a name. Usually, we'll place the name below the line. For example, Fig. 4.36 is the UML analog of the E/R diagram of Fig. 4.17. There are two associations, *Stars-in* and *Owns*; the first connects *Movies* with *Stars* and the second connects *Movies* with *Studios*.

Every association has constraints on the number of objects from each of its classes that can be connected to an object of the other class. We indicate these constraints by a label of the form $m..n$ at each end. The meaning of this label is that each object at the other end is connected to at least m and at most n objects at this end. In addition:

- A $*$ in place of n, as in $m..*$, stands for "infinity." That is, there is no upper limit.
- A $*$ alone, in place of $m..n$, stands for the range $0..*$, that is, no constraint at all on the number of objects.
- If there is no label at all at an end of an association edge, then the label is taken to be 1..1, i.e., "exactly one."

Example 4.36: In Fig. 4.36 we see $0..*$ at the *Movies* end of both associations. That says that a star appears in zero or more movies, and a studio owns zero or more movies; i.e., there is no constraint for either. There is also a $0..*$ at the *Stars* end of association *Stars-in*, telling us that a movie has any number of stars. However, the label on the *Studios* end of association *Owns* is 0..1, which means either 0 or 1 studio. That is, a given movie can either be owned by one studio, or not be owned by any studio in the database. Notice that this constraint is exactly what is said by the pointed arrow entering *Studios* in the E/R diagram of Fig. 4.17. □

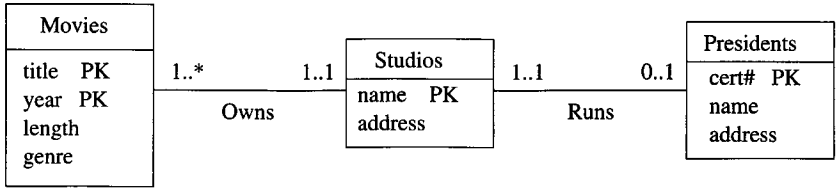

Figure 4.37: Expressing referential integrity in UML

Example 4.37: The UML diagram of Fig. 4.37 is intended to mirror the E/R diagram of Fig. 4.18. Here, we see assumptions somewhat different from those in Example 4.36, about the numbers of movies and studios that can be associated. The label $1..*$ at the *Movies* end of *Owns* says that each studio must own at least one movie (or else it isn't really a studio). There is still no upper limit on how many movies a studio can own.

At the *Studios* end of *Owns*, we see the label 1..1. That label says that a movie must be owned by one studio and only one studio. It is not possible for a movie not to be owned by any studio, as was possible in Fig. 4.36. The label 1..1 says exactly what the rounded arrow in E/R diagrams says.

We also see the association *Runs* between studios and presidents. At the *Studios* end we see label 1..1. That is, a president must be the president of one and only one studio. That label reflects the same constraint as the rounded arrow from *Presidents* to *Studios* in Fig. 4.18. At the other end of association *Runs* is the label 0..1. That label says that a studio can have at most one president, but it could not have a president at some time. This constraint is exactly the constraint of a pointed arrow. □

4.7.4 Self-Associations

An association can have both ends at the same class; such an association is called a *self-association*. To distinguish the two roles played by one class in a self-association, we give the association two names, one for each end.

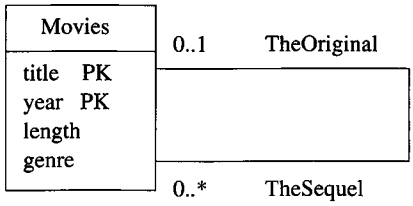

Figure 4.38: A self-association representing sequels of movies

Example 4.38: Figure 4.38 represents the relationship "sequel-of" on movies. We see one association with each end at the class *Movies*. The end with role *TheOriginal* points to the original movie, and it has label 0..1. That is, for a movie to be a sequel, there has to be exactly one movie that was the original. However, some movies are not sequels of any movie. The other role, *TheSequel* has label 0..*. The reasoning is that an original can have any number of sequels. Note we take the point of view that there is an original movie for any sequence of sequels, and a sequel is a sequel of the original, not of the previous movie in the sequence. For instance, *Rocky II* through *Rocky V* are sequels of *Rocky*. We do not assume *Rocky IV* is a sequel of *Rocky III*, and so on. □

4.7.5 Association Classes

We can attach attributes to an association in much the way we did in the E/R model, in Section 4.1.9.[5] In UML, we create a new class, called an *association class*, and attach it to the middle of the association. The association class has its own name, but its attributes may be thought of as attributes of the association to which it attaches.

Example 4.39: Suppose we want to add to the association *Stars-in* between *Movies* and *Stars* some information about the compensation the star received for the movie. This information is not associated with the movie (different stars get different salaries) nor with the star (stars can get different salaries for different movies). Thus, we must attach this information with the association itself. That is, every movie-star pair has its own salary information.

Figure 4.39 shows the association *Stars-in* with an association class called *Compensation*. This class has two attributes, *salary* and *residuals*. Notice

[5]However, the example there in Fig. 4.7 will not carry over directly, because the relationship there is 3-way.

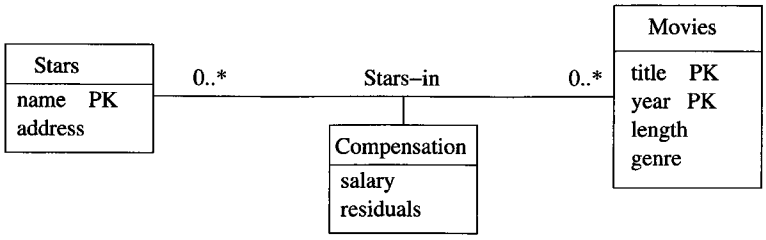

Figure 4.39: *Compensation* is an association class for the association *Stars-in*

that there is no primary key marked for *Compensation*. When we convert a diagram such as Fig. 4.39 to relations, the attributes of *Compensation* will attach to tuples created for movie-star pairs, as was described for relationships in Section 4.5.2. □

4.7.6 Subclasses in UML

Any UML class can have a hierarchy of subclasses below it. The primary key comes from the root of the hierarchy, just as with E/R hierarchies. UML permits a class C to have four different kinds of subclasses below it, depending on our choices of answer to two questions:

1. *Complete versus Partial.* Is every object in the class C a member of some subclass? If so, the subclasses are *complete*; otherwise they are *partial* or *incomplete.*

2. *Disjoint versus Overlapping.* Are the subclasses *disjoint* (an object cannot be in two of the subclasses)? If an object can be in two or more of the subclasses, then the subclasses are said to be *overlapping.*

Note that these decisions are taken at each level of a hierarchy, and the decisions may be made independently at each point.

There are several interesting relationships between the classification of UML subclasses given above, the standard notion of subclasses in object-oriented systems, and the E/R notion of subclasses.

- In a typical object-oriented system, subclasses are disjoint. That is, no object can be in two classes. Of course they inherit properties from their parent class, so in a sense, an object also "belongs" in the parent class. However, the object may not also be in a sibling class.

- The E/R model automatically allows overlapping subclasses.

- Both the E/R model and object-oriented systems allow either complete or partial subclasses. That is, there is no requirement that a member of the superclass be in any subclass.

Subclasses are represented by rectangles, like any class. We assume a subclass inherits the properties (attributes and associations) from its superclass. However, any additional attributes belonging to the subclass are shown in the box for that subclass, and the subclass may have its own, additional, associations to other classes. To represent the class/subclass relationship in UML diagrams, we use a triangular, open arrow pointing to the superclass. The subclasses are usually connected by a horizontal line, feeding into the arrow.

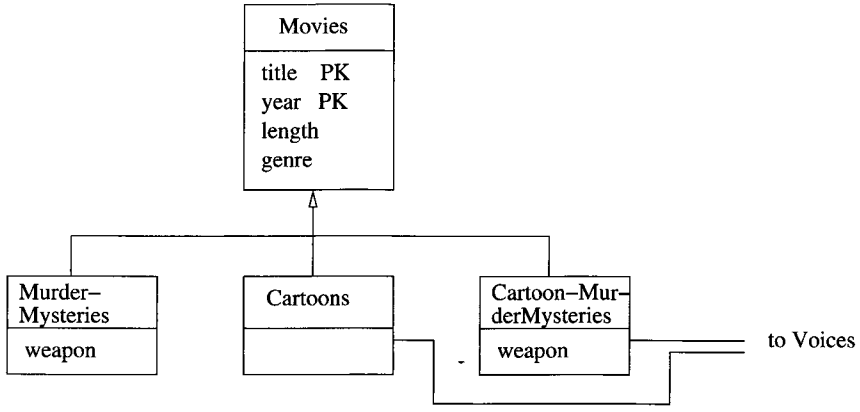

Figure 4.40: Cartoons and murder mysteries as disjoint subclasses of movies

Example 4.40: Figure 4.40 shows a UML variant of the subclass example from Section 4.1.11. However, unlike the E/R subclasses, which are of necessity overlapping, we have chosen here to make the subclasses disjoint. They are partial, of course, since many movies are neither cartoons nor murder mysteries.

Because the subclasses were chosen disjoint, there must be a third subclass for movies like *Roger Rabbit* that are both cartoons and murder mysteries. Notice that both the classes *MurderMysteries* and *Cartoon-MurderMysteries* have additional attribute *weapon*, while the two subclasses *MurderMysteries* and *Cartoon-MurderMysteries* have associations with the unseen class *Voices*. □

4.7.7 Aggregations and Compositions

There are two special notations for many-one associations whose implications are rather subtle. In one sense, they reflect the object-oriented style of programming, where it is common for one class to have references to other classes among its attributes. In another sense, these special notations are really stipulations about how the diagram should be converted to relations; we discuss this aspect of the matter in Section 4.8.3.

An *aggregation* is a line between two classes that ends in an open diamond at one end. The implication of the diamond is that the label at that end must

be 0..1, i.e., the aggregation is a many-one association from the class at the opposite end to the class at the diamond end. Although the aggregation is an association, we do not need to name it, since in practice that name will never be used in a relational implementation.

A *composition* is similar to an association, but the label at the diamond end must be 1..1. That is, every object at the opposite end from the diamond must be connected to exactly one object at the diamond end. Compositions are distinguished by making the diamond be solid black.

Example 4.41: In Fig. 4.41 we see examples of both an aggregation and a composition. It both modifies and elaborates on the situation of Fig. 4.37. We see an association from *Movies* to *Studios*. The label 1..* at the *Movies* end says that a studio has to own at least one movie. We do not need a label at the diamond end, since the open diamond implies a 0..1 label. That is, a movie may or may not be associated with a studio, but cannot be associated with more than one studio. There is also the implication that *Movies* objects will contain a reference to their owning *Studios* object; that reference may be null if the movie is not owned by a studio.

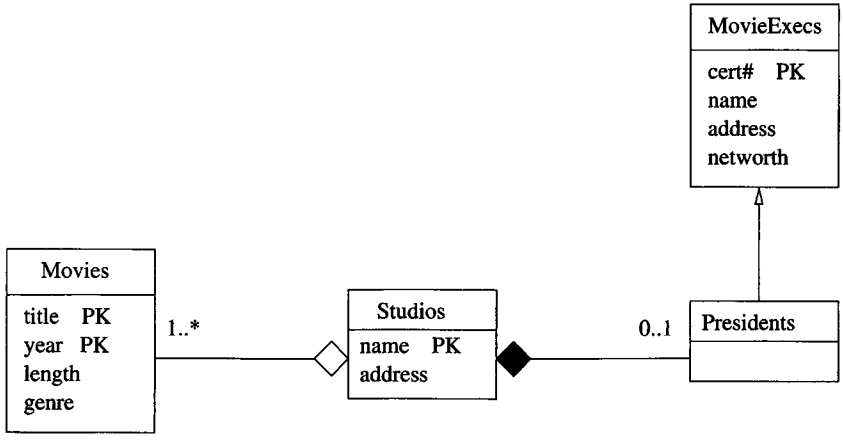

Figure 4.41: An aggregation from *Movies* to *Studios* and a composition from *Presidents* to *Studios*

At the right, we see the class *MovieExecs* with a subclass *Presidents*. There is a composition from *Presidents* to *Studios*. meaning that every president is the president of exactly one studio. A label 1..1 at the *Studios* end is implied by the solid diamond. The implication of the composition is that *Presidents* objects will contain a reference to a *Studios* object, and that this reference cannot be null. □

4.7.8 Exercises for Section 4.7

Exercise 4.7.1: Draw a UML diagram for the problem of Exercise 4.1.1.

Exercise 4.7.2: Modify your diagram from Exercise 4.7.1 in accordance with the requirements of Exercise 4.1.2.

Exercise 4.7.3: Repeat Exercise 4.1.3 using UML.

Exercise 4.7.4: Repeat Exercise 4.1.6 using UML.

Exercise 4.7.5: Repeat Exercise 4.1.7 using UML. Are your subclasses disjoint or overlapping? Are they complete or partial?

Exercise 4.7.6: Repeat Exercise 4.1.9 using UML.

Exercise 4.7.7: Convert the E/R diagram of Fig. 4.30 to a UML diagram.

! **Exercise 4.7.8:** How would you represent the 3-way relationship of *Contracts* among movies, stars, and studios (see Fig. 4.4) in UML?

! **Exercise 4.7.9:** Repeat Exercise 4.2.5 using UML.

Exercise 4.7.10: Usually, when we constrain associations with a label of the form $m..n$, we find that m and n are each either 0, 1, or $*$. Give some examples of associations where it would make sense for at least one of m and n to be something different.

4.8 From UML Diagrams to Relations

Many of the ideas needed to turn E/R diagrams into relations work for UML diagrams as well. We shall therefore briefly review the important techniques, dwelling only on points where the two modeling methods diverge.

4.8.1 UML-to-Relations Basics

Here is an outline of the points that should be familiar from our discussion in Section 4.5:

- *Classes to Relations.* For each class, create a relation whose name is the name of the class, and whose attributes are the attributes of the class.
- *Associations to Relations.* For each association, create a relation with the name of that association. The attributes of the relation are the key attributes of the two connected classes. If there is a coincidence of attributes between the two classes, rename them appropriately. If there is an association class attached to the association, include the attributes of the association class among the attributes of the relation.

Example 4.42: Consider the UML diagram of Fig. 4.36. For the three classes we create relations:

```
Movies(title, year, length genre)
Stars(name, address)
Studios(name, address)
```

For the two associations, we create relations

```
Stars-In(movieTitle, movieYear, starName)
Owns(movieTitle, movieYear, studioName)
```

Note that we have taken some liberties with the names of attributes, for clarity of intention, even though we were not required to do so.

For another example, consider the UML diagram of Fig. 4.39, which shows an association class. The relations for the classes *Movies* and *Stars* would be the same as above. However, for the association, we would have a relation

```
Stars-In(movieTitle, movieYear, starName, salary, residuals)
```

That is, we add to the key attributes of the associated classes, the two attributes of the association class *Compensation*. Note that there is no relation created for *Compensation* itself. □

4.8.2 From UML Subclasses to Relations

The three options we enumerated in Section 4.6 apply to UML subclass hierarchies as well. Recall these options are "E/R style" (relations for each subclass have only the key attributes and attributes of that subclass), "object-oriented" (each entity is represented in the relation for only one subclass), and "use nulls" (one relation for all subclasses). However, if we have information about whether subclasses are disjoint or overlapping, and complete or partial, then we may find one or another method more appropriate. Here are some considerations:

1. If a hierarchy is disjoint at every level, then an object-oriented representation is suggested. We do not have to consider each possible tree of subclasses when forming relations, since we know that each object can belong to only one class and its ancestors in the hierarchy. Thus, there is no possibility of an exponentially exploding number of relations being created.

2. If the hierarchy is both complete and disjoint at every level, then the task is even simpler. If we use the object-oriented approach, then we have only to construct relations for the classes at the leaves of the hierarchy.

3. If the hierarchy is large and overlapping at some or all levels, then the E/R approach is indicated. We are likely to need so many relations that the relational database schema becomes unwieldy.

4.8.3 From Aggregations and Compositions to Relations

Aggregations and compositions are really types of many-one associations. Thus, one approach to their representation in a relational database schema is to convert them as we do for any association in Section 4.8.1. Since these elements are not necessarily named in the UML diagram, we need to invent a name for the corresponding relation.

However, there is a hidden assumption that this implementation of aggregations and compositions is undesirable. Recall from Section 4.5.3 that when we have an entity set E and a many-one relationship R from E to another entity set F, we have the option — some would say the obligation — to combine the relation for E with the relation for R. That is, the one relation constructed from E and R has all the attributes of E plus the key attributes of F.

We suggest that aggregations and compositions be treated routinely in this manner. Construct no relation for the aggregation or composition. Rather, add to the relation for the class at the nondiamond end the key attribute(s) of the class at the diamond end. In the case of an aggregation (but not a composition), it is possible that these attributes can be null.

Example 4.43: Consider the UML diagram of Fig. 4.41. Since there is a small hierarchy, we need to decide how *MovieExecs* and *Presidents* will be translated. Let us adopt the E/R approach, so the `Presidents` relation has only the `cert#` attribute from *MovieExecs*.

The aggregation from *Movies* to *Studios* is represented by putting the key *name* for *Studios* among the attributes for the relation `Movies`. The composition from *Presidents* to *Studios* is represented by adding the key for *Studios* to the relation `Presidents` as well. No relations are constructed for the aggregation or the composition. The following are all the relations we construct from this UML diagram.

```
MovieExecs(cert#, name, address, netWorth)
Presidents(cert#, studioName)
Movies(title, year, length, genre, studioName)
Studios(name, address)
```

As before, we take some liberties with names of attributes to make our intentions clear. □

4.8.4 The UML Analog of Weak Entity Sets

We have not mentioned a UML notation that corresponds to the double-border notation for weak entity sets in the E/R model. There is a sense in which none is needed. The reason is that UML, unlike E/R, draws on the tradition of object-oriented systems, which takes the point of view that each object has its own *object-identity*. That is, we can distinguish two objects, even if they have the same values for each of their attributes and other properties. That object-identity is typically viewed as a reference or pointer to the object.

In UML, we can take the point of view that the objects belonging to a class likewise have object-identity. Thus, even if the stated attributes for a class do not serve to identify a unique object of the class, we can create a new attribute that serves as a key for the corresponding relation and represents the object-identity of the object.

However, it is also possible, in UML, to use a composition as we used supporting relationships for weak entity sets in the E/R model. This composition goes from the "weak" class (the class whose attributes do not provide its key) to the "supporting" class. If there are several "supporting" classes, then several compositions can be used. We shall use a special notation for a *supporting* composition: a small box attached to the *weak* class with "PK" in it will serve as the anchor for the supporting composition. The implication is that the key attribute(s) for the *supporting* class at the other end of the composition is part of the key of the weak class, along with any of the attributes of the weak class that are marked "PK." As with weak entity sets, there can be several supporting compositions and classes, and those supporting classes could themselves be weak, in which case the rule just described is applied recursively.

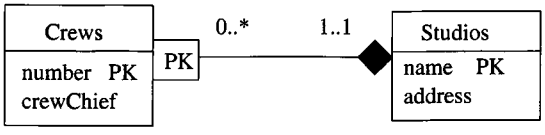

Figure 4.42: Weak class *Crews* supported by a composition and the class *Studios*

Example 4.44: Figure 4.42 shows the analog of the weak entity set *Crews* of Example 4.20. There is a composition from *Crews* to *Studios* anchored by a box labeled "PK" to indicate that this composition provides part of the key for *Crews*. □

We convert weak structures such as Fig. 4.42 to relations exactly as we did in Section 4.5.4. There is a relation for class *Studios* as usual. There is no relation for the composition, again as usual. The relation for class *Crews* includes not only its own attribute *number*, but the key for the class at the end of the composition, which is *Studios*.

Example 4.45: The relations for Example 4.44 are thus:

```
Studios(name, address)
Crews(number, crewChief, studioName)
```

As before, we renamed the attribute *name* of *Studios* in the `Crews` relation, for clarity. □

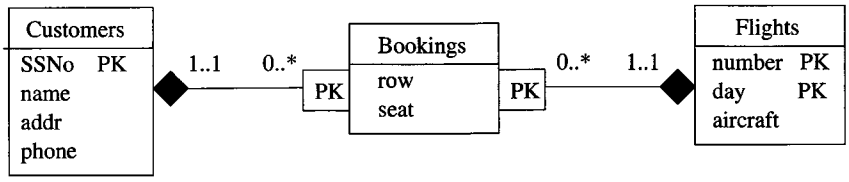

Figure 4.43: A UML diagram analogous to the E/R diagram of Fig. 4.29

4.8.5 Exercises for Section 4.8

Exercise 4.8.1: Convert the UML diagram of Fig. 4.43 to relations.

Exercise 4.8.2: Convert the following UML diagrams to relations:

a) Figure 4.37.

b) Figure 4.40.

c) Your solution to Exercise 4.7.1.

d) Your solution to Exercise 4.7.3.

e) Your solution to Exercise 4.7.4.

f) Your solution to Exercise 4.7.6.

! **Exercise 4.8.3:** How many relations do we create, using the object-oriented approach, if we have a three-level hierarchy with three subclasses of each class at the first and second levels, and that hierarchy is:

a) Disjoint and complete at each level.

b) Disjoint but not complete at each level.

c) Neither disjoint nor complete.

4.9 Object Definition Language

ODL (*Object Definition Language*) is a text-based language for specifying the structure of databases in object-oriented terms. Like UML, the class is the central concept in ODL. Classes in ODL have a name, attributes, and methods, just as UML classes do. Relationships, which are analogous to UML's associations, are not an independent concept in ODL, but are embedded within classes as an additional family of properties.

4.9.1 Class Declarations

A declaration of a class in ODL, in its simplest form, is:

```
class <name> {
   <list of properties>
};
```

That is, the keyword `class` is followed by the name of the class and a bracketed list of properties. A property can be an attribute, a relationship, or a method.

4.9.2 Attributes in ODL

The simplest kind of property is the *attribute*. In ODL, attributes need not be of simple types, such as integers and strings. ODL has a type system, described in Section 4.9.6, that allows us to form structured types and collection types (e.g., sets). For example, an attribute `address` might have a structured type with fields for the street, city, and zip code. An attribute `phones` might have a set of strings as its type, and even more complex types are possible. An attribute is represented in the declaration for its class by the keyword `attribute`, the type of the attribute, and the name of the attribute.

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Genres
              {drama, comedy, sciFi, teen} genre;
    };
```

Figure 4.44: An ODL declaration of the class `Movie`

Example 4.46: In Fig. 4.44 is an ODL declaration of the class of movies. It is not a complete declaration; we shall add more to it later. Line (1) declares `Movie` to be a class. Following line (1) are the declarations of four attributes that all `Movie` objects will have.

Lines (2), (3), and (4) declare three attributes, `title`, `year`, and `length`. The first of these is of character-string type, and the other two are integers. Line (5) declares attribute `genre` to be of enumerated type. The name of the enumeration (list of symbolic constants) is `Genres`, and the four values the attribute `genre` is allowed to take are `drama`, `comedy`, `sciFi`, and `teen`. An enumeration must have a name, which can be used to refer to the same type anywhere. □

Why Name Enumerations and Structures?

The enumeration-name `Genres` in Fig. 4.44 appears to play no role. However, by giving this set of symbolic constants a name, we can refer to it elsewhere, including in the declaration of other classes. In some other class, the *scoped name* `Movie::Genres` can be used to refer to the definition of the enumerated type of this name within the class `Movie`.

Example 4.47: In Example 4.46, all the attributes have primitive types. Here is an example with a complex type. We can define the class `Star` by

```
1)  class Star {
2)      attribute string name;
3)      attribute Struct Addr
            {string street, string city} address;
    };
```

Line (2) specifies an attribute `name` (of the star) that is a string. Line (3) specifies another attribute `address`. This attribute has a type that is a *record structure*. The name of this structure is `Addr`, and the type consists of two fields: `street` and `city`. Both fields are strings. In general, one can define record structure types in ODL by the keyword `Struct` and curly braces around the list of field names and their types. Like enumerations, structure types must have a name, which can be used elsewhere to refer to the same structure type. □

4.9.3 Relationships in ODL

An ODL relationship is declared inside a class declaration, by the keyword `relationship`, a type, and the name of the relationship. The type of a relationship describes what a single object of the class is connected to by the relationship. Typically, this type is either another class (if the relationship is many-one) or a collection type (if the relationship is one-many or many-many). We shall show complex types by example, until the full type system is described in Section 4.9.6.

Example 4.48: Suppose we want to add to the declaration of the `Movie` class from Example 4.46 a property that is a set of stars. More precisely, we want each `Movie` object to connect the set of `Star` objects that are its stars. The best way to represent this connection between the `Movie` and `Star` classes is with a *relationship*. We may represent this relationship by a line:

```
relationship Set<Star> stars;
```

in the declaration of class `Movie`. It says that in each object of class `Movie` there is a set of references to `Star` objects. The set of references is called `stars`. □

4.9.4 Inverse Relationships

Just as we might like to access the stars of a given movie, we might like to know the movies in which a given star acted. To get this information into `Star` objects, we can add the line

```
relationship Set<Movie> starredIn;
```

to the declaration of class `Star` in Example 4.47. However, this line and a similar declaration for `Movie` omits a very important aspect of the relationship between movies and stars. We expect that if a star S is in the `stars` set for movie M, then movie M is in the `starredIn` set for star S. We indicate this connection between the relationships `stars` and `starredIn` by placing in each of their declarations the keyword `inverse` and the name of the other relationship. If the other relationship is in some other class, as it usually is, then we refer to that relationship by its scoped name — the name of its class, followed by a double colon (`::`) and the name of the relationship.

Example 4.49: To define the relationship `starredIn` of class `Star` to be the inverse of the relationship `stars` in class `Movie`, we revise the declarations of these classes, as shown in Fig. 4.45 (which also contains a definition of class `Studio` to be discussed later). Line (6) shows the declaration of relationship `stars` of movies, and says that its inverse is `Star::starredIn`. Since relationship `starredIn` is defined in another class, its scoped name must be used.

Similarly, relationship `starredIn` is declared in line (11). Its inverse is declared by that line to be `stars` of class `Movie`, as it must be, because inverses always are linked in pairs. □

As a general rule, if a relationship R for class C associates with object x of class C with objects $y_1, y_2, \ldots, y_n$ of class D, then the inverse relationship of R associates with each of the y_i's the object x (perhaps along with other objects).

4.9.5 Multiplicity of Relationships

Like the binary relationships of the E/R model, a pair of inverse relationships in ODL can be classified as either many-many, many-one in either direction, or one-one. The type declarations for the pair of relationships tells us which.

1. If we have a many-many relationship between classes C and D, then in class C the type of the relationship is `Set<`D`>`, and in class D the type is `Set<`C`>`.[6]

[6] Actually, the `Set` could be replaced by another "collection type," such as list or bag, as discussed in Section 4.9.6. We shall assume all collections are sets in our exposition of relationships, however.

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Genres
              {drama, comedy, sciFi, teen} genre;
6)     relationship Set<Star> stars
                    inverse Star::starredIn;
7)     relationship Studio ownedBy
                    inverse Studio::owns;
    };

8)  class Star {
9)     attribute string name;
10)    attribute Struct Addr
           {string street, string city} address;
11)    relationship Set<Movie> starredIn
                    inverse Movie::stars;
    };

12) class Studio {
13)    attribute string name;
14)    attribute Star::Addr address;
15)    relationship Set<Movie> owns
                    inverse Movie::ownedBy;
    };
```

Figure 4.45: Some ODL classes and their relationships

2. If the relationship is many-one from C to D, then the type of the relationship in C is just D, while the type of the relationship in D is `Set<`C`>`.

3. If the relationship is many-one from D to C, then the roles of C and D are reversed in (2) above.

4. If the relationship is one-one, then the type of the relationship in C is just D, and in D it is just C.

Note that, as in the E/R model, we allow a many-one or one-one relationship to include the case where for some objects the "one" is actually "none." For instance, a many-one relationship from C to D might have a "null" value of the relationship in some of the C objects. Of course, since a D object could be associated with any set of C objects, it is also permissible for that set to be empty for some D objects.

Example 4.50: In Fig. 4.45 we have the declaration of three classes, `Movie`, `Star`, and `Studio`. The first two of these have already been introduced in Examples 4.46 and 4.47. We also discussed the relationship pair `stars` and `starredIn`. Since each of their types uses `Set`, we see that this pair represents a many-many relationship between `Star` and `Movie`.

`Studio` objects have attributes `name` and `address`; these appear in lines (13) and (14). We have used the same type for addresses of studios as we defined in class `Star` for addresses of stars.

In line (7) we see a relationship `ownedBy` from movies to studios, and the inverse of this relationship is `owns` on line (15). Since the type of `ownedBy` is `Studio`, while the type of `owns` is `Set<Movie>`, we see that this pair of inverse relationships is many-one from `Movie` to `Studio`. □

4.9.6 Types in ODL

ODL offers the database designer a type system similar to that found in C or other conventional programming languages. A type system is built from a basis of types that are defined by themselves and certain recursive rules whereby complex types are built from simpler types. In ODL, the basis consists of:

1. *Primitive types*: integer, float, character, character string, boolean, and *enumerations*. The latter are lists of symbolic names, such as `drama` in line (5) of Fig. 4.45.

2. *Class names*, such as `Movie`, or `Star`, which represent types that are actually structures, with components for each of the attributes and relationships of that class.

These types are combined into structured types using the following *type constructors*:

1. *Set.* If T is any type, then `Set<T>` denotes the type whose values are finite sets of elements of type T. Examples using the set type-constructor occur in lines (6), (11), and (15) of Fig. 4.45.

2. *Bag.* If T is any type, then `Bag<T>` denotes the type whose values are finite bags or *multisets* of elements of type T.

3. *List.* If T is any type, then `List<T>` denotes the type whose values are finite lists of zero or more elements of type T.

4. *Array.* If T is a type and i is an integer, then `Array<T,i>` denotes the type whose elements are arrays of i elements of type T. For example, `Array<char,10>` denotes character strings of length 10.

5. *Dictionary.* If T and S are types, then `Dictionary<T,S>` denotes a type whose values are finite sets of pairs. Each pair consists of a value of the *key type* T and a value of the *range type* S. The dictionary may not contain two pairs with the same key value.

Sets, Bags, and Lists

To understand the distinction between sets, bags, and lists, remember that a set has unordered elements, and only one occurrence of each element. A bag allows more than one occurrence of an element, but the elements and their occurrences are unordered. A list allows more than one occurrence of an element, but the occurrences are ordered. Thus, $\{1,2,1\}$ and $\{2,1,1\}$ are the same bag, but $(1,2,1)$ and $(2,1,1)$ are not the same list.

6. *Structures.* If $T_1, T_2, \ldots, T_n$ are types, and $F_1, F_2, \ldots, F_n$ are names of fields, then

   `Struct N {`T_1 F_1`,` T_2 F_2`, ...,` T_n F_n`}`

   denotes the type named N whose elements are structures with n fields. The ith field is named F_i and has type T_i. For example, line (10) of Fig. 4.45 showed a structure type named `Addr`, with two fields. Both fields are of type `string` and have names `street` and `city`, respectively.

The first five types — set, bag, list, array, and dictionary — are called *collection types*. There are different rules about which types may be associated with attributes and which with relationships.

- The type of a relationship is either a class type or a single use of a collection type constructor applied to a class type.
- The type of an attribute is built starting with a primitive type or types.[7] We may then apply the structure and collection type constructors as we wish, as many times as we wish.

Example 4.51: Some of the possible types of attributes are:

1. `integer`.
2. `Struct N {string field1, integer field2}`.
3. `List<real>`.
4. `Array<Struct N {string field1, integer field2}, 10>`.

[7] Class types may also be used, which makes the attribute behave like a "one-way" relationship. We shall not consider such attributes here.

Example (1) is a primitive type, (2) is a structure of primitive types, (3) a collection of a primitive type, and (4) a collection of structures built from primitive types.

Now, suppose the class names `Movie` and `Star` are available primitive types. Then we may construct relationship types such as `Movie` or `Bag<Star>`. However, the following are illegal as relationship types:

1. `Struct N {Movie field1, Star field2}`. Relationship types cannot involve structures.
2. `Set<integer>`. Relationship types cannot involve primitive types.
3. `Set<Array<Star, 10>>`. Relationship types cannot involve two applications of collection types.

□

4.9.7 Subclasses in ODL

We can declare one class C to be a subclass of another class D. To do so, follow the name C in its declaration with the keyword `extends` and the name D. Then, class C inherits all the properties of D, and may have additional properties of its own.

Example 4.52: Recall Example 4.10, where we declared cartoons to be a subclass of movies, with the additional property of a relationship from a cartoon to a set of stars that are its "voices." We can create a subclass `Cartoon` for Movie with the ODL declaration:

```
class Cartoon extends Movie {
   relationship Set<Star> voices;
};
```

Also in that example, we defined a class of murder mysteries with additional attribute `weapon`.

```
class MurderMystery extends Movie {
   attribute string weapon;
};
```

is a suitable declaration of this subclass. □

Sometimes, as in the case of a movie like "Roger Rabbit," we need a class that is a subclass of two or more other classes at the same time. In ODL, we may follow the keyword `extends` by several classes, separated by colons.[8] Thus, we may declare a fourth class by:

[8]Technically, the second and subsequent names must be "interfaces," rather than classes. Roughly, an *interface* in ODL is a class definition without an associated set of objects.

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

Note that when there is multiple inheritance, there is the potential for a class to inherit two properties with the same name. The way such conflicts are resolved is implementation-dependent.

4.9.8 Declaring Keys in ODL

The declaration of a key or keys for a class is optional. The reason is that ODL, being object-oriented, assumes that all objects have an object-identity, as discussed in connection with UML in Section 4.8.4.

In ODL we may declare one or more attributes to be a key for a class by using the keyword `key` or `keys` (it doesn't matter which) followed by the attribute or attributes forming keys. If there is more than one attribute in a key, the list of attributes must be surrounded by parentheses. The key declaration itself appears inside parentheses, following the name of the class itself in the first line of its declaration.

Example 4.53: To declare that the set of two attributes `title` and `year` form a key for class `Movie`, we could begin its declaration:

```
class Movie (key (title, year)) {
```

We could have used `keys` in place of `key`, even though only one key is declared. □

It is possible that several sets of attributes are keys. If so, then following the word `key(s)` we may place several keys separated by commas. A key that consists of more than one attribute must have parentheses around the list of its attributes, so we can disambiguate a key of several attributes from several keys of one attribute each.

The ODL standard also allows properties other than attributes to appear in keys. There is no fundamental problem with a method or relationship being declared a key or part of a key, since keys are advisory statements that the DBMS can take advantage of or not, as it wishes. For instance, one could declare a method to be a key, meaning that on distinct objects of the class the method is guaranteed to return distinct values.

When we allow many-one relationships to appear in key declarations, we can get an effect similar to that of weak entity sets in the E/R model. We can declare that the object O_1 referred to by an object O_2 on the "many" side of the relationship, perhaps together with other properties of O_2 that are included in the key, is unique for different objects O_2. However, we should remember that there is no requirement that classes have keys; we are never obliged to handle, in some special way, classes that lack attributes of their own to form a key, as we did for weak entity sets.

Example 4.54: Let us review the example of a weak entity set *Crews* in Fig. 4.20. Recall that we hypothesized that crews were identified by their number, and the studio for which they worked, although two studios might have crews with the same number. We might declare the class `Crew` as in Fig. 4.46. Note that we should modify the declaration of `Studio` to include the relationship `crewsOf` that is an inverse to the relationship `unitOf` in `Crew`; we omit this change.

```
class Crew (key (number, unitOf)) {
    attribute integer number;
    attribute string crewChief;
    relationship Studio unitOf
        inverse Studio::crewsOf;
};
```

Figure 4.46: A ODL declaration for crews

What this key declaration asserts is that there cannot be two crews that both have the same value for the `number` attribute and are related to the same studio by `unitOf`. Notice how this assertion resembles the implication of the E/R diagram in Fig. 4.20, which is that the number of a crew and the name of the related studio (i.e., the key for studios) uniquely determine a crew entity. □

4.9.9 Exercises for Section 4.9

Exercise 4.9.1: In Exercise 4.1.1 was the informal description of a bank database. Render this design in ODL, including keys as appropriate.

Exercise 4.9.2: Modify your design of Exercise 4.9.1 in the ways enumerated in Exercise 4.1.2. Describe the changes; do not write a complete, new schema.

Exercise 4.9.3: Render the teams-players-fans database of Exercise 4.1.3 in ODL, including keys, as appropriate. Why does the complication about sets of team colors, which was mentioned in the original exercise, not present a problem in ODL?

! **Exercise 4.9.4:** Suppose we wish to keep a genealogy. We shall have one class, `Person`. The information we wish to record about persons includes their name (an attribute) and the following relationships: mother, father, and children. Give an ODL design for the `Person` class. Be sure to indicate the inverses of the relationships that, like `mother`, `father`, and `children`, are also relationships from `Person` to itself. Is the inverse of the `mother` relationship the `children` relationship? Why or why not? Describe each of the relationships and their inverses as sets of pairs.

! **Exercise 4.9.5:** Let us add to the design of Exercise 4.9.4 the attribute `education`. The value of this attribute is intended to be a collection of the degrees obtained by each person, including the name of the degree (e.g., B.S.), the school, and the date. This collection of structs could be a set, bag, list, or array. Describe the consequences of each of these four choices. What information could be gained or lost by making each choice? Is the information lost likely to be important in practice?

Exercise 4.9.6: In Exercise 4.4.4 we saw two examples of situations where weak entity sets were essential. Render these databases in ODL, including declarations for suitable keys.

Exercise 4.9.7: Give an ODL design for the registrar's database described in Exercise 4.1.9.

!! **Exercise 4.9.8:** Under what circumstances is a relationship its own inverse? *Hint*: Think about the relationship as a set of pairs, as discussed in Section 4.9.4.

4.10 From ODL Designs to Relational Designs

ODL was actually intended as the data-definition part of a language standard for object-oriented DBMS's, analogous to the SQL `CREATE TABLE` statement. Indeed, there have been some attempts to implement such a system. However, it is also possible to see ODL as a text-based, high-level design notation, from which we eventually derive a relational database schema. Thus, in this section we shall consider how to convert ODL designs into relational designs.

Much of the process is similar to that we discussed for E/R diagrams in Section 4.5 and for UML in Section 4.8. Classes become relations, and relationships become relations that connect the key attributes of the classes involved in the relationship. Yet some new problems arise for ODL, including:

1. Entity sets must have keys, but there is no such guarantee for ODL classes.
2. While attributes in E/R, UML, and the relational model are of primitive type, there is no such constraint for ODL attributes.

4.10.1 From ODL Classes to Relations

As a starting point, let us assume that our goal is to have one relation for each class and for that relation to have one attribute for each property. We shall see many ways in which this approach must be modified, but for the moment, let us consider the simplest possible case, where we can indeed convert classes to relations and properties to attributes. The restrictions we assume are:

1. All properties of the class are attributes (not relationships or methods).

2. The types of the attributes are primitive (not structures or sets).

In this case, the ODL class looks almost like an entity set or a UML class. Although there might be no key for the ODL class, ODL assumes object-identity. We can create an artificial attribute to represent the object-identity and serve as a key for the relation; this issue was introduced for UML in Section 4.8.4.

Example 4.55: Figure 4.47 is an ODL description of movie executives. No key is listed, and we do not assume that `name` uniquely determines a movie executive (unlike stars, who will make sure their chosen name is unique).

```
class MovieExec {
   attribute string name;
   attribute string address;
   attribute integer netWorth;
};
```

Figure 4.47: The class `MovieExec`

We create a relation with the same name as the class. The relation has four attributes, one for each attribute of the class, and one for the object-identity:

```
MovieExecs(cert#, name, address, netWorth)
```

We use `cert#` as the key attribute, representing the object-identity. □

4.10.2 Complex Attributes in Classes

Even when a class' properties are all attributes we may have some difficulty converting the class to a relation. The reason is that attributes in ODL can have complex types such as structures, sets, bags, or lists. On the other hand, a fundamental principle of the relational model is that a relation's attributes have a primitive type, such as numbers and strings. Thus, we must find some way to represent complex attribute types as relations.

Record structures whose fields are themselves primitive are the easiest to handle. We simply expand the structure definition, making one attribute of the relation for each field of the structure.

```
class Star (key name) {
   attribute string name;
   attribute Struct Addr
       {string street, string city} address;
};
```

Figure 4.48: Class with a structured attribute

Example 4.56: In Fig. 4.48 is a declaration for class `Star`, with only attributes as properties. The attribute `name` is of primitive type, but attribute `address` is a structure with two fields, `street` and `city`. We represent this class by the relation:

```
Star(name, street, city)
```

The key is `name`, and the attributes `street` and `city` represent the structure `address`. □

4.10.3 Representing Set-Valued Attributes

However, record structures are not the most complex kind of attribute that can appear in ODL class definitions. Values can also be built using type constructors `Set`, `Bag`, `List`, `Array`, and `Dictionary` from Section 4.9.6. Each presents its own problems when migrating to the relational model. We shall only discuss the `Set` constructor, which is the most common, in detail.

One approach to representing a set of values for an attribute A is to make one tuple for each value. That tuple includes the appropriate values for all the other attributes besides A. This approach works, although it is likely to produce unnormalized relations, as we shall see in the next example.

```
class Star (key name) {
   attribute string name;
   attribute Set<
           Struct Addr {string street, string city}
       > address;
   attribute Date birthdate;
};
```

Figure 4.49: Stars with a set of addresses and a birthdate

Example 4.57: Figure 4.49 shows a new definition of the class `Star`, in which we have allowed stars to have a set of addresses and also added a nonkey, primitive attribute `birthdate`. The `birthdate` attribute can be an attribute of the `Star` relation, whose schema now becomes:

```
Star(name, street, city, birthdate)
```

Unfortunately, this relation exhibits the sort of anomalies we saw in Section 3.3.1. If Carrie Fisher has two addresses, say a home and a beach house, then she is represented by two tuples in the relation `Star`. If Harrison Ford has an empty set of addresses, then he does not appear at all in `Star`. A typical set of tuples for `Star` is shown in Fig. 4.50.

name	*street*	*city*	*birthdate*
Carrie Fisher	123 Maple St.	Hollywood	9/9/99
Carrie Fisher	5 Locust Ln.	Malibu	9/9/99
Mark Hamill	456 Oak Rd.	Brentwood	8/8/88

Figure 4.50: Adding birthdates

Although `name` is a key for the class `Star`, our need to have several tuples for one star to represent all their addresses means that `name` is *not* a key for the relation `Star`. In fact, the key for that relation is {`name, street, city`}. Thus, the functional dependency

$$\texttt{name} \rightarrow \texttt{birthdate}$$

is a BCNF violation and the multivalued dependency

$$\texttt{name} \twoheadrightarrow \texttt{street city}$$

is a 4NF violation as well. □

There are several options regarding how to handle set-valued attributes that appear in a class declaration along with other attributes, set-valued or not. One approach is to separate out each set-valued attribute as if it were a many-many relationship between the objects of the class and the values that appear in the sets.

An alternative approach is to place all attributes, set-valued or not, in the schema for the relation, then use the normalization techniques of Sections 3.3 and 3.6 to eliminate the resulting BCNF and 4NF violations. Notice that any set-valued attribute in conjunction with any single-valued attribute leads to a BNCF violation, as in Example 4.57. Two set-valued attributes in the same class declaration will lead to a 4NF violation, even if there are no single-valued attributes.

4.10.4 Representing Other Type Constructors

Besides record structures and sets, an ODL class definition could use `Bag`, `List`, `Array`, or `Dictionary` to construct values. To represent a bag (multiset), in which a single object can be a member of the bag n times, we cannot simply introduce into a relation n identical tuples.[9] Instead, we could add to the relation schema another attribute `count` representing the number of times that

[9]To be precise, we cannot introduce identical tuples into relations of the abstract relational model described in Section 2.2. However, SQL-based relational DBMS's *do* allow duplicate tuples; i.e., relations are bags rather than sets in SQL. See Sections 5.1 and 6.4. If queries are likely to ask for tuple counts, we advise using a scheme such as that described here, even if your DBMS allows duplicate tuples.

each element is a member of the bag. For instance, suppose that address in Fig. 4.49 were a bag instead of a set. We could say that 123 Maple St., Hollywood is Carrie Fisher's address twice and 5 Locust Ln., Malibu is her address 3 times (whatever that may mean) by

name	*street*	*city*	*count*
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

A list of addresses could be represented by a new attribute position, indicating the position in the list. For instance, we could show Carrie Fisher's addresses as a list, with Hollywood first, by:

name	*street*	*city*	*position*
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

A fixed-length array of addresses could be represented by attributes for each position in the array. For instance, if address were to be an array of two street-city structures, we could represent Star objects as:

name	*street1*	*city1*	*street2*	*city2*
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

Finally, a dictionary could be represented as a set, but with attributes for both the key-value and range-value components of the pairs that are members of the dictionary. For instance, suppose that instead of star's addresses, we really wanted to keep, for each star, a dictionary giving the mortgage holder for each of their homes. Then the dictionary would have address as the key value and bank name as the range value. A hypothetical rendering of the Carrie-Fisher object with a dictionary attribute is:

name	*street*	*city*	*mortgage-holder*
Carrie Fisher	123 Maple St.	Hollywood	Bank of Burbank
Carrie Fisher	5 Locust Ln.	Malibu	Torrance Trust

Of course attribute types in ODL may involve more than one type constructor. If a type is any collection type besides dictionary applied to a structure (e.g., a set of structs), then we may apply the techniques from Sections 4.10.3 or 4.10.4 as if the struct were an atomic value, and then replace the single attribute representing the atomic value by several attributes, one for each field of the struct. This strategy was used in the examples above, where the address is a struct. The case of a dictionary applied to structs is similar and left as an exercise.

There are many reasons to limit the complexity of attribute types to an optional struct followed by an optional collection type. We mentioned in Section 4.1.1 that some versions of the E/R model allow exactly this much generality in the types of attributes, although we restricted ourselves to attributes of

primitive type in the E/R model. We recommend that, if you are going to use an ODL design for the purpose of eventual translation to a relational database schema, you similarly limit yourself. We take up in the exercises some options for dealing with more complex types as attributes.

4.10.5 Representing ODL Relationships

Usually, an ODL class definition will contain relationships to other ODL classes. As in the E/R model, we can create for each relationship a new relation that connects the keys of the two related classes. However, in ODL, relationships come in inverse pairs, and we must create only one relation for each pair.

When a relationship is many-one, we have an option to combine it with the relation that is constructed for the class on the "many" side. Doing so has the effect of combining two relations that have a common key, as we discussed in Section 4.5.3. It therefore does not cause a BCNF violation and is a legitimate and commonly followed option.

4.10.6 Exercises for Section 4.10

Exercise 4.10.1: Convert your ODL designs from the following exercises to relational database schemas.

a) Exercise 4.9.1.

b) Exercise 4.9.2 (include all four of the modifications specified by that exercise).

c) Exercise 4.9.3.

d) Exercise 4.9.4.

e) Exercise 4.9.5.

! **Exercise 4.10.2:** Consider an attribute of type `Dictionary` with key and range types both structs of primitive types. Show how to convert a class with an attribute of this type to a relation.

Exercise 4.10.3: We mentioned that when attributes are of a type more complex than a collection of structs, it becomes tricky to convert them to relations; in particular, it becomes necessary to create some intermediate concepts and relations for them. The following sequence of questions will examine increasingly more complex types and how to represent them as relations.

a) A *card* can be represented as a struct with fields `rank` (2, 3, ... , 10, Jack, Queen, King, and Ace) and `suit` (Clubs, Diamonds, Hearts, and Spades). Give a suitable definition of a structured type `Card`. This definition should be independent of any class declarations but available to them all.

b) A *hand* is a set of cards. The number of cards may vary. Give a declaration of a class `Hand` whose objects are hands. That is, this class declaration has an attribute `theHand`, whose type is a hand.

! c) Convert your class declaration `Hand` from (b) to a relation schema.

d) A *poker hand* is a set of five cards. Repeat (b) and (c) for poker hands.

! e) A *deal* is a set of pairs, each pair consisting of the name of a player and a hand for that player. Declare a class `Deal`, whose objects are deals. That is, this class declaration has an attribute `theDeal`, whose type is a deal.

f) Repeat (e), but restrict hands of a deal to be hands of exactly five cards.

g) Repeat (e), using a dictionary for a deal. You may assume the names of players in a deal are unique.

!! h) Convert your class declaration from (e) to a relational database schema.

! i) Suppose we defined deals to be sets of sets of cards, with no player associated with each hand (set of cards). It is proposed that we represent such deals by a relation schema `Deals(dealID, card)`, meaning that the card was a member of one of the hands in the deal with the given ID. What, if anything, is wrong with this representation? How would you fix the problem?

Exercise 4.10.4: Suppose we have a class C defined by

```
class C (key a) {
    attribute string a;
    attribute T b;
};
```

where T is some type. Give the relation schema for the relation derived from C and indicate its key attributes if T is:

a) `Set<Struct S {string f, string g}>`

! b) `Bag<Struct S {string f, string g}>`

! c) `List<Struct S {string f, string }>`

! d) `Dictionary<Struct K {string f, string g}, Struct R {string i, string j}>`

4.11 Summary of Chapter 4

- ✦ *The Entity-Relationship Model*: In the E/R model we describe entity sets, relationships among entity sets, and attributes of entity sets and relationships. Members of entity sets are called entities.
- ✦ *Entity-Relationship Diagrams*: We use rectangles, diamonds, and ovals to draw entity sets, relationships, and attributes, respectively.
- ✦ *Multiplicity of Relationships*: Binary relationships can be one-one, many-one, or many-many. In a one-one relationship, an entity of either set can be associated with at most one entity of the other set. In a many-one relationship, each entity of the "many" side is associated with at most one entity of the other side. Many-many relationships place no restriction.
- ✦ *Good Design*: Designing databases effectively requires that we represent the real world faithfully, that we select appropriate elements (e.g., relationships, attributes), and that we avoid redundancy — saying the same thing twice or saying something in an indirect or overly complex manner.
- ✦ *Subclasses*: The E/R model uses a special relationship *isa* to represent the fact that one entity set is a special case of another. Entity sets may be connected in a hierarchy with each child node a special case of its parent. Entities may have components belonging to any subtree of the hierarchy, as long as the subtree includes the root.
- ✦ *Weak Entity Sets*: These require attributes of some supporting entity set(s) to identify their own entities. A special notation involving diamonds and rectangles with double borders is used to distinguish weak entity sets.
- ✦ *Converting Entity Sets to Relations*: The relation for an entity set has one attribute for each attribute of the entity set. An exception is a weak entity set E, whose relation must also have attributes for the key attributes of its supporting entity sets.
- ✦ *Converting Relationships to Relations*: The relation for an E/R relationship has attributes corresponding to the key attributes of each entity set that participates in the relationship. However, if a relationship is a supporting relationship for some weak entity set, it is not necessary to produce a relation for that relationship.
- ✦ *Converting Isa Hierarchies to Relations*: One approach is to create a relation for each entity set with the key attributes of the hierarchy's root plus the attributes of the entity set itself. A second approach is to create a relation for each possible subset of the entity sets in the hierarchy, and create for each entity one tuple; that tuple is in the relation for exactly the set of entity sets to which the entity belongs. A third approach is to create only one relation and to use null values for those attributes that do not apply to the entity represented by a given tuple.

- *Unified Modeling Language*: In UML, we describe classes and associations between classes. Classes are analogous to E/R entity sets, and associations are like binary E/R relationships. Special kinds of many-one associations, called aggregations and compositions, are used and have implications as to how they are translated to relations.

- *UML Subclass Hierarchies*: UML permits classes to have subclasses, with inheritance from the superclass. The subclasses of a class can be complete or partial, and they can be disjoint or overlapping.

- *Converting UML Diagrams to Relations*: The methods are similar to those used for the E/R model. Classes become relations and associations become relations connecting the keys of the associated classes. Aggregations and compositions are combined with the relation constructed from the class at the "many" end.

- *Object Definition Language*: This language is a notation for formally describing the schemas of databases in an object-oriented style. One defines classes, which may have three kinds of properties: attributes, methods, and relationships.

- *ODL Relationships*: A relationship in ODL must be binary. It is represented, in the two classes it connects, by names that are declared to be inverses of one another. Relationships can be many-many, many-one, or one-one, depending on whether the types of the pair are declared to be a single object or a set of objects.

- *The ODL Type System*: ODL allows types to be constructed, beginning with class names and atomic types such as integer, by applying any of the following type constructors: structure formation, set-of, bag-of, list-of, array-of, and dictionary-of.

- *Keys in ODL*: Keys are optional in ODL. We can declare one or more keys, but because objects have an object-ID that is not one of its properties, a system implementing ODL can tell the difference between objects, even if they have identical values for all properties.

- *Converting ODL Classes to Relations*: The method is the same as for E/R or UML, except if the class has attributes of complex type. If that happens the resulting relation may be unnormalized and will have to be decomposed. It may also be necessary to create a new attribute to represent the object-identity of objects and serve as a key.

- *Converting ODL Relationships to Relations*: The method is the same as for E/R relationships, except that we must first pair ODL relationships and their inverses, and create only one relation for the pair.

4.12 References for Chapter 4

The original paper on the Entity-Relationship model is [5]. Two books on the subject of E/R design are [2] and [7].

The manual defining ODL is [4]. One can also find more about the history of object-oriented database systems from [1], [3], and [6].

1. F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System*, Morgan-Kaufmann, San Francisco, 1992.

2. Carlo Batini, S. Ceri, S. B. Navathe, and Carol Batini, *Conceptual Database Design: an Entity/Relationship Approach*, Addison-Wesley, Boston MA, 1991.

3. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading, MA, 1994.

4. R. G. G. Cattell (ed.), *The Object Database Standard: ODMG–99*, Morgan-Kaufmann, San Francisco, 1999.

5. P. P. Chen, "The entity-relationship model: toward a unified view of data," *ACM Trans. on Database Systems* 1:1, pp. 9–36, 1976.

6. W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, 1994.

7. B. Thalheim, "Fundamentals of Entity-Relationship Modeling," Springer-Verlag, Berlin, 2000.

Part II

Relational Database Programming

Chapter 5

Algebraic and Logical Query Languages

We now switch our attention from modeling to programming for relational databases. We start in this discussion with two abstract programming languages, one algebraic and the other logic-based. The algebraic programming language, relational algebra, was introduced in Section 2.4, to let us see what operations in the relational model look like. However, there is more to the algebra. In this chapter, we extend the set-based algebra of Section 2.4 to bags, which better reflect the way the relational model is implemented in practice. We also extend the algebra so it can handle several more operations than were described previously; for example, we need to do aggregations (e.g., averages) of columns of a relation.

We close the chapter with another form of query language, based on logic. This language, called "Datalog," allows us to express queries by describing the desired results, rather than by giving an algorithm to compute the results, as relational algebra requires.

5.1 Relational Operations on Bags

In this section, we shall consider relations that are bags (multisets) rather than sets. That is, we shall allow the same tuple to appear more than once in a relation. When relations are bags, there are changes that need to be made to the definition of some relational operations, as we shall see. First, let us look at a simple example of a relation that is a bag but not a set.

Example 5.1: The relation in Fig. 5.1 is a bag of tuples. In it, the tuple $(1, 2)$ appears three times and the tuple $(3, 4)$ appears once. If Fig. 5.1 were a set-valued relation, we would have to eliminate two occurrences of the tuple $(1, 2)$. In a bag-valued relation, we *do* allow multiple occurrences of the same tuple, but like sets, the order of tuples does not matter. □

A	B
1	2
3	4
1	2
1	2

Figure 5.1: A bag

5.1.1 Why Bags?

As we mentioned, commercial DBMS's implement relations that are bags, rather than sets. An important motivation for relations as bags is that some relational operations are considerably more efficient if we use the bag model. For example:

1. To take the union of two relations as bags, we simply copy one relation and add to the copy all the tuples of the other relation. There is no need to eliminate duplicate copies of a tuple that happens to be in both relations.

2. When we project relation as sets, we need to compare each projected tuple with all the other projected tuples, to make sure that each projection appears only once. However, if we can accept a bag as the result, then we simply project each tuple and add it to the result; no comparison with other projected tuples is necessary.

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

Figure 5.2: Bag for Example 5.2

Example 5.2: The bag of Fig. 5.1 could be the result of projecting the relation shown in Fig. 5.2 onto attributes A and B, provided we allow the result to be a bag and do not eliminate the duplicate occurrences of $(1, 2)$. Had we used the ordinary projection operator of relational algebra, and therefore eliminated duplicates, the result would be only:

A	B
1	2
3	4

Note that the bag result, although larger, can be computed more quickly, since there is no need to compare each tuple $(1,2)$ or $(3,4)$ with previously generated tuples. □

Another motivation for relations as bags is that there are some situations where the expected answer can only be obtained if we use bags, at least temporarily. Here is an example.

Example 5.3: Suppose we want to take the average of the A-components of a set-valued relation such as Fig. 5.2. We could not use the set model to think of the relation projected onto attribute A. As a set, the average value of A is 2, because there are only two values of A — 1 and 3 — in Fig. 5.2, and their average is 2. However, if we treat the A-column in Fig. 5.2 as a bag $\{1,3,1,1\}$, we get the correct average of A, which is 1.5, among the four tuples of Fig. 5.2. □

5.1.2 Union, Intersection, and Difference of Bags

These three operations have new definitions for bags. Suppose that R and S are bags, and that tuple t appears n times in R and m times in S. Note that either n or m (or both) can be 0. Then:

- In the bag union $R \cup S$, tuple t appears $n + m$ times.
- In the bag intersection $R \cap S$, tuple t appears $\min(n, m)$ times.
- In the bag difference $R - S$, tuple t appears $\max(0, n - m)$ times. That is, if tuple t appears in R more times than it appears in S, then t appears in $R - S$ the number of times it appears in R, minus the number of times it appears in S. However, if t appears at least as many times in S as it appears in R, then t does not appear at all in $R - S$. Intuitively, occurrences of t in S each "cancel" one occurrence in R.

Example 5.4: Let R be the relation of Fig. 5.1, that is, a bag in which tuple $(1,2)$ appears three times and $(3,4)$ appears once. Let S be the bag

A	B
1	2
3	4
3	4
5	6

Then the bag union $R \cup S$ is the bag in which $(1,2)$ appears four times (three times for its occurrences in R and once for its occurrence in S); $(3,4)$ appears three times, and $(5,6)$ appears once.

The bag intersection $R \cap S$ is the bag

A	B
1	2
3	4

with one occurrence each of $(1,2)$ and $(3,4)$. That is, $(1,2)$ appears three times in R and once in S, and $\min(3,1) = 1$, so $(1,2)$ appears once in $R \cap S$. Similarly, $(3,4)$ appears $\min(1,2) = 1$ time in $R \cap S$. Tuple $(5,6)$, which appears once in S but zero times in R appears $\min(0,1) = 0$ times in $R \cap S$. In this case, the result happens to be a set, but any set is also a bag.

The bag difference $R - S$ is the bag

A	B
1	2
1	2

To see why, notice that $(1,2)$ appears three times in R and once in S, so in $R - S$ it appears $\max(0, 3-1) = 2$ times. Tuple $(3,4)$ appears once in R and twice in S, so in $R - S$ it appears $\max(0, 1-2) = 0$ times. No other tuple appears in R, so there can be no other tuples in $R - S$.

As another example, the bag difference $S - R$ is the bag

A	B
3	4
5	6

Tuple $(3,4)$ appears once because that is the number of times it appears in S minus the number of times it appears in R. Tuple $(5,6)$ appears once in $S - R$ for the same reason. □

5.1.3 Projection of Bags

We have already illustrated the projection of bags. As we saw in Example 5.2, each tuple is processed independently during the projection. If R is the bag of Fig. 5.2 and we compute the bag-projection $\pi_{A,B}(R)$, then we get the bag of Fig. 5.1.

If the elimination of one or more attributes during the projection causes the same tuple to be created from several tuples, these duplicate tuples are not eliminated from the result of a bag-projection. Thus, the three tuples $(1,2,5)$, $(1,2,7)$, and $(1,2,8)$ of the relation R from Fig. 5.2 each gave rise to the same tuple $(1,2)$ after projection onto attributes A and B. In the bag result, there are three occurrences of tuple $(1,2)$, while in the set-projection, this tuple appears only once.

Bag Operations on Sets

Imagine we have two sets R and S. Every set may be thought of as a bag; the bag just happens to have at most one occurrence of any tuple. Suppose we intersect $R \cap S$, but we think of R and S as bags and use the bag intersection rule. Then we get the same result as we would get if we thought of R and S as sets. That is, thinking of R and S as bags, a tuple t is in $R \cap S$ the minimum of the number of times it is in R and S. Since R and S are sets, t can be in each only 0 or 1 times. Whether we use the bag or set intersection rules, we find that t can appear at most once in $R \cap S$, and it appears once exactly when it is in both R and S. Similarly, if we use the bag difference rule to compute $R - S$ or $S - R$ we get exactly the same result as if we used the set rule.

However, union behaves differently, depending on whether we think of R and S as sets or bags. If we use the bag rule to compute $R \cup S$, then the result may not be a set, even if R and S are sets. In particular, if tuple t appears in both R and S, then t appears twice in $R \cup S$ if we use the bag rule for union. But if we use the set rule then t appears only once in $R \cup S$.

5.1.4 Selection on Bags

To apply a selection to a bag, we apply the selection condition to each tuple independently. As always with bags, we do not eliminate duplicate tuples in the result.

Example 5.5: If R is the bag

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

then the result of the bag-selection $\sigma_{C \geq 6}(R)$ is

A	B	C
3	4	6
1	2	7
1	2	7

That is, all but the first tuple meets the selection condition. The last two tuples, which are duplicates in R, are each included in the result. □

Algebraic Laws for Bags

An algebraic law is an equivalence between two expressions of relational algebra whose arguments are variables standing for relations. The equivalence asserts that no matter what relations we substitute for these variables, the two expressions define the same relation. An example of a well-known law is the commutative law for union: $R \cup S = S \cup R$. This law happens to hold whether we regard relation-variables R and S as standing for sets or bags. However, there are a number of other laws that hold when relational algebra is applied to sets but that do not hold when relations are interpreted as bags. A simple example of such a law is the distributive law of set difference over union, $(R \cup S) - T = (R - T) \cup (S - T)$. This law holds for sets but not for bags. To see why it fails for bags, suppose R, S, and T each have one copy of tuple t. Then the expression on the left has one t, while the expression on the right has none. As sets, neither would have t. Some exploration of algebraic laws for bags appears in Exercises 5.1.4 and 5.1.5.

5.1.5 Product of Bags

The rule for the Cartesian product of bags is the expected one. Each tuple of one relation is paired with each tuple of the other, regardless of whether it is a duplicate or not. As a result, if a tuple r appears in a relation R m times, and tuple s appears n times in relation S, then in the product $R \times S$, the tuple rs will appear mn times.

Example 5.6: Let R and S be the bags shown in Fig. 5.3. Then the product $R \times S$ consists of six tuples, as shown in Fig. 5.3(c). Note that the usual convention regarding attribute names that we developed for set-relations applies equally well to bags. Thus, the attribute B, which belongs to both relations R and S, appears twice in the product, each time prefixed by one of the relation names. □

5.1.6 Joins of Bags

Joining bags presents no surprises. We compare each tuple of one relation with each tuple of the other, decide whether or not this pair of tuples joins successfully, and if so we put the resulting tuple in the answer. When constructing the answer, we do not eliminate duplicate tuples.

Example 5.7: The natural join $R \bowtie S$ of the relations R and S seen in Fig. 5.3 is

A	B
1	2
1	2

(a) The relation R

B	C
2	3
4	5
4	5

(b) The relation S

A	$R.B$	$S.B$	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) The product $R \times S$

Figure 5.3: Computing the product of bags

A	B	C
1	2	3
1	2	3

That is, tuple $(1, 2)$ of R joins with $(2, 3)$ of S. Since there are two copies of $(1, 2)$ in R and one copy of $(2, 3)$ in S, there are two pairs of tuples that join to give the tuple $(1, 2, 3)$. No other tuples from R and S join successfully.

As another example on the same relations R and S, the theta-join

$$R \bowtie_{R.B<S.B} S$$

produces the bag

A	$R.B$	$S.B$	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5