PART

C Constraints

As presented so far, there is little in the relational model that can be used to indicate the intended meaning of tuples stored in a database. Although this provides database designers and users with great flexibility, the inability to represent so-called *meta-data* can also lead to a variety of problems. For example, it is hard to develop understandable and usable schemas for complex database applications because of the somewhat simplistic nature of the model. Furthermore, the lack of information on the properties of the stored data often leads to inefficient implementations.

In response to these problems, a framework for adding semantics to the relational model has been developed. This is accomplished by incorporating *integrity constraints* (i.e., properties that are supposed to be satisfied by all instances of a database schema). Some of the early research included the study of integrity constraints expressed by essentially arbitrary sentences from first-order logic. However, feasibility considerations have led to the study of more restricted classes of constraints, usually called *dependencies*. A simple example is that of *key dependency*. For instance, we expect that in a relation concerning personnel data, the Social Security number will serve as a "key" (i.e., the Social Security number will uniquely identify tuples in the relation).

The fundamental motivation for the study of dependencies is to incorporate more semantics into the relational model. An alternative is to develop more sophisticated database models with richer constructs than the relational model, yielding schemas conveying explicitly more of the semantics associated with the data. One family of such models, called semantic data models, and its relationship to the relational model are discussed in Chapter 11. A more general family, called object-oriented database models, is studied in Chapter 21 of Part F.

With the development of new models, dependency theory has become somewhat out of fashion. However, we believe that the results presented in this part remain important. The functional, join, and inclusion dependencies are fundamental to the understanding of the more elaborate constructs found in modern database models. Their study in the simple context of the relational model captures the essence of such constructs. Some results presented in this part have also found applications to various fields, such as deductive databases. Finally, some of the techniques that are developed are interesting in their own right and highlight in a nutshell key aspects of computer science.

A broad theory of dependencies has been developed for the relational model.[1] Several natural kinds of dependencies were introduced, often using an *ad hoc* notation, and studied in depth. Subsequently, a framework stemming primarily from mathematical logic was developed that provides a unifying view of virtually all of them. The single most important theoretical question examined in connection with dependencies is that of (*logical*) *implication*: Given a set Σ of dependencies and a dependency σ, if an instance satisfies Σ does it necessarily satisfy σ as well? The implication problem turns out to be a key technical issue in most situations in which dependencies are used. It has been studied primarily from two perspectives. One is focused on developing algorithms for determining implication and on studying the complexity of this problem for different classes of dependencies. The other perspective is focused on the development of *inference rules* that can be used to construct proofs that a dependency is implied.

Dependencies are intimately related to several other important topics in various database contexts. A driving force for the study of dependencies has been schema design. The goal is to select the most appropriate schema for a particular database application. Under one approach, a schema from a semantic data model is transformed into a relational schema with dependencies. An alternative approach starts with a "universal relation" and applies *decomposition* to create new relations that satisfy certain *normal forms*. Other areas of theoretical research have been to study query optimization techniques in the presence of dependencies and the interaction between dependencies and data restructuring. Results include the use of dependencies to optimize conjunctive queries and the study of how dependencies are carried from a database to a view.

All of the dependencies studied in this part are *static*, in the sense that they describe properties that should be satisfied by all possible instances of a schema regardless of the past and future. Research has also been performed in connection with *dynamic dependencies*, which describe properties of the evolution of the database; these are considered briefly in Chapter 22.

The first chapter of this part presents the practical motivations for the incorporation of dependencies into the relational model and then examines the basic themes of dependency theory in connection with two fundamental kinds of dependencies (namely, *functional* and *join* dependencies). This chapter also introduces the *chase*, an elegant tool for optimizing conjunctive queries in the presence of dependencies and for determining implication for dependencies. The second chapter introduces *inclusion dependencies*. Although positive results are obtained for these dependencies considered in isolation, several negative results hold when they are considered with functional dependencies. The third chapter presents a unifying framework for dependencies based on a perspective from mathematical logic. The final chapter of this part considers dependency theory in connection with several issues of database design.

[1] To keep theoreticians in business, some say.

8 Functional and Join Dependency

Alice: *Your model reduces the most interesting information to something flat and boring.*

Vittorio: *You're right, and this causes a lot of problems.*

Sergio: *Designing the schema for a complex application is tough, and it is easy to make mistakes when updating a database.*

Riccardo: *Also, the system knows so little about the data that it is hard to obtain good performance.*

Alice: *Are you telling me that the model is bad?*

Vittorio: *No, wait, we are going to fix it!*

This chapter begins with an informal discussion that introduces some simple dependencies and illustrates the primary motivations for their development and study. The two following sections of the chapter are devoted to two of the simple kinds of dependencies; and the final section introduces the chase, an important tool for analyzing these dependencies and their effect on queries.

Many of the early dependencies introduced in the literature use the named (as opposed to unnamed) perspective on tuples and relations. Dependency theory was one of the main reasons for adopting this perspective in theoretical investigations. This is because dependencies concern the semantics of data, and attribute names carry more semantics than column numbers. The general view of dependencies based on logic, which is considered in Chapter 10, uses the column-number perspective, but a special subcase (called *typed*) retains the spirit of the attribute-name perspective.

8.1 Motivation

Consider the database shown in Fig. 8.1. Although the schema itself makes no restrictions on properties of data that might be stored, the intended application for the schema may involve several such restrictions. For example, we may know that there is only one director associated with each movie title, and that in *Showings*, only one movie title is associated with a given theater-screen pair.[1] Such properties are called *functional dependencies* (fd's) because the values of some attributes of a tuple uniquely or *functionally* determine the values of other attributes of that tuple. In the syntax to be developed in this chapter, the

[1] Gone are the days of seeing two movies for the price of one!

Movies

Title	*Director*	*Actor*
The Birds	Hitchcock	Hedren
The Birds	Hitchcock	Taylor
Bladerunner	Scott	Hannah
Apocalypse Now	Coppola	Brando

Showings

Theater	*Screen*	*Title*	*Snack*
Rex	1	The Birds	coffee
Rex	1	The Birds	popcorn
Rex	2	Bladerunner	coffee
Rex	2	Bladerunner	popcorn
Le Champo	1	The Birds	tea
Le Champo	1	The Birds	popcorn
Cinoche	1	The Birds	Coke
Cinoche	1	The Birds	wine
Cinoche	2	Bladerunner	Coke
Cinoche	2	Bladerunner	wine
Action Christine	1	The Birds	tea
Action Christine	1	The Birds	popcorn

Figure 8.1: Sample database illustrating simple dependencies

dependency in the *Movies* relation is written as

$$Movies : Title \rightarrow Director$$

and that of the *Showings* relation is written as

$$Showings : Theater\ Screen \rightarrow Title.$$

Technically, there are sets of attributes on the left- and right-hand sides of the arrow, but we continue with the convention of omitting set braces when understood from the context.

When there is no confusion from the context, a dependency $R : X \rightarrow Y$ is simply denoted $X \rightarrow Y$. A relation I *satisfies* a functional dependency $X \rightarrow Y$ if for each pair s, t of tuples in I,

$$\pi_X(s) = \pi_X(t) \text{ implies } \pi_Y(s) = \pi_Y(t).$$

An important notion in dependency theory is *implication*. One can observe that any relation satisfying the dependency

(a) $$Title \rightarrow Director$$

also has to satisfy the dependency

(b) $$Title, Actor \rightarrow Director.$$

We will say that dependency (a) implies dependency (b).

A *key dependency* is an fd $X \rightarrow U$, where U is the full set of attributes of the relation. It turns out that dependency (b) is equivalent to the key dependency *Title, Actor* $\rightarrow$ *Title*, *Director*, *Actor*.

A second fundamental kind of dependency is illustrated by the relation *Showings*. A tuple (*th*, *sc*, *ti*, *sn*) is in *Showings* if theater *th* is showing movie *ti* on screen *sc* and if theater *th* offers snack *sn*. Intuitively, one would expect a certain independence between the *Screen-Title* attributes, on the one hand, and the *Snack* attribute, on the other, for a given value of *Theater*. For example, because (Cinoche, 1, The Birds, Coke) and (Cinoche, 2, Bladerunner, wine) are in *Showings*, we also expect (Cinoche, 1, The Birds, wine) and (Cinoche, 2, Bladerunner, Coke) to be present. More precisely, if a relation I has this property, then

$$I = \pi_{Theater,Screen,Title}(I) \bowtie \pi_{Theater,Snack}(I).$$

This is a simple example of a *join dependency* (jd) which is formally expressed by

$$Showings : \bowtie[\{Theater, Screen, Title\}, \{Theater, Snacks\}].$$

In general, a jd may involve more than two attribute sets. *Multivalued dependency* (mvd) is the special case of jd's that have at most two attribute sets. Due to their naturalness, mvd's were introduced before jd's and have several interesting properties, which makes them worth studying on their own.

As will be seen later in this chapter, the fact that the fd *Title* $\rightarrow$ *Director* is satisfied by the *Movies* relation implies that the jd

$$\bowtie[\{Title, Director\}, \{Title, Actor\}]$$

is also satisfied. We will also study such interaction between fd's and jd's.

So far we have considered dependencies that apply to individual relations. Typically these dependencies are used in the context of a database schema, in which case one has to specify the relation concerned by each dependency. We will also consider a third fundamental kind of dependency, called *inclusion dependency* (ind) and also referred to as "referential constraint." In the example, we might expect that each title currently being shown (i.e., occurring in the *Showings* relation) is the title of a movie (i.e., also occurs in the *Movies* relation). This is denoted by

$$Showings[Title] \subseteq Movies[Title].$$

In general, ind's may involve sequences of attributes on both sides. Inclusion dependencies will be studied in depth in Chapter 9.

Data dependencies such as the ones just presented provide a formal mechanism for expressing properties expected from the stored data. If the database is known to satisfy a set of dependencies, this information can be used to (1) improve schema design, (2) protect data by preventing certain erroneous updates, and (3) improve performance. These aspects are considered in turn next.

Schema Design and Update Anomalies

The task of designing the schema in a large database application is far from being trivial, so the designer has to receive support from the system. Dependencies are used to provide information about the semantics of the application so that the system may help the user choose, among all possible schemas, the most appropriate one.

There are various ways in which a schema may not be appropriate. The relations *Movies* and *Showings* illustrate the most prominent kinds of problems associated with fd's and jd's:

Incomplete information: Suppose that one is to insert the title of a new movie and its director without knowing yet any actor of the movie. This turns out to be impossible with the foregoing schema, and it is an *insertion anomaly*. An analogue for deletion, a *deletion anomaly*, occurs if actor Marlon Brando is no longer associated with the movie "Apocalypse Now." Then the tuple ⟨Apocalypse Now, Coppola, Brando⟩ should be deleted from the database. But this has the additional effect of deleting the association between the movie "Apocalypse Now" and the director Coppola from the database, information that may still be valid.

Redundancy: The fact that Coke can be found at the Cinoche is recorded many times. Furthermore, suppose that the management of the Cinoche decided to sell Pepsi instead of Coke. It is not sufficient to modify the tuple ⟨Cinoche, 1, The Birds, Coke⟩ to ⟨Cinoche, 1, The Birds, Pepsi⟩ because this would lead to a violation of the jd. We have to modify several tuples. This is a *modification anomaly*. Insertion and deletion anomalies are also caused by redundancy.

Thus because of a bad choice for the schema, updates can lead to loss of information, inconsistency in the data, and more difficulties in writing correct updates. These problems can be prevented by choosing a more appropriate schema. In the example, the relation *Movies* should be "decomposed" into two relations *M-Director*[*Title*, *Director*] and *M-Actor*[*Title*, *Actor*], where *M-Director* satisfies the fd *Title* $\rightarrow$ *Director*. Similarly, the relation *Showings* should be replaced by two relations *ST-Showings*[*Theater*, *Screen*, *Title*] and *S-Showings*[*Theater*, *Snack*], where *ST-Showings* satisfies the fd *Theater, Screen* $\rightarrow$ *Title*. This approach to schema design is explored in Chapter 11.

Data Integrity

Data dependencies also serve as a filter on proposed updates in a natural fashion: If a database is expected to satisfy a dependency σ and a proposed update would lead to the

violation of σ, then the update is rejected. In fact, the system supports transactions. During a transaction, the database can be in an inconsistent state; but at the end of a transaction, the system checks the integrity of the database. If dependencies are violated, the whole transaction is rejected (*aborted*); otherwise it is accepted (*validated*).

Efficient Implementation and Query Optimization

It is natural to expect that knowledge of structural properties of the stored data be useful in improving the performances of a system for a particular application.

At the physical level, the satisfaction of dependencies leads to a variety of alternatives for storage and access structures. For example, satisfaction of an fd or jd implies that a relation can be physically stored in decomposed form. In addition, satisfaction of a key dependency can be used to reduce indexing space.

A particularly striking theoretical development in dependency theory provides a method for optimizing conjunctive queries in the presence of a large class of dependencies. As a simple example, consider the query

$$ans(d, a) \leftarrow Movies(t, d, a'), Movies(t, d', a),$$

which returns tuples $\langle d, a\rangle$, where actor a acted in a movie directed by d. A naive implementation of this query will require a join. Because *Movies* satisfies *Title* $\rightarrow$ *Director*, this query can be simplified to

$$ans(d, a) \leftarrow Movies(t, d, a),$$

which can be evaluated without a join. Whenever the pattern of tuples $\{\langle t, d, a'\rangle, \langle t, d', a\rangle\}$ is found in relation *Movies*, it must be the case that $d = d'$, so one may as well use just the pattern $\{\langle t, d, a\rangle\}$, yielding the simplified query. This technique for query optimization is based on the chase and is considered in the last section of this chapter.

8.2 Functional and Key Dependencies

Functional dependencies are the most prominent form of dependency, and several elegant results have been developed for them. Key dependencies are a special case of functional dependencies. These are the dependencies perhaps most universally supported by relational systems and used in database applications. Many issues in dependency theory have nice solutions in the context of functional dependencies, and these dependencies lie at the origin of the decomposition approach to schema design.

To specify a class of dependencies, one must define the syntax and the semantics of the dependencies of concern. This is done next for fd's.

DEFINITION 8.2.1 If U is a set of attributes, then a *functional dependency* (fd) over U is an expression of the form $X \rightarrow Y$, where $X, Y \subseteq U$. A *key dependency* over U is an fd of the form $X \rightarrow U$. A relation I over U *satisfies* $X \rightarrow Y$, denoted $I \models X \rightarrow Y$, if for each

pair s, t of tuples in I, $\pi_X(s) = \pi_X(t)$ implies $\pi_Y(s) = \pi_Y(t)$. For a set Σ of fd's, I *satisfies* Σ, denoted $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$.

A functional dependency over a database schema $\mathbf{R}$ is an expression $R : X \to Y$, where $R \in \mathbf{R}$ and $X \to Y$ is a dependency over $sort(R)$. These are sometimes referred to as *tagged* dependencies, because they are "tagged" by the relation that they apply to. The notion of satisfaction of fd's by instances over $\mathbf{R}$ is defined in the obvious way. In the remainder of this chapter, we consider only relational schemas. All can be extended easily to database schemas.

The following simple property provides the basis for the decomposition approach to schema design. Intuitively, it says that if a certain fd holds in a relation, one can store instead of the relation two projections of it, without loss of information. More precisely, the original relation can be reconstructed by joining the projections. Such joins have been termed "lossless joins" and will be discussed in some depth in Section 11.2.

PROPOSITION 8.2.2 Let I be an instance over U that satisfies $X \to Y$ and $Z = U - XY$. Then $I = \pi_{XY}(I) \bowtie \pi_{XZ}(I)$.

Proof The inclusion $I \subseteq \pi_{XY}(I) \bowtie \pi_{XZ}(I)$ holds for all instances I. For the opposite inclusion, let r be a tuple in the join. Then there are tuples $s, t \in I$ such that $\pi_{XY}(r) = \pi_{XY}(s)$ and $\pi_{XZ}(r) = \pi_{XZ}(t)$. Because $\pi_X(r) = \pi_X(t)$, and $I \models X \to Y$, $\pi_Y(r) = \pi_Y(t)$. It follows that $r = t$, so r is in I. ■

Logical Implication

In general, we may know that a set Σ of fd's is satisfied by an instance. A natural question is, What other fd's are necessarily satisfied by this instance? This is captured by the following definition.

DEFINITION 8.2.3 Let Σ and Γ be sets of fd's over an attribute set U. Then Σ (*logically*) *implies* Γ, denoted $\Sigma \models_U \Gamma$ or simply $\Sigma \models \Gamma$, if U is understood from the context, if for all relations I over U, $I \models \Sigma$ implies $I \models \Gamma$. Two sets Γ, Σ are (*logically*) *equivalent*, denoted $\Gamma \equiv \Sigma$, if $\Gamma \models \Sigma$ and $\Sigma \models \Gamma$.

EXAMPLE 8.2.4 Consider the set $\Sigma_1 = \{A \to C, B \to C, CD \to E\}$ of fd's over $\{A, B, C, D, E\}$. Then[2] a simple argument allows to show that $\Sigma_1 \models AD \to E$. In addition, $\Sigma_1 \models CDE \to C$. In fact, $\emptyset \models CDE \to C$ (where $\emptyset$ is the empty set of fd's).

Although the definition just presented focuses on fd's, this definition will be used in connection with other classes of dependencies studied here as well.

---

[2] We generally omit set braces from singleton sets of fd's.

The *fd closure* of a set Σ of fd's over an attribute set U, denoted $\Sigma^{*,U}$ or simply Σ^* if U is understood from the context, is the set

$$\{X \to Y \mid XY \subseteq U \text{ and } \Sigma \models X \to Y\}.$$

It is easily verified that for any set Σ of fd's over U and any sets $Y \subseteq X \subseteq U$, $X \to Y \in \Sigma^{*,U}$. This implies that the closure of a set of fd's depends on the underlying set of attributes. It also implies that $\Sigma^{*,U}$ has size greater than $2^{|U|}$. (It is bounded by $2^{2|U|}$ by definition.) Other properties of fd closures are considered in Exercise 8.3.

Determining Implication for fd's Is Linear Time

One of the key issues in dependency theory is the development of algorithms for testing logical implication. Although a set Σ of fd's implies an exponential (in terms of the number of attributes present in the underlying schema) number of fd's, it is possible to test whether Σ implies an fd $X \to Y$ in time that is linear in the size of Σ and $X \to Y$ (i.e., the space needed to write them).

A central concept used in this algorithm is the *fd closure* of a set of attributes. Given a set Σ of fd's over U and attribute set $X \subseteq U$, the fd closure of X under Σ, denoted $(X, \Sigma)^{*,U}$ or simply X^* if Σ and U are understood, is the set $\{A \in U \mid \Sigma \models X \to A\}$. It turns out that this set is independent of the underlying attribute set U (see Exercise 8.6).

Example 8.2.5 Recall the set Σ_1 of fd's from Example 8.2.4. Then $A^* = AC$, $(AB)^* = ABC$, and $(AD)^* = ACDE$. The family of subsets X of U such that $X^* = X$ is $\{\emptyset, C, D, E, AC, BC, CE, DE, ABC, ACE, ADE, BCE, BDE, CDE, ABCE, ACDE, BCDE, ABCDE\}$.

The following is easily verified (see Exercise 8.4):

Lemma 8.2.6 Let Σ be a set of fd's and $X \to Y$ an fd. Then $\Sigma \models X \to Y$ iff $Y \subseteq X^*$.

Thus testing whether $\Sigma \models X \to Y$ can be accomplished by computing X^*. The following algorithm can be used to compute this set.

Algorithm 8.2.7

Input: a set Σ of fd's and a set X of attributes.

Output: the closure X^* of X under Σ.

1. *unused* := Σ;
2. *closure* := X;
3. repeat until no further change:
   if $W \to Z \in$ *unused* and $W \subseteq$ *closure* then
      i. *unused* := *unused* $- \{W \to Z\}$;
      ii. *closure* := *closure* $\cup Z$
4. output *closure*.

PROPOSITION 8.2.8 On input Σ and X, Algorithm 8.2.7 computes $(X, \Sigma)^*$.

Proof Let U be a set of attributes containing the attributes occurring in Σ or X, and let *result* be the output of the algorithm. Using properties established in Exercise 8.5, an easy induction shows that $result \subseteq X^*$.

For the opposite inclusion, note first that for attribute sets Y, Z, if $Y \subseteq Z$ then $Y^* \subseteq Z^*$. Because $X \subseteq result$, it now suffices to show that $result^* \subseteq result$. It is enough to show that if $A \in U - result$, then $\Sigma \not\models result \to A$. To show this, we construct an instance I over U such that $I \models \Sigma$ but $I \not\models result \to A$ for $A \in U - result$. Let $I = \{s, t\}$, where $\pi_{result}(s) = \pi_{result}(t)$ and $s(A) \neq t(A)$ for each $A \in U - result$. (Observe that this uses the fact that the domain has at least two elements.) Note that, by construction, for each fd $W \to Z \in \Sigma$, if $W \subseteq result$ then $Z \subseteq result$. It easily follows that $I \models \Sigma$. Furthermore, for $A \in U - result$, $s(A) \neq t(A)$, so $I \not\models result \to A$. Thus $\Sigma \not\models result \to A$, and $result^* \subseteq result$. ■

The algorithm provides the means for checking whether a set of dependencies implies a single dependency. To test implication of a *set* of dependencies, it suffices to test independently the implication of each dependency in the set. In addition, one can check that the preceding algorithm runs in time $O(n^2)$, where n is the length of Σ and X. As shown in Exercise 8.7, this algorithm can be improved to linear time. The following summarizes this development.

THEOREM 8.2.9 Given a set Σ of fd's and a single fd σ, determine whether $\Sigma \models \sigma$ can be decided in linear time.

Several interesting properties of fd-closure sets are considered in Exercises 8.11 and 8.12.

Axiomatization for fd's

In addition to developing algorithms for determining logical implication, the second fundamental theme in dependency theory has been the development of inference rules, which can be used to generate symbolic proofs of logical implication. Although the inference rules do not typically yield the most efficient mechanisms for deciding logical implication, in many cases they capture concisely the essential properties of the dependencies under study. The study of inference rules is especially intriguing because (as will be seen in the next section) there are several classes of dependencies for which there is no finite set of inference rules that characterizes logical implication.

Inference rules and algorithms for testing implication provide alternative approaches to showing logical implication between dependencies. In general, the existence of a finite set of inference rules for a class of dependencies is a stronger property than the existence of an algorithm for testing implication. It will be shown in Chapter 9 that

- the existence of a finite set of inference rules for a class of dependencies implies the existence of an algorithm for testing logical implication; and

- there are dependencies for which there is no finite set of inference rules but for which there is an algorithm to test logical implication.

We now present the inference rules for fd's.

FD1: (reflexivity) If $Y \subseteq X$, then $X \to Y$.

FD2: (augmentation) If $X \to Y$, then $XZ \to YZ$.

FD3: (transitivity) If $X \to Y$ and $Y \to Z$, then $X \to Z$.

The variables X, Y, Z range over sets of attributes. The first rule is sometimes called an *axiom* because it is degenerate in the sense that no fd's occur in the antecedent.

The inference rules are used to form proofs about logical implication between fd's, in a manner analogous to the proofs found in mathematical logic. It will be shown that the resulting proof system is "sound" and "complete" for fd's (two classical notions to be recalled soon). Before formally presenting the notion of proof, we give an example.

EXAMPLE 8.2.10 The following is a proof of $AD \to E$ from the set Σ_1 of fd's of Example 8.2.4.

σ_1 :	$A \to C$	$\in \Sigma_1$,
σ_2 :	$AD \to CD$	from σ_1 using FD2,
σ_3 :	$CD \to E$	$\in \Sigma_1$,
σ_4 :	$AD \to E$	from σ_2 and σ_3 using FD3.

Let U be a set of attributes. A *substitution* for an inference rule ρ (relative to U) is a function that maps each variable appearing in ρ to a subset of U, such that each set inclusion indicated in the antecedent of ρ is satisfied by the associated sets. Now let Σ be a set of fd's over U and σ an fd over U. A *proof* of σ from Σ using the set $\mathcal{I} = \{$FD1, FD2, FD3$\}$ is a sequence of fd's $\sigma_1, \ldots, \sigma_n = \sigma$ $(n \geq 1)$ such that for each $i \in [1, n]$, either

(a) $\sigma_i \in \Sigma$, or

(b) there is a substitution for some rule $\rho \in \mathcal{I}$ such that σ_i corresponds to the consequent of ρ, and such that for each fd in the antecedent of ρ the corresponding fd is in the set $\{\sigma_j \mid 1 \leq j < i\}$.

The fd σ is *provable* from Σ using $\mathcal{I}$ (relative to U), denoted $\Sigma \vdash_{\mathcal{I}} \sigma$ or $\Sigma \vdash \sigma$ if $\mathcal{I}$ is understood from the context, if there is a proof of σ from Σ using $\mathcal{I}$.

Let $\mathcal{I}$ be a set of inference rules. Then

$\mathcal{I}$ is *sound* for logical implication of fd's if $\Sigma \vdash_{\mathcal{I}} \sigma$ implies $\Sigma \models \sigma$,

$\mathcal{I}$ is *complete* for logical implication of fd's if $\Sigma \models \sigma$ implies $\Sigma \vdash_{\mathcal{I}} \sigma$.

We will generalize these definitions to other dependencies and other sets of inference rules.

In general, a finite sound and complete set of inference rules for a class $\mathcal{C}$ of dependencies is called a (finite) *axiomatization* of $\mathcal{C}$. In such a case, $\mathcal{C}$ is said to be (finitely) *axiomatizable*.

We now state the following:

THEOREM 8.2.11 The set {FD1, FD2, FD3} is sound and complete for logical implication of fd's.

Proof Suppose that Σ is a set of fd's over an attribute set U. The proof of soundness involves a straightforward induction on proofs $\sigma_1, \ldots, \sigma_n$ from Σ, showing that $\Sigma \models \sigma_i$ for each $i \in [1, n]$ (see Exercise 8.5).

For the proof of completeness, we show that $\Sigma \models X \to Y$ implies $\Sigma \vdash X \to Y$. As a first step, we show that $\Sigma \vdash X \to X^*$ using an induction based on Algorithm 8.2.7. In particular, let $closure_i$ be the value of *closure* after i iterations of step 3 for some fixed execution of that algorithm on input Σ and X. We set $closure_0 = X$. Suppose inductively that a proof $\sigma_1, \ldots, \sigma_{k_i}$ of $X \to closure_i$ has been constructed. [The case for $i = 0$ follows from FD1.] Suppose further that $W \to Z$ is chosen for the $(i+1)^{\text{st}}$ iteration. It follows that $W \subseteq closure_i$ and $closure_{i+1} = closure_i \cup Z$. Extend the proof by adding the following steps:

σ_{k_i+1}	=	$W \to Z$	in Σ
σ_{k_i+2}	=	$closure_i \to W$	by FD1
σ_{k_i+3}	=	$closure_i \to Z$	by FD3
σ_{k_i+4}	=	$closure_i \to closure_{i+1}$	by FD2
σ_{k_i+5}	=	$X \to closure_{i+1}$	by FD3

At the completion of this construction we have a proof $\sigma_1, \ldots, \sigma_n$ of $X \to X^*$. By Lemma 8.2.6, $Y \subseteq X^*$. Using FD1 and FD3, the proof can be extended to yield a proof of $X \to Y$. ■

Other inference rules for fd's are considered in Exercise 8.9.

Armstrong Relations

In the proof of Proposition 8.2.8, an instance I is created such that $I \models \Sigma$ but $I \not\models X \to A$. Intuitively, this instance witnesses the fact that $\Sigma \not\models X \to A$. This raises the following natural question: Given a set Σ of fd's over U, is there a *single* instance I that satisfies Σ and that violates every fd not in Σ^*? It turns out that for each set of fd's, there is such an instance; these are called *Armstrong relations*.

PROPOSITION 8.2.12 If Σ is a set of fd's over U, then there is an instance I such that, for each fd σ over U, $I \models \sigma$ iff $\sigma \in \Sigma^*$.

Crux Suppose first that $\Sigma \not\models \emptyset \to A$ for any A (i.e., $\emptyset^* = \emptyset$). For each set $X \subseteq U$ satisfying $X = X^*$, choose an instance $I_X = \{s_X, t_X\}$ such that $s_X(A) = t_X(A)$ iff $A \in X$. In addition, choose these instances so that $adom(I_X) \cap adom(I_Y) = \emptyset$ for $X \neq Y$. Then

$$\cup\{I_X \mid X \subset U \text{ and } X = X^*\}$$

is an Armstrong relation for Σ.

If $\emptyset^* \neq \emptyset$, then the instances I_X should be modified so that $\pi_A(I_X) = \pi_A(I_Y)$ for each X, Y and $A \in \emptyset^*$. ■

In some applications, the domains of certain attributes may be finite (e.g., *Sex* conventionally has two values, and *Grade* typically consists of a finite set of values). In such cases, the construction of an Armstrong relation may not be possible. This is explored in Exercise 8.13.

Armstrong relations can be used in practice to assist the user in specifying the fd's for a particular application. An interactive, iterative specification process starts with the user specifying a first set of fd's. The system then generates an Armstrong relation for the fd's, which violates all the fd's not included in the specification. This serves as a worst-case counterexample and may result in detecting additional fd's whose satisfaction should be required.

8.3 Join and Multivalued Dependencies

The second kind of simple dependency studied in this chapter is *join dependency* (jd), which is intimately related to the join operator of the relational algebra. As mentioned in Section 8.1, a basic motivation for join dependency stems from its usefulness in connection with relation decomposition. This section also discusses *multivalued dependency* (mvd), an important special case of join dependency that was historically the first to be introduced.

The central results and tools for studying jd's are different from those for fd's. It has been shown that there is no sound and complete set of inference rules for jd's analogous to those for fd's. (An axiomatization for a much larger family of dependencies will be presented in Chapter 10.) In addition, as shown in the following section, logical implication for jd's is decidable. The complexity of implication is polynomial for a fixed database schema but becomes NP-hard if the schema is considered part of the input. (An exact characterization of the complexity remains open.)

The following section also presents an interesting correspondence between mvd's and acyclic join dependencies (i.e., those based on joins that are acyclic in the sense introduced in Chapter 6).

A major focus of the current section is on mvd's; this is because of several positive results that hold for them, including axiomatizability of fd's and mvd's considered together.

Join Dependency and Decomposition

Before defining join dependency, we recall the definition of natural join. For attribute set U, sets $X_1, \ldots, X_n \subseteq U$, and instances I_j over X_j for $j \in [1, n]$, the (*natural*) *join* of the I_j's is

$$\bowtie_{j=1}^{n} \{I_j\} = \{s \text{ over } \cup X_j \mid \pi_{X_j}(s) \in I_j \text{ for each } j \in [1, n]\}.$$

A join dependency is satisfied by an instance I if it is equal to the join of some of its projections.

DEFINITION 8.3.1 A *join dependency* (jd) over attribute set U is an expression of the form $\bowtie[X_1, \ldots, X_n]$, where $X_1, \ldots, X_n \subseteq U$ and $\cup_{i=1}^{n} X_i = U$. A relation I over U *satisfies* $\bowtie[X_1, \ldots, X_n]$ if $I = \bowtie_{j=1}^{n} \{\pi_{X_j}(I)\}$.

A jd σ is *n-ary* if the number of attribute sets involved in σ is n. As discussed earlier, the relation *Showings* of Fig. 8.1 satisfies the 2-ary jd

$$\bowtie[\{Theater, Screen, Title\}, \{Theater, Snacks\}].$$

The 2-ary jd's are also called *multivalued dependencies* (mvd's). These are often denoted in a style reminiscent of fd's.

DEFINITION 8.3.2 If U is a set of attributes, then a *multivalued dependency* (mvd) over U is an expression of the form $X \twoheadrightarrow Y$, where $X, Y \subseteq U$. A relation I over U *satisfies* $X \twoheadrightarrow Y$ if $I \models \bowtie[XY, X(U - Y)]$.

In the preceding definition, it would be equivalent to write $\bowtie[XY, (U - Y)]$; we choose the foregoing form to emphasize the importance of X. For instance, the jd

$$\bowtie[\{Theater, Screen, Title\}, \{Theater, Snack\}]$$

can be written as an mvd using

$$Theater \twoheadrightarrow Screen, Title, \quad \text{or equivalently,} \quad Theater \twoheadrightarrow Snack.$$

Exercise 8.16 explores the original definition of satisfaction of an mvd.

Figure 8.2 shows a relation schema SDT and an instance that satisfies a 3-ary jd. This relation focuses on snacks, distributors, and theaters. We assume for this example that a tuple (s, d, p, t) is in SDT if the conjunction of the following predicates is true:

$P_1(s, d, p)$:	Snack s is supplied by distributor d at price p.
$P_2(d, t)$:	Theater t is a customer of distributor d.
$P_3(s, t)$:	Snack s is bought by theater t.

Under these assumptions, each instance of SDT must satisfy the jd:

$$\bowtie[\{Snack, Distributor, Price\}, \{Distributor, Theater\}, \{Snack, Theater\}].$$

For example, this holds for the instance in Fig. 8.2. Note that if tuple ⟨coffee, Smart, 2.35, Cinoche⟩ were removed, then the instance would no longer satisfy the jd because ⟨coffee, Smart, 2.35⟩, ⟨coffee, Cinoche⟩, and ⟨Smart, Cinoche⟩ would remain in the appropriate projections. We also expect the instances of SDT to satisfy $Snack, Distributor \rightarrow Price$.

It can be argued that schema *SDT* with the aforementioned constraint is unnatural in the following sense. Intuitively, if we choose such a schema, the presence of a tuple

SDT	*Snack*	*Distributor*	*Price*	*Theater*
	coffee	Smart	2.35	Rex
	coffee	Smart	2.35	Le Champo
	coffee	Smart	2.35	Cinoche
	coffee	Leclerc	2.60	Cinoche
	wine	Smart	0.80	Rex
	wine	Smart	0.80	Cinoche
	popcorn	Leclerc	5.60	Cinoche

Figure 8.2: Illustration of join dependency

$\langle s, d, p, t\rangle$ seems to indicate that t buys s from d. If we wish to record just the information about who buys what, who sells what, and who sells to whom, a more appropriate schema would consist of three relations $SD[Snack, Distributor, Price]$, $ST[Snack, Theater]$, and $DT[Distributor, Theater]$ corresponding to the three sets of attributes involved in the preceding jd. The jd then guarantees that no information is lost in the decomposition because the original relation can be reconstructed by joining the projections.

Join Dependencies and Functional Dependencies

The interaction of fd's and jd's is important in the area of schema design and user interfaces to the relational model. Although this is explored in more depth in Chapter 11, we present here one of the first results on the interaction of the two kinds of dependencies.

Proposition 8.3.3 Let U be a set of attributes, $\{X, Y, Z\}$ be a partition of U, and Σ be a set of fd's over U. Then $\Sigma \models \bowtie[XY, XZ]$ iff either $\Sigma \models X \to Y$ or $\Sigma \models X \to Z$.

Crux Sufficiency follows immediately from Proposition 8.2.2. For necessity, suppose that Σ does not imply either of the fd's. Then $Y - X^* \neq \emptyset$ and $Z - X^* \neq \emptyset$, say $C \in Y - X^*$ and $C' \in Z - X^*$. Consider the two-element instance $I = \{u, v\}$ where, $u(A) = v(A) = 0$ if A is in X^* and $u(A) = 0$, $v(A) = 1$ otherwise. Clearly, I satisfies Σ and one can verify that $\pi_{XY}(I) \bowtie \pi_{XZ}(I)$ contains a tuple w with $w(C) = 0$ and $w(C') = 1$. Thus w is not in I, so I violates $\bowtie[XY, XZ]$. ■

Axiomatizations

As will be seen later (Theorem 8.4.12), there is a decision procedure for jd's in isolation, and for jd's and fd's considered together. Here we consider axiomatizations, first for jd's in isolation and then for fd's and mvd's taken together.

We state first the following result without proof.

Theorem 8.3.4 There is no axiomatization for the family of jd's.

In contrast, there is an axiomatization for the class of fd's and multivalued dependencies. Note first that implication for fd's is independent of the underlying set of attributes (i.e., if $\Sigma \cup \{\sigma\}$ is a set of fd's over U and $V \supseteq U$, then $\Sigma \models \sigma$ relative to U iff $\Sigma \models \sigma$ relative to V; see Exercise 8.6). An important difference between fd's and mvd's is that this is not the case for mvd's. Thus the inference rules for mvd's must be used in connection with a fixed underlying set of attributes, and a variable (denoted U) referring to this set is used in one of the rules.

The following lists the four rules for mvd's alone and an additional pair of rules needed when fd's are incorporated.

MVD0: (complementation) If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (U - Y)$.

MVD1: (reflexivity) If $Y \subseteq X$, then $X \twoheadrightarrow Y$.

MVD2: (augmentation) If $X \twoheadrightarrow Y$, then $XZ \twoheadrightarrow YZ$.

MVD3: (transitivity) If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.

FMVD1: (conversion) If $X \to Y$, then $X \twoheadrightarrow Y$.

FMVD2: (interaction) If $X \twoheadrightarrow Y$ and $XY \to Z$, then $X \to (Z - Y)$.

Theorem 8.3.5 The set {FD1, FD2, FD3, MVD0, MVD1, MVD2, MVD3, FMVD1, FMVD2} is sound and complete for logical implication of fd's and mvd's considered together.

Crux Soundness is easily verified. For completeness, let an underlying set U of attributes be fixed, and assume that $\Sigma \not\vdash \sigma$, where $\sigma = X \to Y$ or $\sigma = X \twoheadrightarrow Y$.

The *dependency set* of X is $dep(X) = \{Y \subseteq U \mid \Sigma \vdash X \twoheadrightarrow Y\}$. One first shows that

1. $dep(X)$ is a Boolean algebra of sets for U.

That is, it contains U and is closed under intersection, union, and difference (see Exercise 8.17). In addition,

2. for each $A \in X^+$, $\{A\} \in dep(X)$,

where X^+ denotes $\{A \in U \mid \Sigma \vdash X \to A\}$.

A *dependency basis* of X is a family $\{W_1, \ldots, W_m\} \subseteq dep(X)$ such that (1) $\cup_{i=1}^{n} W_i = U$; (2) $W_i \neq \emptyset$ for $i \in [1, n]$; (3) $W_i \cap W_j = \emptyset$ for $i, j \in [1, n]$ with $i \neq j$; and (4) if $W \in dep(X)$, $W \neq \emptyset$, and $W \subseteq W_i$ for some $i \in [1, n]$, then $W = W_i$. One then proves that

3. there exists a unique dependency basis of X.

Now construct an instance I over U that contains all tuples t satisfying the following conditions:

(a) $t(A) = 0$ for each $A \in X^+$.

(b) If W_i is in the dependency basis and $W_i \neq \{A\}$ for each $A \in X^+$, then $t(B) = 0$ for all $B \in W_i$ or $t(B) = 1$ for all $B \in W_i$.

It can be shown that $I \models \Sigma$ but $I \not\models \sigma$ (see Exercise 8.17). ■

This easily implies the following (see Exercise 8.18):

COROLLARY 8.3.6 The set {MVD0, MVD1, MVD2, MVD3} is sound and complete for logical implication of mvd's considered alone.

8.4 The Chase

This section presents the chase, a remarkable tool for reasoning about dependencies that highlights a strong connection between dependencies and tableau queries. The discussion here is cast in terms of fd's and jd's, but as will be seen in Chapter 10, the chase generalizes naturally to a broader class of dependencies. At the end of this section, we explore important applications of the chase technique. We show how it can also be used to determine logical implication between sets of dependencies and to optimize conjunctive queries.

The following example illustrates an intriguing connection between dependencies and tableau queries.

EXAMPLE 8.4.1 Consider the tableau query (T, t) shown in Fig. 8.3(a). Suppose the query is applied only to instances I satisfying some set Σ of fd's and jd's. The chase is based on the following simple idea. If ν is a valuation embedding T into an instance I satisfying Σ, $\nu(T)$ must satisfy Σ. Valuations that do not satisfy Σ are therefore of no use. The chase is a procedure that eliminates the useless valuations by changing (T, t) itself so that T, viewed as an instance, satisfies Σ. We will show that the tableau query resulting from the chase is then equivalent to the original on instances satisfying Σ. As we shall see, this can be used to optimize queries and test implication of dependencies.

Let us return to the example. Suppose first that $\Sigma = \{B \to D\}$. Suppose (T, t) is applied to an instance I satisfying Σ. In each valuation embedding T into I, it must be the case that z and z' are mapped to the same constant. Thus in this context one might as well replace T by the tableau where $z = z'$. This transformation is called "applying the fd $B \to D$" to (T, t). It is easy to see that the resulting tableau query is in fact equivalent to the identity, because T contains an entire row of distinguished variables.

Consider next an example involving both fd's and jd's. Let Σ consist of the following two dependencies over $ABCD$: the jd $\bowtie[AB, BCD]$ and the fd $A \to C$. In this example we argue that for each I satisfying these dependencies, $(T, t)(I) = I$ or, in other words, in the context of input instances that satisfy the dependencies, the query (T, t) is equivalent to the identity query $(\{t\}, t)$.

Let I be an instance over $ABCD$ satisfying the two dependencies. We first explain why $(T, t)(I) = (T', t)(I)$ for the tableau query (T', t) of Fig. 8.3(b). It is clear that $(T', t)(I) \subseteq (T, t)(I)$, because T' is a superset of T. For the opposite inclusion, suppose that ν is a valuation for T with $\nu(T) \subseteq I$. Then, in particular, both $\nu(\langle w, x, y, z' \rangle)$ and $\nu(\langle w', x, y', z \rangle)$ are in I. Because $I \models \bowtie[AB, BCD]$, it follows that $\nu(\langle w, x, y', z \rangle) \in I$. Thus $\nu(T') \subseteq I$ and $\nu(t) \in (T', t)(I)$. The transformation from (T, t) to (T', t) is termed "applying the jd $\bowtie[AB, BCD]$," because T' is the result of adding a member of $\pi_{AB}(T) \bowtie$

	A	B	C	D
T	w	x	y	z'
	w'	x	y'	z
t	w	x	y	z

(a) The tableau query (T, t)

	A	B	C	D
T'	w	x	y	z'
	w'	x	y'	z
	w	x	y'	z
t	w	x	y	z

(b) (One) result of applying the jd $\bowtie[AB, BCD]$

	A	B	C	D
T''	w	x	y	z'
	w'	x	y	z
	w	x	y	z
t	w	x	y	z

(c) Result of applying the fd $A \to C$

Figure 8.3: Illustration of the chase

$\pi_{BCD}(T)$ to T. We shall see that, by repeated applications of a jd, one can eventually "force" the tableau to satisfy the jd.

The tableau T'' of Fig. 8.3(c) is the result of chasing (T', t) with the fd $A \to C$ (i.e., replacing all occurrences of y' by y). We now argue that $(T', t)(I) = (T'', t)(I)$. First, by Theorem 6.2.3, $(T', t)(I) \supseteq (T'', t)(I)$ because there is a homomorphism from (T', t) to (T'', t). For the opposite inclusion, suppose now that $\nu(T') \subseteq I$. This implies that ν embeds the first tuple of T'' into I. In addition, because $\nu(\langle w, x, y, z'\rangle)$ and $\nu(\langle w, x, y', z\rangle)$ are in I and $I \models A \to C$, it follows that $\nu(y) = \nu(y')$. Thus $\nu(\langle w', x, y, z\rangle) = \nu(\langle w', x, y', z\rangle) \in I$, and $\nu(\langle w, x, y, z\rangle) = \nu(\langle w, x, y', z\rangle) \in I$, [i.e., ν embeds the second and third tuples of T'' into I, such that $\nu(T'') \subseteq I$]. Note that (T'', t) is the result of identifying a pair of variables that caused a violation of $A \to C$ in T'. We will see that by repeated applications of an fd, one can eventually "force" a tableau to satisfy the fd. Note that in this case, chasing with respect to $A \to C$ has no effect before chasing with respect to $\bowtie[AB, BCD]$.

Finally, note that by the Homomorphism Theorem 6.2.3 of Chapter 6, $(T'', t) \equiv (\{t\}, t)$. It follows, then, that for all instances I that satisfy $\{A \to C, \bowtie[AB, BCD]\}$, (T, t) and $(\{t\}, t)$ yield the same answer.

Defining the Chase

As seen in Example 8.4.1, the chase relates to equivalence of queries over a family of instances satisfying certain dependencies. For a family $\mathcal{F}$ of instances over $\mathbf{R}$, we say that q_1 is *contained* in q_2 *relative to* $\mathcal{F}$, denoted $q_1 \subseteq_{\mathcal{F}} q_2$, if $q_1(\mathbf{I}) \subseteq q_2(\mathbf{I})$ for each instance $\mathbf{I}$ in $\mathcal{F}$. We are particularly interested in families $\mathcal{F}$ that are defined by a set Σ of dependencies (in the current context, fd's and jd's). Let Σ be a set of (functional and join) dependencies over $\mathbf{R}$. The *satisfaction family* of Σ, denoted $sat(\mathbf{R}, \Sigma)$ or simply $sat(\Sigma)$ if $\mathbf{R}$ is understood from the context, is the family

$$sat(\Sigma) = \{\mathbf{I} \text{ over } \mathbf{R} \mid \mathbf{I} \models \Sigma\}.$$

Query q_1 is contained in q_2 *relative to* Σ, denoted $q_1 \subseteq_\Sigma q_2$, if $q_1 \subseteq_{sat(\Sigma)} q_2$. Equivalence relative to a family of instances ($\equiv_{\mathcal{F}}$) and to a set of dependencies ($\equiv_\Sigma$) are defined similarly.

The chase is a general technique that can be used, given a set of dependencies Σ, to transform a tableau query q into a query q' such that $q \equiv_\Sigma q'$. The chase is defined as a nondeterministic procedure based on the successive application of individual dependencies from Σ, but as will be seen this process is "Church-Rosser" in the sense that the procedure necessarily terminates with a unique end result. As a final step in this development, the chase will be used to characterize equivalence of conjunctive queries with respect to a set Σ of dependencies ($\equiv_\Sigma$).

In the following, we let R be a fixed relation schema, and we focus on sets Σ of fd's and jd's over R and tableau queries with no constants over R. The entire development can be generalized to database schemas and conjunctive queries with constants (Exercise 8.27) and to a considerably larger class of dependencies (Chapter 10).

For technical convenience, we assume that there is a total order $\leq$ on the set **var**. Let R be a fixed relation schema and suppose that (T, t) is a tableau query over R. The chase is based on the successive application of the following two rules:

fd rule: Let $\sigma = X \to A$ be an fd over R, and let $u, v \in T$ be such that $\pi_X(u) = \pi_X(v)$ and $u(A) \neq v(A)$. Let x be the lesser variable in $\{u(A), v(A)\}$ under the ordering $\leq$, and let y be the other one (i.e., $\{x, y\} = \{u(A), v(A)\}$ and $x < y$). The *result of applying* the fd σ to u, v in (T, t) is the tableau query $(\theta(T), \theta(t))$, where θ is the substitution that maps y to x and is the identity elsewhere.

jd rule: Let $\sigma = \bowtie[X_1, \ldots, X_n]$ be a jd over R, let u be a free tuple over R not in T, and suppose that $u_1, \ldots, u_n \in T$ satisfy $\pi_{X_i}(u_i) = \pi_{X_i}(u)$ for $i \in [1, n]$. Then the *result of applying* the jd σ to $(u_1, \ldots, u_n)$ in (T, t) is the tableau query $(T \cup \{u\}, t)$.

Following the lead of Example 8.4.1, the following is easily verified (see Exercise 8.24a).

Proposition 8.4.2 Suppose that Σ is a set of fd's and jd's over R, $\sigma \in \Sigma$, and q is a tableau query over R. If q' is the result of applying σ to some tuples in q, then $q' \equiv_\Sigma q$.

A *chasing sequence* of (T, t) by Σ is a (possibly infinite) sequence

$$(T, t) = (T_0, t_0), \ldots, (T_i, t_i), \ldots$$

such that for each $i \geq 0$, (T_{i+1}, t_{i+1}) (if defined) is the result of applying some dependency in Σ to (T_i, t_i). The sequence is *terminal* if it is finite and no dependency in Σ can be applied to it. The last element of the terminal sequence is called its *result*. The notion of *satisfaction* of a dependency is extended naturally to tableaux. The following is an important property of terminal chasing sequences (Exercise 8.24b).

Lemma 8.4.3 Let (T', t') be the result of a terminal chasing sequence of (T, t) by Σ. Then T', considered as an instance, satisfies Σ.

Because the chasing rules do not introduce new variables, it turns out that the chase procedure always terminates. The following is easily verified (Exercise 8.24c):

LEMMA 8.4.4 Let (T, t) be a tableau query over R and Σ a set of fd's and jd's over R. Then each chasing sequence of (T, t) by Σ is finite and is the initial subsequence of a terminal chasing sequence.

An important question now is whether the results of different terminal chasing sequences are the same. This turns out to be the case. This property of chasing sequences is called the *Church-Rosser property*. We provide the proof of the Church-Rosser property for the chase at the end of this section (Theorem 8.4.18).

Because the Church-Rosser property holds, we can define without ambiguity the result of chasing a tableau query by a set of fd's and jd's.

DEFINITION 8.4.5 If (T, t) is a tableau query over R and Σ a set of fd's and jd's over R, then the *chase* of (T, t) by Σ, denoted $chase(T, t, \Sigma)$, is the result of some (any) terminal chasing sequence of (T, t) by Σ.

From the previous discussion, $chase(T, t, \Sigma)$ can be computed as follows. The dependencies are picked in some arbitrary order and arbitrarily applied to the tableau. Applying an fd to a tableau query q can be performed within time polynomial in the size of q. However, determining whether a jd can be applied to q is NP-complete in the size of q. Thus the best-known algorithm for computing the chase is exponential (see Exercise 8.25). However, the complexity is polynomial if the schema is considered fixed.

Until now, besides the informal discussion in Section 8.1, the *chase* remains a purely syntactic technique. We next state a result that shows that the chase is in fact determined by the semantics of the dependencies in Σ and not just their syntax.

In the following proposition, recall that by definition, $\Sigma \equiv \Sigma'$ if $\Sigma \models \Sigma'$ and $\Sigma' \models \Sigma$. The proof, which we omit, uses the Church-Rosser property of the chase (see also Exercise 8.26).

PROPOSITION 8.4.6 Let Σ and Σ' be sets of fd's and jd's over R, and let (T, t) be a tableau query over R. If $\Sigma \equiv \Sigma'$, then $chase(T, t, \Sigma)$ and $chase(T, t, \Sigma')$ coincide.

We next consider several important uses of the chase that illustrate the power of this technique.

Query Equivalence

We consider first the problem of checking the equivalence of tableau queries in the presence of a set of fd's and jd's. This allows, for example, checking whether a tableau query can be replaced by a simpler tableau query when the dependencies are satisfied. Suppose now that (T', t') and (T'', t'') are two tableau queries and Σ a set of fd's and jd's such that $(T', t') \equiv_\Sigma (T'', t'')$. From the preceding development (Proposition 8.4.2), it follows that

$$chase(T', t', \Sigma) \equiv_\Sigma (T', t') \equiv_\Sigma (T'', t'') \equiv_\Sigma chase(T'', t'', \Sigma).$$

We now show that, in fact, $chase(T', t', \Sigma) \equiv chase(T'', t'', \Sigma)$. Furthermore, this condition is sufficient as well as necessary.

To demonstrate this result, we first establish the following more general fact.

THEOREM 8.4.7 Let $\mathcal{F}$ be a family of instances over relation schema R that is closed under isomorphism, and let (T_1, t_1), (T_2, t_2), (T_1', t_1'), and (T_2', t_2') be tableau queries over R. Suppose further that for $i = 1, 2$,

(a) $(T_i', t_i') \equiv_\mathcal{F} (T_i, t_i)$ and

(b) T_i', considered as an instance, is in $\mathcal{F}$.[3]

Then $(T_1, t_1) \subseteq_\mathcal{F} (T_2, t_2)$ iff $(T_1', t_1') \subseteq (T_2', t_2')$.

Proof The if direction is immediate. For the only-if direction, suppose that $(T_1, t_1) \subseteq_\mathcal{F} (T_2, t_2)$. It suffices by the Homomorphism Theorem 6.2.3 to exhibit a homomorphism that embeds (T_2', t_2') into (T_1', t_1'). Because T_1', considered as an instance, is in $\mathcal{F}$,

$$t_1' \in (T_1', t_1')(T_1') \Rightarrow t_1' \in (T_1, t_1)(T_1') \Rightarrow t_1' \in (T_2, t_2)(T_1') \Rightarrow t_1' \in (T_2', t_2')(T_1').$$

It follows that there is a homomorphism h such that $h(T_2') \subseteq T_1'$ and $h(t_2') = t_1'$. Thus $(T_1', t_1') \subseteq (T_2', t_2')$. This completes the proof. ■

Together with Lemma 8.4.3, this implies the following:

THEOREM 8.4.8 Let (T_1, t_1) and (T_2, t_2) be tableau queries over R and Σ a set of fd's and jd's over R. Then

1. $(T_1, t_1) \subseteq_\Sigma (T_2, t_2)$ iff $chase(T_1, t_1, \Sigma) \subseteq chase(T_2, t_2, \Sigma)$.
2. $(T_1, t_1) \equiv_\Sigma (T_2, t_2)$ iff $chase(T_1, t_1, \Sigma) \equiv chase(T_2, t_2, \Sigma)$.

Query Optimization

As suggested in Example 8.4.1, the chase can be used to optimize tableau queries in the presence of dependencies such as fd's and jd's. Given a tableau query (T, t) and a set Σ of fd's and jd's, $chase(T, t, \Sigma)$ is equivalent to (T, t) on all instances satisfying Σ. A priori, it is not clear that the new tableau is an improvement over the first. It turns out that the chase using fd's can never yield a more complicated tableau and, as shown in Example 8.4.1, can yield a much simpler one. On the other hand, the chase using jd's may yield a more complicated tableau, although it may also produce a simpler one.

We start by looking at the effect on tableau minimization of the chase using fd's. In the following, we denote by $min(T, t)$ the tableau resulting from the minimization of

[3] More precisely, T' considered as an instance is in $\mathcal{F}$ means that some instance isomorphic to T' is in $\mathcal{F}$.

the tableau (T, t) using the Homomorphism Theorem 6.2.3 for tableau queries, and by $|min(T, t)|$ we mean the cardinality of the tableau of $min(T, t)$.

Lemma 8.4.9 Let (T, t) be a tableau query and Σ a set of fd's. Then $|min(chase(T, t, \Sigma))| \leq |min(T, t)|$.

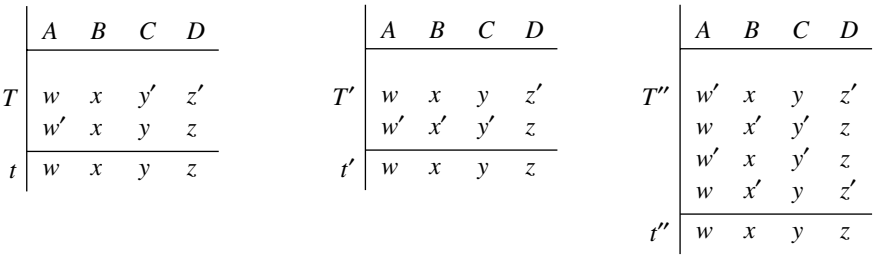
Crux By the Church-Rosser property of the chase, the order of the dependencies used in a chase sequence is irrelevant. Clearly it is sufficient to show that for each tableau query (T', t') and $\sigma \in \Sigma$, $|min(chase(T', t', \sigma))| \leq |min(T', t')|$. We can assume without loss of generality that σ is of the form $X \to A$, where A is a single attribute.

Let $(T'', t'') = chase(T', t', \{X \to A\})$, and let θ be the *chase homomorphism* of a chasing sequence for $chase(T', t', \{X \to A\})$, i.e., the homomorphism obtained by composing the substitutions used in that chasing sequence (see the proof of Theorem 8.4.18). We will use here the Church-Rosser property of the chase (Theorem 8.4.18) as well as a related property stating that the homomorphism θ, like the result, is also the same for all chase sequences (this follows from the proof of Theorem 8.4.18).

By Theorem 6.2.6, there is some $S \subseteq T'$ such that (S, t') is a minimal tableau query equivalent to (T', t'); we shall use this as the representative of $min(T', t')$. Let h be a homomorphism such that $h(T', t') = (S, t')$. Consider the mapping f on (T'', t'') defined by $f(\theta(x)) = \theta(h(x))$, where x is a variable in (T', t'). If we show that f is well defined, we are done. [If f is well defined, then f is a homomorphism from (T'', t'') to $\theta(S, t') = (\theta(S), t'')$, and so $(T'', t'') \supseteq \theta(S, t')$. On the other hand, the $\theta(S) \subseteq \theta(T') = T''$, and so $(T'', t'') \subseteq \theta(S, t')$. Thus, $(T'', t'') \equiv \theta(S, t') = \theta(min(T', t'))$, and so $|min(T'', t'')| = |min(\theta(min(T', t')))| \leq |\theta(min(T', t'))| \leq |min(T', t')|$.]

To see that f is well defined, suppose $\theta(x) = \theta(y)$. We have to show that $\theta(h(x)) = \theta(h(y))$. Consider a terminal chasing sequence of (T', t') using $X \to A$, and $(u_1, v_1), \ldots, (u_n, v_n)$ as the sequence of pairs of tuples used in the sequence, yielding the chase homomorphism θ. Consider the sequence $(h(u_1), h(v_1)), \ldots, (h(u_n), h(v_n))$. Clearly if $X \to A$ can be applied to (u, v), then it can be applied to $(h(u), h(v))$, unless $h(u(A)) = h(v(A))$. Let $(h(u_{i_1}), h(v_{i_1})), \ldots, (h(u_{i_k}), h(v_{i_k}))$ be the subsequence of these pairs for which $X \to A$ can be applied. It can be easily verified that there is a chasing sequence of $(h(T'), t')$ using $X \to A$ that uses the pairs $(h(u_{i_1}), h(v_{i_1})), \ldots, (h(u_{i_k}), h(v_{i_k}))$, with chase homomorphism θ'. Note that for all x', y', if $\theta(x') = \theta(y')$ then $\theta'(h(x')) = \theta'(h(y'))$. In particular, $\theta'(h(x)) = \theta'(h(y))$. Because $h(T') \subseteq T'$, θ' is the chase homomorphism of a chasing sequence $\sigma_1, \ldots, \sigma_k$ of (T', t'). Let θ'' be the chase homomorphism formed from a terminal chasing sequence that extends $\sigma_1, \ldots, \sigma_k$. Then $\theta''(h(x)) = \theta''(h(y))$. Finally, by the uniqueness of the chase homomorphism, $\theta'' = \theta$, and so $\theta(h(x)) = \theta(h(y))$ as desired. This concludes the proof. ■

It turns out that jd's behave differently than fd's with respect to minimization of tableaux. The following shows that the chase using jd's may yield simpler but also more complicated tableaux.



	A	B	C	D
T	w	x	y′	z′
	w′	x	y	z
t	w	x	y	z

(a) The tableau query (T, t)

	A	B	C	D
T′	w	x	y	z′
	w′	x′	y′	z
t′	w	x	y	z

(b) The tableau query (T', t')

	A	B	C	D
T″	w′	x	y	z′
	w	x′	y′	z
	w′	x	y′	z
	w	x′	y	z′
t″	w	x	y	z

(c) The tableau query $chase(T', t', \{\bowtie[AB, CD]\})$

Figure 8.4: Minimization and the chase using jd's

EXAMPLE 8.4.10 Consider the tableau query (T, t) shown in Fig. 8.4(a) and the jd $\sigma = \bowtie[AB, BCD]$. Clearly (T, t) is minimal, so $|min(T, t)| = 2$. Next consider $chase(T, t, \sigma)$. It is easy to check that $\langle w, x, y, z\rangle \in chase(T, t, \sigma)$, so $chase(T, t, \sigma)$ is equivalent to the identity and

$$|min(chase(T, t, \sigma))| = 1.$$

Next let (T', t') be the tableau query in Fig. 8.4(b) and $\sigma = \bowtie[AB, CD]$. Again (T', t') is minimal. Now $chase(T', t', \sigma)$ is represented in Fig. 8.4(c) and is minimal. Thus

$$|min(chase(T', t', \sigma))| = 4 > |min(T', t')|.$$

Despite the limitations illustrated by the preceding example, the chase in conjunction with tableau minimization provides a powerful optimization technique that yields good results in many cases. This is illustrated by the following example and by Exercise 8.28.

EXAMPLE 8.4.11 Consider the SPJ expression

$$q = \pi_{AB}(\pi_{BCD}(R) \bowtie \pi_{ACD}(R)) \bowtie \pi_{AD}(R),$$

where R is a relation with attributes $ABCD$. Suppose we wish to optimize the query on databases satisfying the dependencies

$$\Sigma = \{B \to D, D \to C, \bowtie[AB, ACD]\}.$$

The tableau (T, t) corresponding to q is represented in Fig. 8.5(a). Note that (T, t) is minimal. Next we chase (T, t) using the dependencies in Σ. The chase using the fd's in Σ does not change (T, t), which already satisfies them. The chase using the jd

	A	B	C	D
T	w'	x	y'	z'
	w	x'	y'	z'
	w	x''	y''	z
t	w	x	y	z

(a) The tableau query (T, t) corresponding to q

	A	B	C	D
T''	w'	x	y'	z'
	w	x'	y'	z
	w	x''	y'	z
t''	w	x	y	z

(c) The tableau query $(T'', t'') = chase(T', t', \{B \to D, D \to C\})$

	A	B	C	D
T'	w'	x	y'	z'
	w	x'	y'	z'
	w	x''	y''	z
	w	x'	y''	z
	w	x''	y'	z'
t'	w	x	y	z

(b) The tableau query $(T', t') = chase(T, t, \{\bowtie[AB, ACD]\})$

	A	B	C	D
T'''	w'	x	y'	z
	w	x'	y'	z
t'''	w	x	y	z

(d) The tableau query $(T''', t''') = min(T'', t'')$

Figure 8.5: Optimization of SPJ expressions by tableau minimization and the chase

$\bowtie[AB, ACD]$ yields the tableau (T', t') in Fig. 8.5(b). Now the fd's can be applied to (T', t') yielding the tableau (T'', t'') in Fig. 8.5(c). Finally (T'', t'') is minimized to (T''', t''') in Fig. 8.5(d). Note that (T''', t''') satisfies Σ, so the chase can no longer be applied. The SPJ expression corresponding to (T''', t''') is $\pi_{ABD}(\pi_{BCD}(R) \bowtie \pi_{ACD}(R))$. Thus, the optimization of q resulted in saving one join operation. Note that the new query is not simply a subexpression of the original. In general, the shape of queries can be changed radically by the foregoing procedure.

The Chase and Logical Implication

We consider a natural correspondence between dependency satisfaction and conjunctive query containment. This correspondence uses tableaux to represent dependencies. We will see that the chase provides an alternative point of view to dependency implication.

First consider a jd $\sigma = \bowtie[X_1, \ldots, X_n]$. It is immediate to see that an instance I satisfies σ iff $q_\sigma(I) \subseteq q_{id}(I)$, where

$$q_\sigma = [X_1] \bowtie \cdots \bowtie [X_n]$$

and q_{id} is the identity query. Both q_σ and q_{id} are PSJR expressions. We can look at alternative formalisms for expressing q_σ and q_{id}. For instance, the *tableau query of* σ is (T_σ, t), where for some $t_1, \ldots, t_n$,

- t is a free tuple over R with a distinct variable for each coordinate,
- $T_\sigma = \{t_1, \ldots, t_n\}$,
- $\pi_{X_i}(t_i) = \pi_{X_i}(t)$ for $i \in [1, n]$, and
- the other coordinates of the t_i's hold distinct variables.

It is again easy to see that $q_\sigma = (T_\sigma, t)$, so $I \models \sigma$ iff $(T_\sigma, t)(I) \subseteq (\{t\}, t)(I)$.

For fd's, the situation is only slightly more complicated. Consider an fd $\sigma' = X \to A$ over U. It is easy to see that $I \models \sigma'$ iff $(T_{\sigma'}, t_{\sigma'})(I) \subseteq (T_{\sigma'}, t'_{\sigma'})(I)$, where

	X	A	$(U - AX)$
$T_{\sigma'}$	u	x	v_1
	u	x'	v_2
$t_{\sigma'}$	x	x'	

	X	A	$(U - AX)$
	u	x	v_1
	u	x'	v_2
$t'_{\sigma'}$	x	x	

where u, v_1, v_2 are vectors of distinct variables and x, x' are distinct variables occurring in none of these vectors. The *tableau query* of σ' is $(T_{\sigma'}, t_{\sigma'})$.

Again observe that $(T_{\sigma'}, t_{\sigma'})$, (T_σ, t_σ) can be expressed as PSJR expressions, so fd satisfaction also reduces to containment of PSJR expressions. It will thus be natural to look more generally at all dependencies expressed as containment of PSJR expressions. In Chapter 10, we will consider the general class of *algebraic dependencies* based on containment of these expressions.

Returning to the chase, we next use the tableau representation of dependencies to obtain a characterization of logical implication (Exercise 8.29). This result is generalized by Corollary 10.2.3.

Theorem 8.4.12 Let Σ and $\{\sigma\}$ be sets of fd's and jd's over relation schema R, let (T_σ, t_σ) be the tableau query of σ, and let T be the tableau in $chase(T_\sigma, t_\sigma, \Sigma)$. Then $\Sigma \models \sigma$ iff

(a) $\sigma = X \to A$ and $|\pi_A(T)| = 1$, that is, the projection over A of T is a singleton; or

(b) $\sigma = \bowtie[X_1, \ldots, X_n]$ and $t_\sigma \in T$.

This implies that determining logical implication for jd's alone, and for fd's and jd's taken together, is decidable. On the other hand, tableau techniques are also used to obtain the following complexity results for logical implication of jd's (see Exercise 8.30).

THEOREM 8.4.13

(a) Testing whether a jd and an fd imply a jd is NP-complete.

(b) Testing whether a set of mvd's implies a jd is NP-hard.

Acyclic Join Dependencies

In Section 6.4, a special family of joins called acyclic was introduced and was shown to enjoy a number of desirable properties. We show now a connection between those results, join dependencies, and multivalued dependencies.

A jd $\bowtie[X_1, \ldots, X_n]$ is *acyclic* if the hypergraph corresponding to $[X_1, \ldots, X_n]$ is acyclic (as defined in Section 6.4).

Using the chase, we show here that a jd is acyclic iff it is equivalent to a set of mvd's. The discussion relies on the notation and techniques developed in the discussion of acyclic joins in Section 6.4.

We shall use the following lemma.

LEMMA 8.4.14 Let $\sigma = \bowtie\mathbf{X}$ be a jd over U, and let $X, Y \subseteq U$ be disjoint sets. Then the following are equivalent:

(i) $\sigma \models X \twoheadrightarrow Y$;

(ii) there is no $X_i \in \mathbf{X}$ such that $X_i \cap Y \neq \emptyset$ and $X_i \cap (U - XY) \neq \emptyset$;

(iii) Y is a union of connected components of the hypergraph $\mathbf{X}|_{U-X}$.

Proof Let $Z = U - XY$. Let τ denote the mvd $X \twoheadrightarrow Y$, and let (T_τ, t_τ) be the tableau query corresponding to τ. Let $T_\tau = \{t_Y, t_Z\}$ where $t_Y[XY] = t_\tau[XY]$ and $t_Z[XZ] = t_\tau[XZ]$ and distinct variables are used elsewhere in t_Y and t_Z.

We show now that (i) implies (ii). By Theorem 8.4.12, $t_\tau \in T = chase(T_\tau, t_\tau, \sigma)$. Let $X_i \in \mathbf{X}$. Suppose that t is a new tuple created by an application of σ during the computation of T. Then $t[X_i]$ agrees with $t'[X_i]$ for some already existing tuple. An induction implies that $t_\tau[X_i] = t_Y[X_i]$ or $t_\tau[X_i] = t_Z[X_i]$. Because t_Y and t_Z agree only on X, this implies that X_i cannot intersect with both Y and Z.

That (ii) implies (iii) is immediate. To see that (iii) implies (i), consider an application of the jd $\bowtie\mathbf{X}$ on T_τ, where $X_i \in \mathbf{X}$ is associated with t_Y if $X_i - X \subseteq Y$, and X_i is associated with t_Z otherwise. This builds the tuple t_τ, and so by Theorem 8.4.12, $\sigma \models X \twoheadrightarrow Y$. ■

We now have the following:

THEOREM 8.4.15 A jd σ is acyclic iff there is a set Σ of mvd's that is equivalent to σ.

Proof (**only if**) Suppose that $\sigma = \bowtie\mathbf{X}$ over U is acyclic. By Theorem 6.4.5, this implies that the output of the GYO algorithm on $\mathbf{X}$ is empty. Let $X_1, \ldots, X_n$ be an enumeration of $\mathbf{X}$ in the order of an execution of the GYO algorithm. In particular, X_i is an ear of the hypergraph formed by $\{X_{i+1}, \ldots, X_n\}$.

For each $i \in [1, n-1]$, let $P_i = \cup_{j\in[1,i]} X_j$ and $Q_i = \cup_{j\in[i+1,n]} X_j$. Let $\Sigma = \{[P_i \cap Q_i] \twoheadrightarrow Q_i \mid i \in [1, n-1]\}$. By Lemma 8.4.14 and the choice of sequence $X_1, \ldots, X_n$, $\sigma \models \Sigma$. To show that $\Sigma \models \sigma$, we construct a chasing sequence of (T_σ, t_σ) using Σ that yields t_σ. This chase shall inductively produce a sequence $t_1, \ldots, t_n$ of tuples, such that $t_i[P_i] = t_\sigma[P_i]$ for $i \in [1, n]$.

We begin by setting t_1 to be the tuple of T_σ that corresponds to X_1. Then $t_1[P_1] = t_\sigma[P_1]$ because $P_1 = X_1$. More generally, given t_i with $i \geq 1$, the mvd $[P_i \cap Q_i] \twoheadrightarrow Q_i$ on t_i and the tuple corresponding to X_{i+1} can be used to construct tuple t_{i+1} with the desired property. The final tuple t_n constructed by this process is t_σ, and so $\Sigma \models \sigma$ as desired.

(**if**) Suppose that $\sigma = \bowtie\mathbf{X}$ over U is equivalent to the set Σ of mvd's but that σ is not acyclic. From the definition of acyclic, this implies that there is some $W \subseteq U$ such that $\mathbf{Y} = \mathbf{X}|_W$ has no articulation pair. Without loss of generality we assume that $\mathbf{Y}$ is connected.

Let $\mathbf{Y} = \{Y_1, \ldots, Y_m\}$. Suppose that $s_1, \ldots$ are the tuples produced by some chasing sequence of (T_σ, t_σ). We argue by induction that for each $k \geq 1$, $s_k[W] \in \pi_W(T_\sigma)$. Suppose otherwise, and let s_k be the first where this does not hold. Suppose that s_k is the result of applying an mvd $X \twoheadrightarrow Y$ in Σ. Without loss of generality we assume that $X \cap Y = \emptyset$. Let $Z = U - XY$. Because s_k results from $X \twoheadrightarrow Y$, there are two tuples s' and s'' either in T_σ or already produced, such that $s_k[XY] = s'[XY]$ and $s_k[XZ] = s''[XZ]$. Because s_k is chosen to be least, there are tuples t_i and t_j in T_σ, which correspond to X_i and X_j, respectively, such that $s'[W] = t_i[W]$ and $s''[W] = t_j[W]$.

Because t_i and t_j correspond to X_i and X_j, for each attribute $A \in U$ we have $t_i[A] = t_j[A]$ iff $A \in X_i \cap X_j$. Thus $X \cap W \subseteq X_i \cap X_j$.

Because $s_k[W] \neq t_i[W]$, $W - XZ \neq \emptyset$, and because $s_k[W] \neq t_j[W]$, $W - XY \neq \emptyset$. Now, by Lemma 8.4.14, because $X \twoheadrightarrow Y$ is implied by σ, there is no $X_k \in \mathbf{X}$ such that $X_k \cap Y \neq \emptyset$ and $X_k \cap Z \neq \emptyset$. It follows that $\mathbf{Y}|_{W-X}$ is disconnected. Finally, let $Y = X_i \cap W$ and $Y' = X_j \cap W$. Because $X \cap W \subseteq X_i \cap X_j$, it follows that $Y \cap Y'$ is an articulation set for $\mathbf{Y}$, a contradiction. ■

We conclude with a complexity result about acyclic jd's. The first part follows from the proof of the preceding theorem and the fact that the GYO algorithm runs in polynomial time. The second part, stated without proof, is an interesting converse of the first part.

PROPOSITION 8.4.16

(a) There is a PTIME algorithm that, given an acyclic jd σ, produces a set of mvd's equivalent to σ.

(b) There is a PTIME algorithm that, given a set Σ of mvd's, finds a jd equivalent to Σ or determines that there is none.

The Chase Is Church-Rosser

To conclude this section, we provide the proof that the results of all terminal chasing sequences of a tableau query q by a set Σ of fd's and jd's are identical. To this end, we first introduce tools to describe correspondences between the free tuples occurring in the different elements of chasing sequences.

Let $(T, t) = (T_0, t_0), \ldots, (T_n, t_n)$ be a chasing sequence of (T, t) by Σ. Then for each $i \in [1, n]$, the *chase homomorphism* for step i, denoted θ_i, is an assignment with domain $var(T_i)$ defined as follows:

(a) If (T_{i+1}, t_{i+1}) is the result of applying the fd rule to (T_i, t_i), which replaces all occurrences of variable y by variable x, then θ_{i+1} is defined so that $\theta_{i+1}(y) = x$ and θ_{i+1} is the identity on $var(T_i) - \{y\}$.

(b) If (T_{i+1}, t_{i+1}) is the result of applying the jd rule to (T_i, t_i), then θ_{i+1} is the identity on $var(T_i)$.

The *chase homomorphism* of this chasing sequence is $\theta = \theta_1 \circ \cdots \circ \theta_n$. If $w \in (T \cup \{t\})$, then the tuple *corresponding* to w in (T_i, t_i) is $w_i = \theta_1 \circ \cdots \circ \theta_i(w)$. It may arise that $u_i = v_i$ for distinct tuples u, v in T. Observe that $\theta_1 \circ \cdots \circ \theta_i(T) \subseteq T_i$ and that, because of the jd rule, the inclusion may be strict.

We now have the following:

LEMMA 8.4.17 Suppose that $I \models \Sigma$, ν is a substitution over $var(T)$, $\nu(T) \subseteq I$, and $(T_0, t_0), \ldots, (T_n, t_n)$ is a chasing sequence of (T, t) by Σ. Then

$$\nu(w_i) = \nu(w) \text{ for each } i \in [1, n] \text{ and each } w \in (T \cup \{t\}),$$

and $\nu(T_i) \subseteq I$ for each $i \in [1, n]$.

Crux Use an induction on the chasing sequence (Exercise 8.24d). ■

Observe that this also holds if I is a tableau over R that satisfies Σ. This is used in the following result.

THEOREM 8.4.18 Let (T, t) be a tableau query over R and Σ a set of fd's and jd's over R. Then the results of all terminal chasing sequences of (T, t) by Σ are identical.

Proof Let (T', t') and (T'', t'') be the results of two terminal chasing sequences on (T, t) using Σ, and let θ', θ'' be the chase homomorphisms of these chasing sequences. For each tuple $w \in T$, let w' denote the tuple of T' that corresponds to w, and similarly for w'', T''.

By construction, $\theta''(T) \subseteq T''$ and $\theta''(t) = t''$. Because $T'' \models \Sigma$ and $\theta''(T) \subseteq T''$, $\theta''(T') \subseteq T''$ by Lemma 8.4.17 considering the chasing sequence leading to T'. The same argument shows that $\theta''(w') = w''$ for each w in T and $\theta''(t') = t''$. By symmetry, $\theta'(T'') \subseteq T'$, $\theta'(w'') = w'$ for each w in T and $\theta'(t'') = t'$.

We next prove that

(*) θ'' is an isomorphism from (T', t') to (T'', t'').

Let w'' be in T'' for some w in T. Then

$$\theta' \circ \theta''(w'') = \theta''(\theta'(w'')) = \theta''(w') = w''.$$

Observe that each variable x in $var(T'')$ occurs in w'', for some w in T. Thus $\theta' \circ \theta''$ is the identity over $var(T'')$. We therefore have

$$\theta' \circ \theta''(T'') = T''.$$

By symmetry, $\theta'' \circ \theta'$ is the identity over $var(T')$ and

$$\theta'' \circ \theta'(T') = T'.$$

Thus $|T''| = |T'|$. Because $\theta''(T') \subseteq T''$, $\theta''(T') = T''$ and θ'' is an isomorphism from (T', t') to (T'', t''), so (*) holds.

To conclude, we prove that

(**) θ'' is the identity over $var(T')$.

We first show that for each pair x, y of variables occurring in T,

(†) $\theta''(x) = \theta''(y)$ iff $\theta'(x) = \theta'(y)$.

Suppose that $\theta''(x) = \theta''(y)$. Then for some tuples $u, v \in T$ and attributes A, B, we have $u(A) = x$, $v(B) = y$ and $u''(A) = \theta''(x) = \theta''(y) = v''(B)$. Next $\theta'(x) = u'(A)$ and $\theta'(y) = v'(B)$. Because θ' is an isomorphism from (T'', t'') to (T', t') and $\theta'(u'') = u', \theta'(v'') = v'$, it follows that $u'(A) = v'(B)$. Hence $\theta'(x) = u'(A) = v'(B) = \theta'(y)$ as desired. The if direction follows by symmetry.

Now let $x \in var(T')$. To prove (**) and the theorem, it now suffices to show that $\theta''(x) = x$. Let

$$\mathcal{A}' = \{y \in var(T) \mid \theta'(y) = \theta'(x)\},$$
$$\mathcal{A}'' = \{y \in var(T) \mid \theta''(y) = \theta''(x)\}.$$

First (†) implies that $\mathcal{A}' = \mathcal{A}''$. Furthermore, an induction on the chasing sequence for (T', t') shows that for each $z \in \mathcal{A}'$, $\theta'(z)$ is the least (under the ordering on **var**) element of $\mathcal{A}'$, and similarly for (T'', t''). Thus θ' and θ'' map all elements of $\mathcal{A}'$ and $\mathcal{A}''$ to the same variable z. Because $x \in var(T')$, it follows that $z = x$ so, in particular, $\theta'(x) = \theta''(x) = x$. ■

Bibliographic Notes

On a general note, we first mention that comprehensive presentations of dependency theory can be found in [Var87, FV86]. A more dense presentation is provided in [Kan91]. Dependency theory is also the topic of the book [Tha91].

Research on general integrity constraints considered from the perspective of first-order logic is presented in [GM78]. Other early work in this framework includes [Nic78], which observes that fd's and mvd's have a natural representation in logic, and [Nic82], which

considers incremental maintanence of integrity constraints under updates to the underlying state.

Functional dependencies were introduced by Codd [Cod72b]. The axiomatization is due to [Arm74]. The problem of implication is studied in [BB79, Mai80]. Several alternative formulations of fd implication, including formulation in terms of the propositional calculus perspective (see Exercise 8.22), are mentioned in [Kan91]; they are due to [SDPF81, CK85, CKS86].

Armstrong relations were introduced and studied in [Fag82b, Fag82a, BDFS84]. Interesting practical applications of Armstrong relations are proposed in [SM81, MR85]. The idea is that, given a set Σ of fd's, the system presents an Armstrong relation for Σ with natural column entries to a user, who can then determine whether Σ includes all of the desired restrictions.

The structure of families of instances specified by a set of fd's is studied in [GZ82, Hul84].

Multivalued dependencies were discovered independently in [Zan76, Fag77b, Del78]. They were generalized in [Ris77, Nic78, ABU79]. The axiomatization of fd's and mvd's is from [BFH77]. A probabilistic view of mvd's in terms of conditional independence is presented in [PV88, Pea88]. This provides an alternative motivation for the study of such dependencies.

The issue of whether there is an axiomatization for jd's has a lengthy history. As will be detailed in Chapter 10, the family of full typed dependencies subsumes the family of jd's, and an axiomatization for these was presented in [BV84a, YP82]; see also [SU82]. More focused axiomatizations, which start with jd's and end with jd's but use slightly more general dependencies at intermediate stages, are presented in [Sci82] and [BV85]; see also [BV81b]. Reference [BV85] also develops an axiomatization for join dependencies based on Gentzen-style proofs (see, e.g., [Kle67]); proofs in this framework maintain a form of scratch paper in addition to a sequence of inferred sentences. Finally, [Pet89] settled the issue by establishing that there is no axiomatization (in the sense defined in Section 8.2) for the family of jd's.

As noted in Chapter 6, acyclic joins received wide interest in the late 1970s and early 1980s so Theorem 8.4.15 was demonstrated in [FMU82]. Proposition 8.4.16 is from [GT83].

An ancestor to the chase can be found in [ABU79]. The notion of chase was articulated in [MMS79]. Related results can be found in [MSY81, Var83]. The relationship between the chase and both tableau queries and logical implication was originally presented in [MMS79] and builds on ideas originally introduced in [ASU79b, ASU79a]. The chase technique is extended to more general dependencies in [BV84c]; see also Chapter 10. The connection between the chase and the more general theorem-proving technique of resolution with paramodulation (see [CL73]) is observed and analyzed in [BV80b]. The chase technique is applied to datalog programs in [RSUV89, RSUV93].

Exercises

Exercise 8.1 Describe the set of fd's, mvd's, and jd's that are tautologies (i.e., dependencies that are satisfied by all instances) for a relation schema R.

Exercise 8.2 Let Σ_1 be as in Example 8.2.4. Prove that $\Sigma_1 \models AD \to E$ and $\Sigma_1 \models CDE \to C$.

Exercise 8.3 Let U be a set of attributes, and let Σ, Γ be sets of dependencies over U. Show that

(a) $\Sigma \subseteq \Sigma^*$.

(b) $(\Sigma^*)^* = \Sigma^*$.

(c) If $\Gamma \subseteq \Sigma$, then $\Gamma^* \subseteq \Sigma^*$.

State and prove analogous results for fd closures of attribute sets.

Exercise 8.4 Prove Lemma 8.2.6.

Exercise 8.5 Let U be a set of attributes and Σ a set of fd's over U. Prove the soundness of FD1, FD2, FD3 and show that

$$\text{If } \Sigma \vdash X \to Y \text{ and } \Sigma \vdash X \to Z, \text{ then } \Sigma \vdash X \to YZ.$$

Exercise 8.6 Let Σ be a set of fd's over U.

(a) Suppose that $X \subseteq U$ and $U \subseteq V$. Show that $(X, \Sigma)^{*,U} = (X, \Sigma)^{*,V}$. *Hint:* Use the proof of Proposition 8.2.8.

(b) Suppose that $XY \subseteq U$, and $U \subseteq V$. Show that $\Sigma \models_U X \to Y$ iff $\Sigma \models_V X \to Y$.

♠ **Exercise 8.7** [BB79] Describe how to improve the efficiency of Algorithm 8.2.7 to linear time. *Hint:* For each unused fd $W \to Z$ in Σ, record the number attributes of W not yet in *closure*. To do this efficiently, maintain a list for each attribute A of those unused fd's of Σ for which A occurs in the left-hand side.

Exercise 8.8 Give a proof of $AB \to F$ from $\Sigma = \{AB \to C, A \to D, CD \to EF\}$ using {FD1, FD2, FD3}.

Exercise 8.9 Prove or disprove the soundness of the following rules:

FD4: (pseudo-transitivity) If $X \to Y$ and $YW \to Z$, then $XW \to Z$.

FD5: (union) If $X \to Y$ and $X \to Z$, then $X \to YZ$.

FD6: (decomposition) If $X \to YZ$, then $X \to Y$.

MVD4: (pseudo-transitivity) If $X \twoheadrightarrow Y$ and $YW \twoheadrightarrow Z$, then $XW \twoheadrightarrow Z - Y$.

MVD5: (union) If $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow YZ$.

MVD6: (decomposition) If $X \twoheadrightarrow Y$ and $X \twoheadrightarrow Z$, then $X \twoheadrightarrow Y \cap Z$, $X \twoheadrightarrow Y - Z$, and $X \twoheadrightarrow Z - Y$.

bad-FD1: If $XW \to Y$ and $XY \to Z$, then $X \to (Z - W)$.

bad-MVD1: If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow Z$.

bad-FMVD1: If $X \twoheadrightarrow Y$ and $XY \to Z$, then $X \to Z$.

(The use of the hint is optional.)

Exercise 8.10 Continuing with Exercise 8.9,

(a) [BFH77] Find a two-element subset of {FD1, FD2, FD3, FD4, FD5, FD6} that is sound and complete for inferring logical implication of fd's.

(b) Prove that there is exactly one two-element subset of {FD1, FD2, FD3, FD4, FD5, FD6} that is sound and complete for inferring logical implication of fd's.

Exercise 8.11 [Arm74] Let U be a fixed set of attributes. An attribute set $X \subseteq U$ is *saturated* with respect to a set Σ of fd's over U if $X = X^*$. The family of saturated sets of Σ with respect to U is $satset(\Sigma) = \{X \subseteq U \mid X \text{ is saturated with respect to } \Sigma\}$.

(a) Show that $satset = satset(\Sigma)$ satisfies the following properties:

S1: $U \in satset$.
S2: If $Y \in satset$ and $Z \in satset$, then $Y \cap Z \in satset$.

★(b) Suppose that *satset* is a family of subsets of U satisfying properties (**S1**) and (**S2**). Prove that $satset = satset(\Gamma)$ for some set Γ of fd's over U. *Hint:* Use $\Gamma = \{Y \to Z |$ for each $X \in satset$, if $Y \subseteq X$ then $Z \subseteq X\}$.

Exercise 8.12 Let Σ and Γ be sets of fd's over U. Using the notation of Exercise 8.11,

(a) Show that $satset(\Sigma \cup \Gamma) = satset(\Sigma) \cap satset(\Gamma)$.

(b) Show that $satset(\Sigma^* \cap \Gamma^*) = satset(\Sigma) \wedge satset(\Gamma)$, where for families $\mathcal{F}, \mathcal{G}$, the *wedge* of $\mathcal{F}$ and $\mathcal{G}$ is $\mathcal{F} \wedge \mathcal{G} = \{X \cap Y \mid X \in \mathcal{F} \text{ and } Y \in \mathcal{G}\}$.

(c) For $V \subseteq U$, define $\pi_V \Sigma = \{X \to Y \in \Sigma \mid XY \subseteq V\}$. For $V \subseteq U$ characterize *satset* $(\pi_V(\Sigma^*))$ (where this family is defined with respect to V).

Exercise 8.13

(a) Exhibit a set Σ_1 of fd's over $\{A, B\}$ such that each Armstrong relation for Σ has at least three distinct values occurring in the A column. Exhibit a set Σ_2 of fd's over $\{A, B, C\}$ such that each Armstrong relation for Σ has at least four distinct values occurring in the A column.

(b) [GH83, BDFS84] Let Σ be a set of fd's over U. Recall the notion of saturated set from Exercise 8.11. For an instance I over U, the *agreement set* of I is $agset(I) = \{X \subseteq U \mid \exists\, s, t \in I$ such that $s(A) = t(A)$ iff $A \in X\}$. For a family $\mathcal{F}$ of subsets of U, the *intersection closure* of $\mathcal{F}$ is $intclo(\mathcal{F}) = \{\cap_{i=1}^{n} X_i \mid n \geq 0$ and each $X_i \in \mathcal{F}\}$ (where the empty intersection is defined to be U). Prove that I is an Armstrong relation for Σ iff $intclo(agset(I)) = satset(\Sigma)$.

Exercise 8.14 [Mai80] Let Σ be a set of fd's over U, $X \to Y \in \Sigma$, and let A be an attribute. A is *extraneous* in $X \to Y$ with respect to Σ if either

(a) $(\Sigma - \{X \to Y\}) \cup \{X \to (Y - A)\} \models X \to Y$; or

(b) $(\Sigma - \{X \to Y\}) \cup \{(X - A) \to Y\} \models X \to Y$.

Develop an $O(n^2)$ algorithm that takes as input a set Σ of fd's and produces as output a set $\Sigma' \equiv \Sigma$, where Σ' has no extraneous attributes.

Exercise 8.15 Show that there is no set Σ of jd's and fd $X \to A$ such that $\Sigma \models X \to A$. *Hint:* Show that for any instance I there exists an instance I' such that $I \subseteq I'$ and $I' \models \Sigma$. Then choose I violating $X \to A$.

Exercise 8.16 [Fag77b, Zan76] This exercise refers to the original definition of mvd's. Let U be a set of attributes and $X, Y \subseteq U$. Given an instance I over U and a tuple $x \in \pi_X(I)$, the *image*

of x on Y in I is the set $image_Y(x, I) = \pi_Y(\sigma_{X=x}(I))$ of tuples over Y. Prove that $I \models X \twoheadrightarrow Y$ iff

$$\text{for each } x \in \pi_X(I) \text{ and each } z \in image_Z(x, I),\ image_Y(x, I) = image_Y(xz, I),$$

where $Z = U - XY$ and xz denotes the tuple w over XZ such that $\pi_X(w) = x$ and $\pi_Z(w) = z$.

★ **Exercise 8.17** [BFH77] Complete the proof of Theorem 8.3.5. *Hint:* Of course, the inference rules can be used when reasoning about I. The following claims are also useful:

Claim 1: If $A \in X^+$, then $I \models \emptyset \to A$.

Claim 2: If $A, B \in W_i$ for some $i \in [1, n]$, then $I \models A \to B$.

Claim 3: For each $i \in [1, n]$, $I \models \emptyset \twoheadrightarrow W_i$.

Exercise 8.18 Prove Corollary 8.3.6.

Exercise 8.19 [Kan91] Consider the following set of inference rules:

MVD7: $X \twoheadrightarrow U - X$.

MVD8: If $Y \cap Z = \emptyset$, $X \twoheadrightarrow Y$, and $Z \twoheadrightarrow W$, then $X \twoheadrightarrow W - Y$.

FMVD3: If $Y \cap Z = \emptyset$, $X \twoheadrightarrow Y$, and $Z \to W$, then $X \to Y \cap W$.

Prove that {MVD7, MVD2, MVD8} are sound and complete for inferring implication for mvd's, and that {FD1, FD2, FD3, MVD7, MVD2, MVD8, FMVD1, FMVD3} are sound and complete for inferring implication for fd's and mvd's considered together.

Exercise 8.20 [Bee80] Let Σ be a set of fd's and mvd's, and let $m(\Sigma) = \{X \twoheadrightarrow Y \mid X \twoheadrightarrow Y \in \Sigma\} \cup \{X \twoheadrightarrow A \mid A \in Y \text{ for some } X \to Y \in \Sigma\}$. Prove that

(a) $\Sigma \models X \to Y$ implies $m(\Sigma) \models X \twoheadrightarrow Y$; and

(b) $\Sigma \models X \twoheadrightarrow Y$ iff $m(\Sigma) \models X \twoheadrightarrow Y$.

Hint: For (b) do an induction on proofs using the inference rules.

Exercise 8.21 For sets Σ and Γ of dependencies over U, Σ *implies* Γ *for two-element instances*, denoted $\Sigma \models_2 \Gamma$, if for each instance I over U with $|I| \leq 2$, $I \models \Sigma$ implies $I \models \Gamma$.

(a) [SDPF81] Prove that if $\Sigma \cup \{\sigma\}$ is a set of fd's and mvd's, then $\Sigma \models_2 \sigma$ iff $\Sigma \models \sigma$.

(b) Prove that the equivalence of part (a) does not hold if jd's are included.

(c) Exhibit a jd σ such that there is no set Σ of mvd's with $\sigma \equiv \Sigma$.

♠ **Exercise 8.22** [SDPF81] This exercise develops a close connection between fd's and mvd's, on the one hand, and a fragment of propositional logic, on the other. Let U be a fixed set of attributes. We view each attribute $A \in U$ as a propositional variable. For the purposes of this exercise, a *truth assignment* is a mapping $\xi : U \to \{T, F\}$ (where T denotes *true* and F denotes *false*). Truth assignments are extended to mappings on subsets X of U by $\xi(X) = \wedge_{A \in X} \xi(A)$. A truth assignment ξ *satisfies* an fd $X \to Y$, denoted $\xi \models X \to Y$, if $\xi(X) = T$ implies $\xi(Y) = T$. It satisfies an mvd $X \twoheadrightarrow Y$, denoted $\xi \models X \twoheadrightarrow Y$, if $\xi(X) = T$ implies that either $\xi(Y) = T$ or $\xi(U - Y) = T$. Given a set $\Sigma \cup \{\sigma\}$ of fd's and mvd's, Σ implies σ *in the propositional calculus*, denoted $\Sigma \models_{\text{prop}} \sigma$, if for each truth assignment ξ, $\xi \models \Sigma$ implies $\xi \models \sigma$. Prove that for all sets $\Sigma \cup \{\sigma\}$ of fd's and mvd's, $\Sigma \models \sigma$ iff $\Sigma \models_{\text{prop}} \sigma$.

★ **Exercise 8.23** [Bis80] Exhibit a set of inference rules for mvd's that are sound and complete in the context in which an underlying set of attributes is not fixed.

♠ **Exercise 8.24**

(a) Prove Proposition 8.4.2.

(b) Prove Lemma 8.4.3.

(c) Prove Lemma 8.4.4. What is the maximum size attainable by the tableau in the result of a terminal chasing sequence?

(d) Prove Lemma 8.4.17.

♠ **Exercise 8.25**

(a) Describe a polynomial time algorithm for computing the chase of a tableau query by Σ, assuming that Σ contains only fd's.

(b) Show that the problem of deciding whether a jd can be applied to a tableau query is NP-complete if the schema is considered variable, and polynomial if the schema is considered fixed. *Hint:* Use Exercise 6.16.

(c) Prove that it is NP-hard, given a tableau query (T, t) and a set Σ of fd's and jd's, to compute $chase(T, t, \Sigma)$ (this assumes that the schema is part of the input and thus not fixed).

(d) Describe an exponential time algorithm for computing the chase by a set of fd's and jd's. (Again the schema is not considered fixed.)

Exercise 8.26 Prove Proposition 8.4.6. *Hint:* Rather than modifying the proof of Theorem 8.4.18, prove as a lemma that if $\Sigma \models \sigma$, then $chase(T, t, \Sigma) = chase(T, t, \Sigma \cup \{\sigma\})$.

Exercise 8.27

(a) Verify that the results concerning the chase generalize immediately to the context in which database schemas as opposed to relation schemas are used.

(b) Describe how to generalize the chase to tableau in which constants occur, and state and prove the results about the chase and tableau queries. *Hint:* If the chase procedure attempts to equate two distinct constants (a situation not occurring before), we obtain a particular new tableau, called T_{false}, which corresponds to the query producing an empty result on all input instances.

Exercise 8.28 For each of the following relation schemas R, SPJ expressions q over R, and dependencies Σ over R, simplify q knowing that it is applied only to instances over R satisfying Σ. Use tableau minimization and the chase.

(a) $sort(R) = ABC$, $q = \pi_{AC}(\pi_{AB}(\sigma_{A=2}(R) \bowtie \pi_{BC}(R)) \bowtie \pi_{AB}(\sigma_{B=8}(R) \bowtie \pi_{BC}(R))$, $\Sigma = \{A \to C, B \to C\}$

(b) $sort(R) = ABCD$, $q = \pi_{BC}(R) \bowtie \pi_{ABD}(R)$, $\Sigma = \{B \twoheadrightarrow CD, B \twoheadrightarrow D\}$

(c) $sort(R) = ABCD$, $q = \pi_{ABD}(R) \bowtie \pi_{AC}(R)$, $\Sigma = \{A \to B, B \twoheadrightarrow C\}$.

♠ **Exercise 8.29** Prove Theorem 8.4.12.

♠ **Exercise 8.30** Prove Theorem 8.4.13(a) [BV80a] and Theorem 8.4.13(b) [FT83].

Exercise 8.31 [MMS79] Describe an algorithm based on the chase for

(a) computing the closure of an attribute set X under a set Σ of fd's and jd's (where the notion of closure is extended to include all fd's implied by Σ); and

(b) computing the dependency basis (see Section 8.3) of a set X of attributes under a set Σ of fd's and jd's (where the notion of dependency basis is extended to include fd's in the natural manner).

Exercise 8.32 [GH86] Suppose that the underlying domain **dom** has a total order $\leq$. Let $U = \{A_1, \ldots, A_n\}$ be a set of attributes. For each $X \subseteq U$, define the partial order $\leq_X$ over the set of tuples of X by $t \leq_X t'$ iff $t(A) \leq t'(A)$ for each $A \in X$. A *sort set dependency* (SSD) over U is an expression of the form $s(X)$, where $X \subseteq U$. An instance I over U satisfies $s(X)$, denoted $I \models s(X)$, if $\leq_X$ is a total order on $\pi_X(I)$.

(a) Show that the following set of inference rules is sound and complete for finite logical implication between SSDs:
SSD1: If A is an attribute, then $s(A)$.
SSD2: If $s(X)$ and $Y \subseteq X$, then $s(Y)$.
SSD3: If $s(X)$, $s(Y)$ and $s(X \triangle Y)$, then $s(XY)$ [where $X \triangle Y$ denotes $(X - Y) \cup (Y - X)$, i.e., the symmetric difference of X and Y].

(b) Exhibit a polynomial time algorithm for inferring logical implication between sets of SSDs.

(c) Describe how SSDs might be used in connection with indexes.

9 Inclusion Dependency

Vittorio: *Fd's and jd's give some structure to relations.*
Alice: *But there are no connections between them.*
Sergio: *Making connections is the next step . . .*
Riccardo: *. . . with some unexpected consequences.*

The story of inclusion dependencies starts in a manner similar to that for functional dependencies: Implication is decidable (although here it is PSPACE-complete), and there is a simple set of inference rules that is sound and complete. But the story becomes much more intriguing when functional and inclusion dependencies are taken together. First, the notion of logical implication will have to be refined because the behavior of these dependencies taken together is different depending on whether infinite instances are permitted. Second, both notions of logical implication are nonrecursive. And third, it can be proven in a formal sense that no "finite" axiomatization exists for either notion of logical implication of the dependencies taken together. At the end of this chapter, two restricted classes of inclusion dependencies are discussed. These are significant because they arise in modeling certain natural relationships such as those encountered in semantic data models. Positive results have been obtained for inclusion dependencies from these restricted classes considered with fd's and other dependencies.

Unlike fd's or jd's, a single inclusion dependency may refer to more than one relation. Also unlike fd's and jd's, inclusion dependencies are "untyped" in the sense that they may call for the comparison of values from columns (of the same or different relations) that are labeled by different attributes. A final important difference from fd's and jd's is that inclusion dependencies are "embedded." Speaking intuitively, to satisfy an inclusion dependency the presence of one tuple in an instance may call for the presence of another tuple, of which only some coordinate values are determined by the dependency and the first tuple. These and other differences will be discussed further in Chapter 10.

9.1 Inclusion Dependency in Isolation

To accommodate the fact that inclusion dependencies permit the comparison of values from different columns of one or more relations, we introduce the following notation. Let R be a relation schema and $X = A_1, \ldots, A_n$ a *sequence* of attributes (possibly with repeats) from R. For an instance I of R, the *projection* of I onto the sequence X, denoted $I[X]$, is the n-ary relation $\{\langle t(A_1), \ldots, t(A_n)\rangle \mid t \in I\}$.

The syntax and semantics of inclusion dependencies is now given by the following:

Definition 9.1.1 Let $\mathbf{R}$ be a relational schema. An *inclusion dependency* (ind) over $\mathbf{R}$ is an expression of the form $\sigma = R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$, where

(a) R, S are (possibly identical) relation names in $\mathbf{R}$,

(b) $A_1, \ldots, A_m$ is a sequence of distinct attributes of $sort(R)$, and

(c) $B_1, \ldots, B_m$ is a sequence of distinct attributes of $sort(S)$.

An instance $\mathbf{I}$ of $\mathbf{R}$ *satisfies* σ, denoted $\mathbf{I} \models \sigma$, if

$$\mathbf{I}(R)[A_1, \ldots, A_m] \subseteq \mathbf{I}(S)[B_1, \ldots, B_m].$$

Satisfaction of a set of ind's is defined in the natural manner.

To illustrate this definition, we recall an example from the previous chapter.

Example 9.1.2 There are two relations: *Movies* with attributes *Title, Director, Actor* and *Showings* with *Theater, Screen, Title, Snack*; and we have an ind

$$Showings[Title] \subseteq Movies[Title].$$

The generalization of ind's to permit repeated attributes on the left-or right-hand side is considered in Exercise 9.4.

The notion of *logical implication* between sets of ind's is defined in analogy with that for fd's. (This will be refined later when fd's and ind's are considered together.)

Rules for Inferring ind Implication

The following set of inference rules will be shown sound and complete for inferring logical implication between sets of ind's. The variables X, Y, and Z range over sequences of distinct attributes; and R, S, and T range over relation names.

IND1: (reflexivity) $R[X] \subseteq R[X]$.

IND2: (projection and permutation) If $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$, then $R[A_{i_1}, \ldots, A_{i_k}] \subseteq S[B_{i_1}, \ldots, B_{i_k}]$ for each sequence $i_1, \ldots, i_k$ of distinct integers in $\{1, \ldots, m\}$.

IND3: (transitivity) If $R[X] \subseteq S[Y]$ and $S[Y] \subseteq T[Z]$, then $R[X] \subseteq T[Z]$.

The notions of proof and of provability (denoted $\vdash$) using these rules are defined in analogy with that for fd's.

Theorem 9.1.3 The set {IND1, IND2, IND3} is sound and complete for logical implication of ind's.

Proof Soundness of the rules is easily verified. For completeness, let Σ be a set of ind's over database schema $\mathbf{R} = \{R_1, \ldots, R_n\}$, and let $\sigma = R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$

be an ind over $\mathbf{R}$ such that $\Sigma \models \sigma$. We construct an instance $\mathbf{I}$ of $\mathbf{R}$ and use it to demonstrate that $\Sigma \vdash \sigma$.

To begin, let s' be the tuple over R_a such that $s'(A_i) = i$ for $i \in [1, m]$ and $s'(B) = 0$ otherwise. Set $\mathbf{I}(R_a) = \{s'\}$ and $\mathbf{I}(R_j) = \emptyset$ for $j \neq a$. We now apply the following rule to $\mathbf{I}$ until it can no longer be applied.

$$
\begin{aligned}
&\text{If } R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma \text{ and } t \in \mathbf{I}(R_i), \text{ then add} \\
(*) \qquad &u \text{ to } \mathbf{I}(R_j), \text{ where } u(D_l) = t(C_l) \text{ for } l \in [1, k] \text{ and } u(D) = 0 \text{ for } D \\
&\qquad \notin \{D_1, \ldots, D_k\}.
\end{aligned}
$$

Application of this rule will surely terminate, because all tuples are constructed from a set of at most $m + 1$ values. Clearly the result of applying this rule until termination is unique, so let $\mathbf{J}$ be this result.

Remark 9.1.4 This construction is reminiscent of the chase for join dependencies. It differs because the ind's may be embedded. Intuitively, an ind may not specify all the entries of the tuples we are adding. In the preceding rule (∗), the same value (0) is always used for tuple entries that are otherwise unspecified. ■

It is easily seen that $\mathbf{J} \models \Sigma$. Because $\Sigma \models \sigma$, we have $\mathbf{J} \models \sigma$. To conclude the proof, we show the following:

$$
\begin{aligned}
&\text{If for some } R_j \text{ in } \mathbf{R}, u \in \mathbf{J}(R_j), \text{ integer } q, \text{ and distinct attributes} \\
(**) \qquad &C_1, \ldots, C_q \text{ in } sort(R_j), u(C_p) > 0 \text{ for } p \in [1, q], \text{ then} \\
&\qquad \Sigma \vdash R_a[A_{u(C_1)}, \ldots, A_{u(C_q)}] \subseteq R_j[C_1, \ldots, C_q].
\end{aligned}
$$

Suppose that (∗∗) holds. Let s'' be a tuple of $\mathbf{J}(R_b)$ such that $s''[B_1, \ldots, B_m] = s'[A_1, \ldots, A_m]$. (Such a tuple exists because $\mathbf{J} \models \sigma$.) Use (∗∗) with $R_j = R_b$, $q = m$, $C_1, \ldots, C_q = B_1, \ldots, B_m$.

To demonstrate (∗∗), we show inductively that it holds for all tuples of $\mathbf{J}$ by considering them in the order in which they were inserted. The claim holds for s in $\mathbf{J}(R_a)$ by IND1. Suppose now that

- $\mathbf{I}'$ is the instance obtained after k applications of the rule for some $k \geq 0$;
- the claim holds for all tuples in $\mathbf{I}'$; and
- u is added to R_j by the next application of rule (∗), due to the ind $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma$ and tuple $t \in \mathbf{I}'(R_i)$.

Now let $\{E_1, \ldots, E_q\}$ be a set of distinct attributes in $sort(R_j)$ with $u(E_p) > 0$ for $p \in [1, q]$. By the construction of u in (*), $\{E_1, \ldots, E_q\} \subseteq \{D_1, \ldots, D_k\}$. Choose the mapping ρ such that $D_{\rho(p)} = E_p$ for $p \in [1, q]$. Because $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma$, IND2 yields

$$\Sigma \vdash R_i[C_{\rho(1)}, \ldots, C_{\rho(q)}] \subseteq R_j[E_1, \ldots, E_q].$$

By the inductive assumption,

$$\Sigma \vdash R_a[A_{t(C_{\rho(1)})}, \ldots, A_{t(C_{\rho(q)})}] \subseteq R_i[C_{\rho(1)}, \ldots, C_{\rho(q)}].$$

Thus, by IND3,

$$\Sigma \vdash R_a[A_{t(C_{\rho(1)})}, \ldots, A_{t(C_{\rho(q)})}] \subseteq R_j[E_1, \ldots, E_q].$$

Finally, observe that for each p, $t(C_{\rho(p)}) = u(D_{\rho(p)}) = u(E_p)$, so

$$\Sigma \vdash R_a[A_{u(E_1)}, \ldots, A_{u(E_q)}] \subseteq R_j[E_1, \ldots, E_q].$$

Deciding Logical Implication for ind's

The proof of Theorem 9.1.3 yields a decision procedure for determining logical implication between ind's. To see this, we use the following result:

Proposition 9.1.5 Let Σ be a set of ind's over $\mathbf{R}$ and $R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$. Then $\Sigma \models R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$ iff there is a sequence $R_{i_1}[\vec{C}_1], \ldots, R_{i_k}[\vec{C}_k]$ such that

(a) $R_{i_j} \in \mathbf{R}$ for $j \in [1, k]$;
(b) $\vec{C}_j$ is a sequence of m distinct attributes in $sort(R_{i_j})$ for $j \in [1, k]$;
(c) $R_{i_1}[\vec{C}_1] = R_a[A_1, \ldots, A_m]$;
(d) $R_{i_k}[\vec{C}_k] = R_b[B_1, \ldots, B_m]$;
(e) $R_{i_j}[\vec{C}_j] \subseteq R_{i_{j+1}}[\vec{C}_{j+1}]$ can be obtained from an ind in Σ by one application of rule IND2, for $j \in [1, (k-1)]$.

Crux Use the instance $\mathbf{J}$ constructed in the proof of Theorem 9.1.3. Working backward from the tuple s'' in $\mathbf{J}(R_b)$, a chain of relation-tuple pairs (R_{i_j}, s_j) can be constructed so that each of $1, \ldots, m$ occurs exactly once in s_j, and s_{j+1} is inserted into $\mathbf{I}$ as a result of s_j and IND2. ■

Based on this, it is straightforward to verify that the following algorithm determines logical implication between ind's. Note that only ind's of arity m are considered in the algorithm.

Algorithm 9.1.6

Input: A set Σ of ind's over $\mathbf{R}$ and ind $R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$.

Output: Determine whether $\Sigma \models R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$.

Procedure: Build a set $\mathcal{E}$ of expressions of the form $R_i[C_1, \ldots, C_m]$ as follows:

1. $\mathcal{E} := \{R_a(A_1, \ldots, A_m)\}$.

2. Repeat until $R_b[B_1, \ldots, B_m] \in \mathcal{E}$ or no change possible:
   If $R_i[C_1, \ldots, C_m] \in \mathcal{E}$ and

$$R_i[C_1, \ldots, C_m] \subseteq R_j[D_1, \ldots, D_m]$$

   can be derived from an ind of Σ by one application of IND2, then insert $R_j[D_1, \ldots, D_m]$ into $\mathcal{E}$.
3. If $R_b[B_1, \ldots, B_m] \in \mathcal{E}$ then return *yes*; else return *no*. ■

As presented, the preceding algorithm is nondeterministic and might therefore take more than polynomial time. The following result shows that this is indeed likely for any algorithm for deciding implication between ind's.

THEOREM 9.1.7 Deciding logical implication for ind's is PSPACE-complete.

Crux Algorithm 9.1.6 can be used to develop a nondeterministic polynomial space procedure for deciding logical implication between ind's. By Savitch's theorem (which states that PSPACE = NPSPACE), this can be transformed into a deterministic algorithm that runs in polynomial space. To show that the problem is PSPACE-hard, we describe a reduction from the problem of linear space acceptance.

A (Turing) machine is *linear bounded* if on each input of size n, the machine does not use more that n tape cells. The problem is the following:

Linear Space Acceptance (LSA) problem

Input: The description of a linear bounded machine M and an input word x;

Output: yes iff M accepts x.

The heart of the proof is, given an instance (M, x) of the LSA problem, to construct a set Σ of ind's and an ind σ such that $\Sigma \models \sigma$ iff x is accepted by M.

Let $M = (K, \Gamma, \Delta, s, h)$ be a Turing machine with states K, alphabet Γ, transition relation Δ, start state s, and accepting state h; and let $x = x_1 \ldots x_n \in \Gamma^*$ have length n.

Configurations of M are viewed as elements of $\Gamma^* K \Gamma^+$ with length $n + 1$, where the placement of the state indicates the head position (the state is listed immediately left of the scanned letter). Observe that transitions can be described by expressions of the form $\alpha_1, \alpha_2, \alpha_3 \to \beta_1, \beta_2, \beta_3$ with $\alpha_1, \ldots, \beta_3$ in $(K \cup \Gamma)$. For instance, the transition

"if reading b in state p, then overwrite with c and move left"

corresponds to $a, p, b \to p, a, c$ for each a in Γ. Let χ be the set of all such expressions corresponding to transitions of M.

The initial configuration is sx. The final configuration is $h\not{b}^n$ for some particular letter $\not{b}$, iff M accepts x.

The ind's of Σ are defined over a single relation R. The attributes of R are $\{A_{i,j} \mid i \in (K \cup \Gamma), j \in \{1, 2, \ldots, n+1\}\}$. The intuition here is that the attribute $A_{p,j}$ corresponds to the statement that the j^{th} symbol in a given configuration is p. To simplify the presentation, attribute $A_{a,k}$ is simply denoted by the pair (a, k).

The ind σ is

$$R[(s, 1), (x_1, 2), \ldots, (x_n, n+1)] \subseteq R[(h, 1), (\not{p}, 2), \ldots, (\not{p}, n+1)].$$

The ind's in Σ correspond to valid moves of M. In particular, for each $j \in [1, n-1]$, Σ includes all ind's of the form

$$R[(\alpha_1, j), (\alpha_2, j+1), (\alpha_3, j+2), \vec{A}\,] \subseteq R[(\beta_1, j), (\beta_2, j+1), (\beta_3, j+2), \vec{A}\,],$$

where $\alpha_1, \alpha_2, \alpha_3 \to \beta_1, \beta_2, \beta_3$ is in χ, and $\vec{A}$ is an arbitrary fixed sequence that lists all of the attributes in $\Gamma \times \{1, \ldots, j-1, j+3, \ldots, n+1\}$. Thus each ind in Σ has arity $3 + (n-2)|\Gamma|$, and $|\Sigma| \leq n|\Delta|$.

Although the choice of $\vec{A}$ permits the introduction of many ind's, observe that the construction is still polynomial in the size of the linear space automaton problem (M, x). Using Proposition 9.1.5, it is now straightforward to verify that $\Sigma \models \sigma$ iff M has an accepting computation of x. ■

Although the general problem of deciding implication for ind's is PSPACE-complete, naturally arising special cases of the problem have polynomial time solutions. This includes the family of ind's that are at most k-ary (ones in which the sequences of attributes have length at most some fixed k) and ind's that have the form $R[\vec{A}\,] \subseteq S[\vec{A}\,]$ (see Exercise 9.10). The latter case arises in examples such as $Grad - Stud[Name, Major] \subseteq Student[Name, Major]$. This theme is also examined at the end of this chapter.

9.2 Finite versus Infinite Implication

We now turn to the interaction between ind's and fd's, which leads to three interesting phenomena. The first of these requires a closer look at the notion of logical implication.

Consider the notion of logical implication used until now: Σ logically implies σ if for all relation (or database) instances $\mathbf{I}$, $\mathbf{I} \models \Sigma$ implies $\mathbf{I} \models \sigma$. Although this notion is close to the corresponding notion of mathematical logic, it is different in a crucial way: In the context of databases considered until now, only *finite* instances are considered. From the point of view of logic, the study of logical implication conducted so far lies within *finite model theory*.

It is also interesting to consider logical implication in the traditional mathematical logic framework in which infinite database instances are permitted. As will be seen shortly, when fd's or ind's are considered separately, permitting infinite instances has no impact on logical implication. However, when fd's and ind's are taken together, the two flavors of logical implication do not coincide.

The notion of infinite relation and database instances is defined in the natural manner. An *unrestricted* relation (database) instance is a relation (database) instance that is either finite or infinite. Based on this, we now redefine "unrestricted implication" to permit infinite instances, and we define "finite logical implication" for the case in which only finite instances are considered.

R	A	B
	1	0
	2	1
	3	2
	4	3
	⋮	⋮

(a)

R	A	B
	1	1
	2	1
	3	2
	4	3
	⋮	⋮

(b)

Figure 9.1: Instances used for distinguishing $\models_{\text{fin}}$ and $\models_{\text{unr}}$

DEFINITION 9.2.1 A set Σ of dependencies over $\mathbf{R}$ *implies without restriction* a dependency σ, denoted $\Sigma \models_{\text{unr}} \sigma$, if for each unrestricted instance $\mathbf{I}$ of $\mathbf{R}$, $\mathbf{I} \models \Sigma$ implies $\mathbf{I} \models \sigma$. A set Σ of dependencies over $\mathbf{R}$ *finitely implies* a dependency σ, denoted $\Sigma \models_{\text{fin}} \sigma$, if for each (finite) instance $\mathbf{I}$ of $\mathbf{R}$, $\mathbf{I} \models \Sigma$ implies $\mathbf{I} \models \sigma$.

If finite and unrestricted implication coincide, or if the kind of implication is understood from the context, then we may use $\models$ rather than $\models_{\text{fin}}$ or $\models_{\text{unr}}$. This is what we implicitly did so far by using $\models$ in place of $\models_{\text{fin}}$.

Of course, if $\Sigma \models_{\text{unr}} \sigma$, then $\Sigma \models_{\text{fin}} \sigma$. The following shows that the converse need not hold:

THEOREM 9.2.2

(a) There is a set Σ of fd's and ind's and an ind σ such that $\Sigma \models_{\text{fin}} \sigma$ but $\Sigma \not\models_{\text{unr}} \sigma$.

(b) There is a set Σ of fd's and ind's and an fd σ such that $\Sigma \models_{\text{fin}} \sigma$ but $\Sigma \not\models_{\text{unr}} \sigma$.

Proof For part (a), let R be binary with attributes A, B; let $\Sigma = \{A \to B, R[A] \subseteq R[B]\}$; and let σ be $R[B] \subseteq R[A]$. To see that $\Sigma \models_{\text{fin}} \sigma$, let I be a finite instance of R that satisfies Σ. Because $I \models A \to B$, $|\pi_A(I)| \geq |\pi_B(I)|$ and because $I \models R[A] \subseteq R[B]$, $|\pi_B(I)| \geq |\pi_A(I)|$. It follows that $|\pi_A(I)| = |\pi_B(I)|$. Because I is finite and $\pi_A(I) \subseteq \pi_B(I)$, it follows that $\pi_B(I) \subseteq \pi_A(I)$ and $I \models R[B] \subseteq R[A]$.

On the other hand, the instance shown in Fig. 9.1(a) demonstrates that $\Sigma \not\models_{\text{unr}} \sigma$.

For part (b), let Σ be as before, and let σ be the fd $B \to A$. As before, if $I \models \Sigma$, then $|\pi_A(I)| = |\pi_B(I)|$. Because $I \models A \to B$, each tuple in I has a distinct A-value. Thus the number of B-values occurring in I equals the number of tuples in I. Because I is finite, this implies that $I \models B \to A$. Thus $\Sigma \models_{\text{fin}} \sigma$. On the other hand, the instance shown in Fig. 9.1(b) demonstrates that $\Sigma \not\models_{\text{unr}} \sigma$. ■

It is now natural to reconsider implication for fd's, jd's, and inds taken separately and in combinations. Are unrestricted and finite implication different in these cases? The answer is given by the following:

THEOREM 9.2.3 Unrestricted and finite implication coincide for fd's and jd's considered separately or together and for ind's considered alone.

Proof Unrestricted implication implies finite implication by definition. For fd's and jd's taken separately or together, Theorem 8.4.12 on the relationship between chasing and logical implication can be used to obtain the opposite implication. For ind's, Theorem 9.1.3 shows that finite implication and provability by the ind inference rules are equivalent. It is easily verified that these rules are also sound for unrestricted implication. Thus finite implication implies unrestricted implication for ind's as well. ■

The notion of finite versus unrestricted implication will be revisited in Chapter 10, where dependencies are recast into a logic-based formalism.

Implication Is Undecidable for fd's + ind's

As will be detailed in Chapter 10, fd's and ind's (and most other relational dependencies) can be represented as sentences in first-order logic. By Gödel's Completeness Theorem implication is recursively enumerable for first-order logic. It follows that unrestricted implication is r.e. for fd's and ind's considered together. On the other hand, finite implication for fd's and ind's taken together is co-r.e. This follows from the fact that there is an effective enumeration of all finite instances over a fixed schema; if $\Sigma \not\models_{\text{fin}} \sigma$, then a witness of this fact will eventually be found. When unrestricted and finite implication coincide, this pair of observations is sufficient to imply decidability of implication; this is not the case for fd's and ind's.

The Word Problem for (Finite) Monoids

The proof that (finite) implication for fd's and ind's taken together is undecidable uses a reduction from the word problem for monoids, which we discuss next.

A monoid is a set with an associative binary operation $\circ$ defined on it and an identity element ε. Let Γ be a finite alphabet and Γ^* the *free monoid* generated by Γ (i.e., the set of finite words with letters in Γ with the concatenation operation). Let $E = \{\alpha_i = \beta_i \mid i \in [1..n]\}$ be a finite set of equalities, and let e be an additional equality $\alpha = \beta$, where $\alpha_i, \beta_i, \alpha, \beta \in \Gamma^*$. Then E *(finitely) implies* e, denoted $E \models_{\text{unr}} e$ ($E \models_{\text{fin}} e$), if for each (finite) monoid M and homomorphism $h : \Gamma^* \to M$, if $h(\alpha_i) = h(\beta_i)$ for each $i \in [1..n]$, then $h(\alpha) = h(\beta)$. The *word problem* for (finite) monoids is to decide, given E and e, whether $E \models_{\text{unr}} e$ ($E \models_{\text{fin}} e$). Both the word problem for monoids and the word problem for finite monoids are undecidable.

Using this, we have the following:

THEOREM 9.2.4 Unrestricted and finite implication for fd's and ind's considered together are undecidable. In particular, let Σ range over sets of fd's and ind's. The following sets are not recursive:

(a) $\{(\Sigma, \sigma) \mid \sigma \text{ an ind and } \Sigma \models_{\text{unr}} \sigma\}$; $\{(\Sigma, \sigma) \mid \sigma \text{ an ind and } \Sigma \models_{\text{fin}} \sigma\}$;

(b) $\{(\Sigma, \sigma) \mid \sigma \text{ an fd and } \Sigma \models_{\text{unr}} \sigma\}$; and $\{(\Sigma, \sigma) \mid \sigma \text{ an fd and } \Sigma \models_{\text{fin}} \sigma\}$.

Crux We prove (a) using a reduction from the word problem for (finite) monoids to the (finite) implication problem for fd's and ind's. The proof of part (b) is similar and is left for Exercise 9.5. We first consider the unrestricted case.

Let Γ be a fixed alphabet. Let $E = \{\alpha_i = \beta_i \mid i \in [1, n]\}$ be a set of equalities over Γ^*, and let e be another equality $\alpha = \beta$. A *prefix* is defined to be any prefix of α_i, β_i, α, or β (including the empty string ε, and full words α_1, β_1, etc.). A single relation R is used, which has attributes

(i) A_γ, for each prefix γ;
(ii) A_x, A_y, A_{xy};
(iii) A_{ya}, for each $a \in \Gamma$; and
(iv) A_{xya}, for each $a \in \Gamma$;

where x and y are two fixed symbols.

To understand the correspondence between constrained relations and homomorphisms over monoids, suppose that there is a homomorphism h from Γ^* to some monoid M. Intuitively, a tuple of R will hold information about two elements $h(x), h(y)$ of M (in columns A_x, A_y, respectively) and their product $h(x) \circ h(y) = h(xy)$ (in column A_{xy}). For each a in Γ, tuples will also hold information about $h(ya)$ and $h(xya)$ in columns A_{ya}, A_{xya}. More precisely, the instance $I_{M,h}$ corresponding to the monoid M and the homomorphism $h : \Gamma^* \to M$ is defined by

$$I_{M,h} = \{t_{u,v} \mid u, v \in \Gamma^*\},$$

where for each $u, v \in \Gamma^*$, $t_{u,v}$ is the tuple such that

$$t_{u,v}(A_x) = h(u), \qquad t_{u,v}(A_\gamma) = h(\gamma), \text{ for each prefix } \gamma,$$
$$t_{u,v}(A_y) = h(v), \qquad t_{u,v}(A_{ya}) = h(va), \text{ for each } a \in \Gamma,$$
$$t_{u,v}(A_{xy}) = h(uv), \qquad t_{u,v}(A_{xya}) = h(uva), \text{ for each } a \in \Gamma.$$

Formally, to force the correspondence between the relations and homomorphisms over monoids, we use a set Σ of dependencies. In other words, we wish to find a set Σ of dependencies that characterizes precisely the instances over R that correspond to some homomorphism h from Γ^* to some monoid M. The key to the proof is that this can be done using just fd's and ind's. Strictly speaking, the dependencies of (8) in the following list are not ind's because an attribute is repeated in the left-hand side. As discussed in Exercise 9.4(e), the set of dependencies used here can be modified to a set of proper ind's that has the desired properties. In addition, we use fd's with an empty left-hand side, which are sometimes not considered as real fd's. The use of such dependencies is not crucial. A slightly more complicated proof can be found that uses only fd's with a nonempty left-hand side. The set Σ is defined as follows:

1. $\emptyset \rightarrow A_\gamma$ for each prefix γ;
2. $A_x A_y \rightarrow A_{xy}$;
3. $A_y \rightarrow A_{ya}$, for each $a \in \Gamma$;
4. $R[A_\varepsilon] \subseteq R[A_y]$;
5. $R[A_\gamma, A_{\gamma a}] \subseteq R[A_y, A_{ya}]$, for each $a \in \Gamma$ and prefix γ;
6. $R[A_{xy}, A_{xya}] \subseteq R[A_y, A_{ya}]$, for each $a \in \Gamma$;
7. $R[A_x, A_{ya}, A_{xya}] \subseteq R[A_x, A_y, A_{xy}]$, for each $a \in \Gamma$;
8. $R[A_y, A_\varepsilon, A_y] \subseteq R[A_x, A_y, A_{xy}]$; and
9. $R[A_{\alpha_i}] \subseteq R[A_{\beta_i}]$, for each $i \in [1, n]$.

The ind σ is $R[A_\alpha] \subseteq R[A_\beta]$.

Let I be an instance satisfying Σ. Observe that I has to satisfy a number of implied properties. In particular, one can verify that I also satisfies the following property:

$$R[A_{xya}] \subseteq R[A_{ya}] \subseteq R[A_y] = R[A_{xy}] \subseteq R[A_x]$$

and that $adom(I) \subseteq I[A_x]$.

We now show that $\Sigma \models_{\text{unr}} \sigma$ iff $E \models_{\text{unr}} e$. We first show that $E \not\models_{\text{unr}} e$ implies $\Sigma \not\models_{\text{unr}} \sigma$. Suppose that there is a monoid M and homomorphism $h : \Gamma^* \rightarrow M$ that satisfies the equations of E but violates the equation e. Consider $I_{M,h}$ defined earlier. It is straightforward to verify that $I \models \Sigma$ but $I \not\models \sigma$.

For the opposite direction, suppose now that $E \models_{\text{unr}} e$, and let I be a (possibly infinite) instance of R that satisfies Σ. To conclude the proof, it must be shown that $I[A_\alpha] \subseteq I[A_\beta]$. (Observe that these two relations both consist of a single tuple because of the fd's with an empty left-hand-side.)

We now define a function $h : \Gamma^* \rightarrow adom(I)$. We will prove that h is a homomorphism from Γ^* to a free monoid whose elements are $h(\Gamma^*)$ and that satisfies the equations of E (and hence, e). We will use the fact that the monoid satisfies e to derive that $I[A_\alpha] \subseteq I[A_\beta]$.

We now give an inductive definition of h and show that it has the property that $h(v) \in I[A_y]$ for each $v \in \Gamma^*$.

Basis: Set $h(\varepsilon)$ to be the element in $I[A_\varepsilon]$. Note that $h(\varepsilon)$ is also in $I[A_y]$ because $R[A_\varepsilon] \subseteq R[A_y] \in \Sigma$.

Inductive step: Given $h(v)$ and $a \in \Gamma$, let $t \in I$ be such that $t[A_y] = h(v)$. Define $h(va) = t(A_{ya})$. This is uniquely determined because $A_y \rightarrow A_{ya} \in \Sigma$. In addition, $h(va) \in I[A_y]$ because $R[A_x, A_{ya}, A_{xya}] \subseteq R[A_x, A_y, A_{xy}] \in \Sigma$.

We next show by induction on v that

$$(\dagger) \qquad \langle h(u), h(v), h(uv)\rangle \in I[A_x, A_y, A_{xy}] \text{ for each } u, v \in \Gamma^*.$$

For a fixed u, the basis (i.e., $v = \varepsilon$) is provided by the fact that $h(u) \in I[A_y]$ and the ind $R[A_y, A_\varepsilon, A_y] \subseteq R[A_x, A_y, A_{xy}] \in \Sigma$. For the inductive step, let $\langle h(u), h(v), h(uv)\rangle \in I[A_x, A_y, A_{xy}]$ and $a \in \Gamma$. Let $t \in I$ be such that $t[A_x, A_y, A_{xy}] = \langle h(u), h(v), h(uv)\rangle$.

Then by construction of h, $h(va) = t(A_{ya})$, and from the ind $R[A_{xy}, A_{xya}] \subseteq R[A_y, A_{ya}]$, we have $h(uva) = t(A_{xya})$. Finally, the ind $R[A_x, A_{ya}, A_{xya}] \subseteq R[A_x, A_y, A_{xy}]$ implies that $\langle h(u), h(va), h(uva)\rangle \in I[A_x, A_y, A_{xy}]$ as desired.

Define the binary operation $\circ$ on $h(\Gamma^*)$ as follows. For $a, b \in h(\Gamma^*)$, let

$$a \circ b = c \text{ if for some } t \in I,\ t[A_x, A_y, A_{xy}] = \langle a, b, c\rangle.$$

There is such a tuple by (†) and c is uniquely defined because $A_x, A_y \to A_{xy} \in \Sigma$. Furthermore, by (†), for each u, v, $h(u) \circ h(v) = h(uv)$. Thus for $h(u), h(v), h(w)$ in $h(\Gamma^*)$,

$$(h(u) \circ h(v)) \circ h(w) = h(uvw) = (h(u) \circ h(v)) \circ h(w),$$

and

$$h(u) \circ h(\varepsilon) = h(u)$$

so $(h(\Gamma^*), \circ)$ is a monoid. In addition, h is a homomorphism from the free monoid over Γ^* to the monoid $(h(\Gamma^*), \circ)$.

It is easy to see that $I[A_{\alpha_i}] = \{h(\alpha_i)\}$ and $I[A_{\beta_i}] = \{h(\beta_i)\}$ for $i \in [1, n]$. Let i be fixed. Because $R[A_{\alpha_i}] \subseteq R[A_{\beta_i}]$, $h(\alpha_i) = h(\beta_i)$. Because $E \models_{\text{unr}} e$, $h(\alpha) = h(\beta)$. Thus $I[A_\alpha] = \{h(\alpha)\} = \{h(\beta)\} = I[A_\beta]$. It follows that $I \models_{\text{unr}} R[A_\alpha] \subseteq R[A_\beta]$ as desired.

This completes the proof for the unrestricted case. For the finite case, note that everything has to be finite: The monoid is finite, I is finite, and the monoid $h[\Gamma^*]$ is finite. The rest of the argument is the same. ■

The issue of decidability of finite and unrestricted implication for classes of dependencies is revisited in Chapter 10.

9.3 Nonaxiomatizability of fd's + ind's

The inference rules given previously for fd's, mvd's and ind's can be viewed as "inference rule schemas," in the sense that each of them can be instantiated with specific attribute sets (sequences) to create infinitely many ground inference rules. In these cases the family of inference rule schemas is finite, and we informally refer to them as "finite axiomatizations."

Rather than formalizing the somewhat fuzzy notion of inference rule schema, we focus in this section on families $\mathcal{R}$ of ground inference rules. A (*ground*) *axiomatization* of a family $\mathcal{S}$ of dependencies is a set of ground inference rules that is sound and complete for (finite or unrestricted) implication for $\mathcal{S}$. Two properties of an axiomatization $\mathcal{R}$ will be considered, namely: (1) $\mathcal{R}$ is recursive, and (2) $\mathcal{R}$ is k-ary, in the sense (formally defined later in this section) that each rule in $\mathcal{R}$ has at most k dependencies in its condition.

Speaking intuitively, if $\mathcal{S}$ has a "finite axiomatization," that is, if there is a finite family $\mathcal{R}'$ of inference rule schemas that is sound and complete for $\mathcal{S}$, then $\mathcal{R}'$ specifies a ground axiomatization for $\mathcal{S}$ that is both recursive and k-ary for some k. Two results are demonstrated in this section: (1) There is no recursive axiomatization for finite implication

of fd's and ind's, and (2) there is no k-ary axiomatization for finite implication of fd's and ind's. It is also known that there is no k-ary axiomatization for unrestricted implication of fd's and ind's. The intuitive conclusion is that the family of fd's and ind's does not have a "finite axiomatization" for finite implication or for unrestricted implication.

To establish the framework and some notation, we assume temporarily that we are dealing with a family $\mathcal{F}$ of database instances over a fixed database schema $\mathbf{R} = \{R_1, \ldots, R_n\}$. Typically, $\mathcal{F}$ will be the set of all finite instances over $\mathbf{R}$, or the set of all (finite or infinite) instances over $\mathbf{R}$. All the notions that are defined are *with respect to* $\mathcal{F}$. Let $\mathcal{S}$ be a family of dependencies over $\mathbf{R}$. (At present, $\mathcal{S}$ would be the set of fd's and ind's over $\mathbf{R}$.) Logical implication $\models$ among dependencies in $\mathcal{S}$ is defined with respect to $\mathcal{F}$ in the natural manner. In particular, $\models_{\text{unr}}$ and $\models_{\text{fin}}$ are obtained by letting $\mathcal{F}$ be the set of unrestricted or finite instances.

A (*ground*) *inference rule* over $\mathcal{S}$ is an expression of the form

$$\rho = \text{if } S \text{ then } s,$$

where $S \subseteq \mathcal{S}$ and $s \in \mathcal{S}$.

Let $\mathcal{R}$ be a set of rules over $\mathbf{R}$. Then $\mathcal{R}$ is *sound* if each rule in $\mathcal{R}$ is sound. Let $\Sigma \cup \{\sigma\} \subseteq \mathcal{S}$ be a set of dependencies over $\mathbf{R}$. A *proof* of σ from Σ using $\mathcal{R}$ is a finite sequence $\sigma_1, \ldots, \sigma_n = \sigma$ such that for each $i \in [1, n]$, either (1) $\sigma_i \in \Sigma$, or (2) for some rule 'if S then s' in $\mathcal{R}$, $\sigma_i = s$ and $S \subseteq \{\sigma_1, \ldots, \sigma_{i-1}\}$. We write $\Sigma \vdash_{\mathcal{R}} \sigma$ (or $\Sigma \vdash \sigma$ if $\mathcal{R}$ is understood) if there is a proof of σ from Σ using $\mathcal{R}$. Clearly, if each rule in $\mathcal{R}$ is sound, then $\Sigma \vdash \sigma$ implies $\Sigma \models \sigma$. The set $\mathcal{R}$ is *complete* if for each pair (Σ, σ), $\Sigma \models \sigma$ implies $\Sigma \vdash_{\mathcal{R}} \sigma$. A (*sound and complete*) *axiomatization* for logical implication is a set $\mathcal{R}$ of rules that is sound and complete.

The aforementioned notions are now generalized to permit all schemas $\mathbf{R}$. In particular, we consider a set $\mathcal{R}$ of rules that is a union $\cup\{\mathcal{R}_{\mathbf{R}} \mid \mathbf{R} \text{ is a schema}\}$. The notions of sound, proof, etc.can be generalized in the natural fashion.

Note that with the preceding definition, every set $\mathcal{S}$ of dependencies has a sound and complete axiomatization. This is provided by the set $\mathcal{R}$ of all rules of the form

$$\text{if } S \text{ then } s,$$

where $S \models s$. Clearly, such trivial axiomatizations hold no interest. In particular, they are not necessarily effective (i.e., one may not be able to tell if a rule is in $\mathcal{R}$, so one may not be able to construct proofs that can be checked). It is thus natural to restrict $\mathcal{R}$ to be recursive.

We now present the first result of this section, which will imply that there is no recursive axiomatization for finite implication of fd's and ind's. In this result we assume that the dependencies in $\mathcal{S}$ are sentences in first-order logic.

Proposition 9.3.1 Let $\mathcal{S}$ be a class of dependencies. If $\mathcal{S}$ has a recursive axiomatization for finite implications, then finite implication is decidable for $\mathcal{S}$.

Crux Suppose that $\mathcal{S}$ has a recursive axiomatization. Consider the set

$$\mathit{Implic} = \{(S, s) \mid S \subseteq \mathcal{S}, s \in \mathcal{S}, \text{ and } S \models_{\text{fin}} s\}.$$

First note that the set *Implic* is r.e.; indeed, let $\mathcal{R}$ be a recursive axiomatization for $\mathcal{S}$. One can effectively enumerate all proofs of implication that use rules in $\mathcal{R}$. This allows one to enumerate *Implic* effectively. Thus *Implic* is r.e. We argue next that *Implic* is also co-r.e. To conclude that a pair (S, s) is not in *Implic*, it is sufficient to exhibit a finite instance satisfying S and violating s. To enumerate all pairs (S, s) not in *Implic*, one proceeds as follows. The set of *all* pairs (S, s) is clearly r.e., as is the set of all instances over a fixed schema. Repeat for all positive integers n the following. Enumerate the first n pairs (S, s) and the first n instances. For each (S, s) among the n, check whether one of the n instances is a counterexample to the implication $S \models s$, in which case output (S, s). Clearly, this procedure enumerates the complement of *Implic*, so *Implic* is co-r.e. Because it is both r.e. and co-r.e., *Implic* is recursive, so there is an algorithm testing whether (S, s) is in *Implic*. ■

It follows that there is no recursive axiomatization for finite implication of fd's and ind's. [To see this, note that by Theorem 9.2.4, logical implication for fd's and ind's is undecidable. By Proposition 9.3.1, it follows that there can be no finite axiomatization for fd's and ind's.] Because implication for jd's is decidable (Theorem 8.4.12), but there is no axiomatization for them (Theorem 8.3.4), the converse of the preceding proposition does not hold.

Speaking intuitively, the preceding development implies that there is no finite set of inference rule schemas that is sound and complete for finite implication of fd's and ind's. However, the proof is rather indirect. Furthermore, the approach cannot be used in connection with unrestricted implication, nor with classes of dependencies for which finite implication is decidable (see Exercise 9.9). The notion of k-ary axiomatization developed now shall overcome these objections.

A rule 'if S then s' is *k-ary* for some $k \geq 0$ if $|S| = k$. An axiomatization $\mathcal{R}$ is *k-ary* if each rule in $\mathcal{R}$ is l-ary for some $l \leq k$. For example, the instantiations of rules FD1 and IND1 are 0-ary, those of rules FD2 and IND2 are 1-ary, and those of FD3 and IND3 are 2-ary. Theorem 9.3.3 below shows that there is no k-ary axiomatization for finite implication of fd's and ind's.

We now turn to an analog in terms of logical implication of k-ary axiomatizability. Again let $\mathcal{S}$ be a set of dependencies over $\mathbf{R}$, and let $\mathcal{F}$ be a family of instances over $\mathbf{R}$. Let $k \geq 0$. A set $\Gamma \subseteq \mathcal{S}$ is:

closed under implication with respect to $\mathcal{S}$ if $\sigma \in \Gamma$ whenever

$$\text{(a) } \sigma \in \mathcal{S} \text{ and (b) } \Gamma \models \sigma$$

closed under k-ary implication with respect to $\mathcal{S}$ if $\sigma \in \Gamma$ whenever

$$\text{(a) } \sigma \in \mathcal{S}, \text{ and for some } \Sigma \subseteq \Gamma, \ (b_1)\ \Sigma \models \sigma \text{ and } (b_2)\ |\Sigma| \leq k.$$

Clearly, if Γ is closed under implication, then it is closed under k-ary implication for each

$k \geq 0$, and if Γ is closed under k-ary implication, then it is closed under k'-ary implication for each $k' \leq k$.

Proposition 9.3.2 Let $\mathbf{R}$ be a database schema, $\mathcal{S}$ a set of dependencies over $\mathbf{R}$, and $k \geq 0$. Then there is a k-ary axiomatization for $\mathcal{S}$ iff whenever $\Gamma \subseteq \mathcal{S}$ is closed under k-ary implication, then Γ is closed under implication.

Proof Suppose that there is a k-ary axiomatization for $\mathcal{S}$, and let $\Gamma \subseteq \mathcal{S}$ be closed under k-ary implication. Suppose further that $\Gamma \models \sigma$ for some $\sigma \in \mathcal{S}$. Let $\sigma_1, \ldots, \sigma_n$ be a proof of σ from Γ using $\mathcal{R}$. Using the fact that $\mathcal{R}$ is k-ary and that Γ is closed under k-ary implication, a straightforward induction shows that $\sigma_i \in \Gamma$ for $i \in [1, n]$.

Suppose now that for each $\Gamma \subseteq \mathcal{S}$, if Γ is closed under k-ary implication, then Γ is closed under implication. Set

$$\mathcal{R} = \{\text{`if } S \text{ then } s\text{'} \mid S \subseteq \mathcal{S}, s \in \mathcal{S}, |S| \leq k, \text{ and } S \models s\}.$$

To see that $\mathcal{R}$ is complete, suppose that $\Gamma \models \sigma$. Consider the set $\Gamma^* = \{\gamma \mid \Gamma \vdash_{\mathcal{R}} \gamma\}$. From the construction of $\mathcal{R}$, Γ^* is closed under k-ary implication. By assumption it is closed under implication, and so $\Gamma \vdash_{\mathcal{R}} \sigma$ as desired. ■

In the following, we consider finite implication, so $\mathcal{F}$ is the set of finite instances.

Theorem 9.3.3 For no k does there exist a k-ary sound and complete axiomatization for finite implication of fd's and ind's taken together. More specifically, for each k there is a schema $\mathbf{R}$ for which there is no k-ary sound and complete axiomatization for finite implication of fd's and ind's over $\mathbf{R}$.

Proof Let $k \geq 0$ be fixed. Let $\mathbf{R} = \{R_0, \ldots, R_k\}$ be a database schema where $sort(R_i) = \{A, B\}$ for each $i \in [0, k]$. In the remainder of this proof, addition is always done modulo $k + 1$. The dependencies $\Sigma = \Sigma_a \cup \Sigma_b$ and σ are defined by

(a) $\Sigma_a = \{R_i : A \to B \mid i \in [0, k]\}$;
(b) $\Sigma_b = \{R_i[A] \subseteq R_{i+1}[B] \mid i \in [0, k]\}$; and
(c) $\sigma = R_0[B] \subseteq R_k[A]$.

Let Γ be the union of Σ with all fd's and ind's that are tautologies (i.e., that are satisfied by all finite instances over $\mathbf{R}$).

In the remainder of the proof, it is shown that (1) Γ is not closed under finite implication, but (2) Γ is closed under k-ary finite implication. Proposition 9.3.2 will then imply that the family of fd's and ind's has no k-ary sound and complete axiomatization for $\mathbf{R}$.

First observe that Γ does not contain σ, so to show that Γ is not closed under finite implication, it suffices to demonstrate that $\Sigma \models_{\text{fin}} \sigma$. Let $\mathbf{I}$ be a finite instance of $\mathbf{R}$ that satisfies Σ. By the ind's of Σ, $|\mathbf{I}(R_i)[A]| \leq |\mathbf{I}(R_{i+1})[B]|$ for each $i \in [0, k]$, and by the fd's of Σ, $|\mathbf{I}(R_i)[B]| \leq |\mathbf{I}(R_i)[A]|$ for each $i \in [0, k]$. From this we obtain

$$\begin{aligned}|\mathbf{I}(R_0)[A]| &\le |\mathbf{I}(R_1)[B]| \le |\mathbf{I}(R_1)[A]| \\ &\le \quad \cdots \\ &\le |\mathbf{I}(R_k)[B]| \le |\mathbf{I}(R_k)[A]| \le |\mathbf{I}(R_0)[B]| \le |\mathbf{I}(R_0)[A]|.\end{aligned}$$

In particular, $|\mathbf{I}(R_k)[A]| = |\mathbf{I}(R_0)[B]|$. Since $\mathbf{I}$ is finite and we have $\mathbf{I}(R_k)[A] \subseteq \mathbf{I}(R_0)[B]$ and $|\mathbf{I}(R_k)[A]| = |\mathbf{I}(R_0)[B]|$, it follows that $\mathbf{I}(R_0)[B] \subseteq \mathbf{I}(R_k)[A]$ as desired.

We now show that Γ is closed under k-ary finite implication. Suppose that $\Delta \subseteq \Gamma$ has no more than k elements ($|\Delta| \le k$). It must be shown that if γ is an fd or ind and $\Delta \models_{\text{fin}} \gamma$, then $\gamma \in \Gamma$. Because Σ contains $k+1$ ind's, any subset Δ of Γ that has no more than k members must omit some ind δ of Σ. We shall exhibit an instance $\mathbf{I}$ such that $\mathbf{I} \models \gamma$ iff $\gamma \in \Gamma - \{\delta\}$. (Thus $\mathbf{I}$ will be an Armstrong instance for $\Gamma - \{\delta\}$.) It will then follow that $\Gamma - \{\delta\}$ is closed under finite implication. Because $\Delta \subseteq \Gamma - \{\delta\}$, this will imply that for each fd or ind γ, if $\Delta \models_{\text{fin}} \gamma$, then $\Gamma - \{\delta\} \models_{\text{fin}} \gamma$, so $\gamma \in \Gamma$.

Because Σ is symmetric with regard to ind's, we can assume without loss of generality that δ is the ind $R_k[A] \subseteq R_0[B]$. Assuming that $\mathbf{N} \times \mathbf{N}$ is contained in the underlying domain, define $\mathbf{I}$ so that

$$\mathbf{I}(R_0) = \{\langle(0,0),(0,k+1)\rangle,\ \langle(1,0),(1,k+1)\rangle,\ \langle(2,0),(1,k+1)\rangle\}$$

and for each $i \in [1,k]$,

$$\begin{aligned}\mathbf{I}(R_i) = \{&\langle(0,i),(0,i-1)\rangle,\ \langle(1,i),(1,i-1)\rangle,\ldots, \\ &\langle(2i+1,i),(2i+1,i-1)\rangle,\ \langle(2i+2,i),(2i+1,i-1)\rangle\}.\end{aligned}$$

Figure 9.2 shows $\mathbf{I}$ for the case $k = 3$.

We now show for each fd and ind γ over $\mathbf{R}$ that $\mathbf{I} \models \gamma$ iff $\gamma \in \Gamma - \delta$. Three cases arise:

1. γ is a tautology. Then this clearly holds.
2. γ is an fd that is not a tautology. Then γ is equivalent to one of the following for some $i \in [0,k]$:

$$\begin{aligned}&R_i : A \to B, \quad && R_i : B \to A, \\ &R_i : \emptyset \to A, && R_i : \emptyset \to B, \\ &\text{or} && R_i : \emptyset \to AB.\end{aligned}$$

   If γ is $R_i : A \to B$, then $\gamma \in \Gamma$ and clearly $\mathbf{I} \models \gamma$. In the other cases, $\gamma \notin \Gamma$ and $\mathbf{I} \not\models \gamma$.
3. γ is an ind that is not a tautology. Considering now which ind's $\mathbf{I}$ satisfies, note that the only pairs of nondisjoint columns of relations in $\mathbf{I}$ are

$$\begin{aligned}&\mathbf{I}(R_0)[A], \mathbf{I}(R_1)[B]; \\ &\mathbf{I}(R_1)[A], \mathbf{I}(R_2)[B]; \quad \ldots; \\ &\mathbf{I}(R_{k-1})[A], \mathbf{I}(R_k)[B].\end{aligned}$$

   Furthermore, $\mathbf{I} \not\models R_{i+1}[B] \subseteq R_i[A]$ for each $i \in [0,k]$; and $\mathbf{I} \models R_i[A] \subseteq R_{i+1}[B]$.

This implies that $\mathbf{I} \models \gamma$ iff $\gamma \in \Gamma - \{\delta\}$, as desired. ■

$\mathbf{I}(R_0)$

A	B
(0,0)	(0,4)
(1,0)	(1,4)
(2,0)	(1,4)

$\mathbf{I}(R_1)$

A	B
(0,1)	(0,0)
(1,1)	(1,0)
(2,1)	(2,0)
(3,1)	(3,0)
(4,1)	(3,0)

$\mathbf{I}(R_2)$

A	B
(0,2)	(0,1)
(1,2)	(1,1)
(2,2)	(2,1)
(3,2)	(3,1)
(4,2)	(4,1)
(5,2)	(5,1)
(6,2)	(5,1)

$\mathbf{I}(R_3)$

A	B
(0,3)	(0,2)
(1,3)	(1,2)
(2,3)	(2,2)
(3,3)	(3,2)
(4,3)	(4,2)
(5,3)	(5,2)
(6,3)	(6,2)
(7,3)	(7,2)
(8,3)	(7,2)

Figure 9.2: An Armstrong relation for $\Gamma - \delta$

In the proof of the preceding theorem all relations used are binary, and all fd's and ind's are *unary*, in the sense that at most one attribute appears on either side of each dependency. In proofs that there is no k-ary axiomatization for unrestricted implication of fd's and ind's, some of the ind's used involve at least two attributes on each side. This cannot be improved to unary ind's, because there is a 2-ary sound and complete axiomatization for unrestricted implication of unary ind's and arbitrary fd's (see Exercise 9.18).

9.4 Restricted Kinds of Inclusion Dependency

This section explores two restrictions on ind's for which several positive results have been obtained. The first one focuses on sets of ind's that are acyclic in a natural sense, and the second restricts the ind's to having only one attribute on either side. The restricted dependencies are important because they are sufficient to model many natural relationships, such as those captured by semantic models (see Chapter 11). These include subtype relationships of the kind "every student is also a person."

This section also presents a generalization of the chase that incorporates ind's. Because ind's are embedded, chasing in this context may lead to infinite chasing sequences. In the context of acyclic sets of ind's, however, the chasing sequences are guaranteed to terminate. The study of infinite chasing sequences will be taken up in earnest in Chapter 10.

Ind's and the Chase

Because ind's may involve more than one relation, the formal notation of the chase must be extended. Suppose now that $\mathbf{R}$ is a database schema, and let $q = (\mathbf{T}, t)$ be a tableau query over $\mathbf{R}$. The fd and jd rules are generalized to this context in the natural fashion.

We first present an example and then describe the rule that is used for ind's.

Example 9.4.1 Consider the database schemas consisting of two relation schemas P, Q with $sort(P) = ABC$, $sort(Q) = DEF$, the dependencies

$$Q[DE] \subseteq P[AB] \qquad \text{and} \qquad P : A \to B,$$

and the tableau $\mathbf{T}$ shown in Fig. 9.3. Consider $\mathbf{T}_1$ and $\mathbf{T}_2$ in the same figure. The tableau $\mathbf{T}_1$ is obtained by applying to $\mathbf{T}$ the ind rule given after this example. The intuition is that the tuples $\langle x, y_i \rangle$ should also be in the P-relation because of the ind. Then $\mathbf{T}_2$ is obtained by applying the fd rule. Tableau minimization can be applied to obtain $\mathbf{T}_3$.

The following rule is used for ind's.

ind rule: Let $\sigma = R[X] \subseteq S[Y]$ be an ind, let $u \in \mathbf{T}(R)$, and suppose that there is no free tuple $v \in \mathbf{T}(S)$ such that $v[Y] = u[X]$. In this case, we say that σ is *applicable* to $R(u)$. Let w be a free tuple over S such that $w[Y] = u[X]$ and w has distinct new variables in all coordinates of $sort(S) - Y$ that are greater than all variables occurring in q. Then "the" result of applying σ to $R(u)$ is $(\mathbf{T}', t)$, where

- $\mathbf{T}'(P) = \mathbf{T}(P)$ for each relation name $P \in \mathbf{R} - \{S\}$, and
- $\mathbf{T}'(S) = \mathbf{T}(S) \cup \{w\}$.

For a tableau query q and a set Σ of ind's, it is possible that two terminal chasing sequences end with nonisomorphic tableau queries, that there are no finite terminal chasing sequences, or that there are both finite terminal chasing sequences and infinite chasing sequences (see Exercise 9.12). General approaches to resolving this problem will be considered in Chapter 10. In the present discussion, we focus on acyclic sets of ind's, for which the chase always terminates after a finite number of steps.

Acyclic Inclusion Dependencies

Definition 9.4.2 A family Σ of ind's over $\mathbf{R}$ is *acyclic* if there is no sequence $R_i[X_i] \subseteq S_i[Y_i]$ $(i \in [1, n])$ of ind's in Σ where for $i \in [1, n]$, $R_{i+1} = S_i$ for $i \in [1, n-1]$, and $R_1 = S_n$. A family Σ of dependencies has *acyclic* ind's if the set of ind's in Σ is acyclic.

The following is easily verified (see Exercise 9.14):

Proposition 9.4.3 Let q be a tableau query and Σ a set of fd's, jd's, and acyclic ind's over $\mathbf{R}$. Then each chasing sequence of q by Σ terminates after an exponentially bounded number of steps.

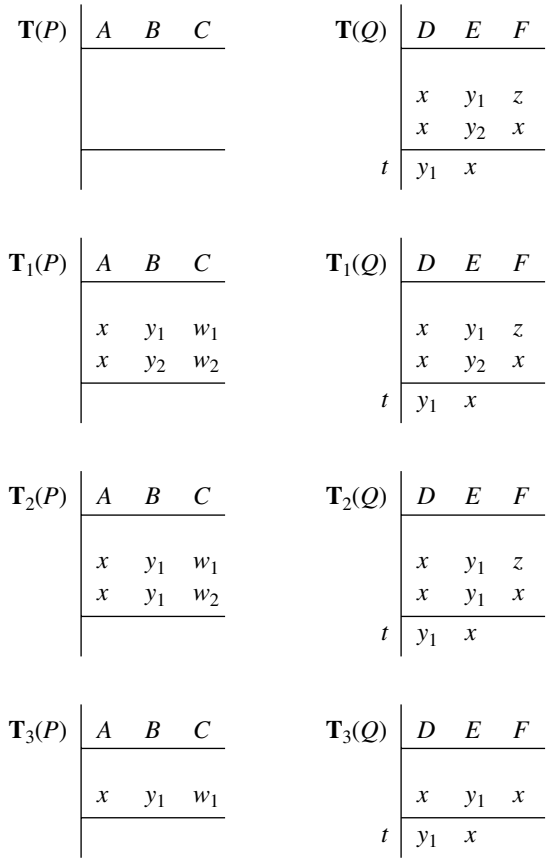


Figure 9.3: Chasing with ind's

For each tableau query q and set Σ of fd's, jd's, and acyclic ind's, let $chase(q, \Sigma)$ denote the result of some arbitrary chasing sequence of q by Σ. (One can easily come up with some syntactic strategy for arbitrarily choosing this sequence.)

Using an analog to Lemma 8.4.3, one obtains the following result on tableau query containment (an analog to Theorem 8.4.8).

THEOREM 9.4.4 Let q, q' be tableau queries and Σ a set of fd's, jd's, and acyclic ind's over $\mathbf{R}$. Then $q \subseteq_\Sigma q'$ iff $chase(q, \Sigma) \subseteq chase(q', \Sigma)$.

Next we consider the application of the chase to implication of dependencies. For database schema $\mathbf{R}$ and ind $\sigma = R[X] \subseteq S[Y]$ over $\mathbf{R}$, the *tableau query* of σ is $q_\sigma = (\{R(u_\sigma)\}, \langle u_\sigma \rangle)$, where u_σ is a free tuple all of whose entries are distinct. For example, given $R[ABCD]$, $S[EFG]$, and $\sigma = R[BC] \subseteq S[GE]$, $q_\sigma = (\{R(x_1, x_2, x_3, x_4)\}, \langle x_1, x_2,$

$x_3, x_4\rangle$). In analogy with Theorem 8.4.12, we have the following for fd's, jd's, and acyclic ind's.

THEOREM 9.4.5 Let Σ be a set of fd's, jd's, and acyclic ind's over database schema $\mathbf{R}$ and let $\mathbf{T}$ be the tableau in $chase(q_\sigma, \Sigma)$. Then $\Sigma \models_{\text{unr}} \sigma$ iff

(a) For fd or jd σ over $\mathbf{R}$, $\mathbf{T}$ satisfies the conditions of Theorem 8.4.12.
(b) For ind $\sigma = R[X] \in S[Y]$, $u_\sigma[X] \in \mathbf{T}(S)[Y]$.

This yields the following:

COROLLARY 9.4.6 Finite and unrestricted implication for sets of fd's, jd's, and acyclic ind's coincide and are decidable in exponential time.

An improvement of the complexity here seems unlikely, because implication of an ind by an acyclic set of ind's is NP-complete (see Exercise 9.14).

Unary Inclusion Dependencies

A *unary inclusion dependency* (uind) is an ind in which exactly one attribute appears on each side. The uind's arise frequently in relation schemas in which certain columns range over values that correspond to entity types (e.g., if *SS#* is a key for the *Person* relation and is also used to identify people in the *Employee* relation).

As with arbitrary ind's, unrestricted and finite implication do not coincide for fd's and uind's (proof of Theorem 9.2.2). However, both forms of implication are decidable in polynomial time. In this section, the focus is on finite implication. We present a sound and complete axiomatization for finite implication of fd's and uind's (but in agreement with Theorem 9.3.3, it is not k-ary for any k).

For uind's considered in isolation, the inference rules for ind's are specialized to yield the following two rules, which are sound and complete for (unrestricted and finite) implication. Here A, B, and C range over attributes and R, S, and T over relation names:

UIND1: (reflexivity) $R[A] \subseteq R[A]$.

UIND2: (transitivity) If $R[A] \subseteq S[B]$ and $S[B] \subseteq T[C]$, then $R[A] \subseteq T[C]$.

To capture the interaction of fd's and uind's in the finite case, the following family of rules is used:

C*:* (cycle rules) For each positive integer n,

$$\text{if} \begin{cases} R_1 : A_1 \to B_1, \\ R_2[A_2] \subseteq R_1[B_1], \\ \ldots, \\ R_n : A_n \to B_n, \text{ and} \\ R_1[A_1] \subseteq R_n[B_n] \end{cases} \quad \text{then} \begin{cases} R_1 : B_1 \to A_1, \\ R_1[B_1] \subseteq R_2[A_2], \\ \ldots, \\ R_n : B_n \to A_n, \text{ and} \\ R_n[B_n] \subseteq R_1[A_1]. \end{cases}$$

The soundness of this family of rules follows from a straightforward cardinality argument. More generally, we have the following (see Exercise 9.16):

THEOREM 9.4.7 The set {FD1, FD2, FD3, UIND1, UIND2} along with the cycle rules (C) is sound and complete for finite implication of fd's and uind's. Furthermore, finite implication is decidable in polynomial time.

Bibliographic Notes

Inclusion dependency is based on the notion of referential integrity, which was known to the broader database community during the 1970s (see, e.g., [Dat81]). A seminal paper on the theory of ind's is [CFP84], in which inference rules for ind's are presented and the nonaxiomatizability of both finite and unrestricted implication for fd's and ind's is demonstrated. A non-k-ary sound and complete set of inference rules for finite implication of fd's and ind's is presented in [Mit83b]. Another seminal paper is [JK84b], which also observed the distinction between finite and unrestricted implication for fd's and ind's, generalized the chase to incorporate fd's and ind's, and used this to characterize containment between conjunctive queries. Related work is reported in [LMG83].

Undecidability of (finite) implication for fd's and ind's taken together was shown independently by [CV85] and [Mit83a]. The proof of Theorem 9.2.4 is taken from [CV85]. (The undecidability of the word problem for monoids is from [Pos47], and of the word problem for finite monoids is from [Gur66].)

Acyclic ind's were introduced in [Sci86]. Complexity results for acyclic ind's include that implication for acyclic ind's alone is NP-complete [CK86], and implication for fd's and acyclic ind's has an exponential lower bound [CK85].

Given the PSPACE complexity of implication for ind's and the negative results in connection with fd's, unary ind's emerged as a more tractable form of inclusion dependency. The decision problems for finite and unrestricted implication for uind's and fd's taken together, although not coextensive, both lie in polynomial time [CKV90]. This extensive paper also develops axiomatizations of both finite and unrestricted logical implication for unary ind's and fd's considered together, and develops results for uind's with some of the more general dependencies studied in Chapter 10.

Typed ind's are studied in [CK86]. In addition to using traditional techniques from dependency theory, such as chasing, this work develops tools for analyzing ind's using equational theories.

Ind's in connection with other dependencies are also studied in [CV83].

Exercises

Exercise 9.1 Complete the proof of Proposition 9.1.5.

Exercise 9.2 Complete the proof of Theorem 9.1.7.

Exercise 9.3 [CFP84] (In this exercise, by a slight abuse of notation, we allow fd's with sequences rather than sets of attributes.) Demonstrate the following:

(a) If $|\vec{A}| = |\vec{B}|$, then $\{R[\vec{A}\vec{C}] \subseteq S[\vec{B}\vec{D}], S : \vec{B} \to \vec{D}\} \models_{\text{unr}} R : \vec{A} \to \vec{C}$.

(b) If $|\vec{A}| = |\vec{B}|$, then $\{R[\vec{A}\vec{C}] \subseteq S[\vec{B}\vec{D}], R[\vec{A}\vec{E}] \subseteq S[\vec{B}\vec{F}], S : \vec{B} \to \vec{D}\} \models_{\text{unr}} R[\vec{A}\vec{C}\vec{E}] \subseteq S[\vec{B}\vec{D}\vec{F}]$.

(c) Suppose that $|\vec{A}| = |\vec{B}|$; $\Sigma = \{R[\vec{A}\vec{C}] \subseteq S[\vec{B}\vec{D}], R[\vec{A}\vec{E}] \subseteq S[\vec{B}\vec{D}], S : \vec{B} \to \vec{D}\}$; and $\mathbf{I} \models \Sigma$. Then $u[\vec{C}] = u[\vec{E}]$ for each $u \in \mathbf{I}(R)$.

Exercise 9.4 As defined in the text, we require in ind $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ that the A_i's and B_i's are distinct. A *repeats-permitted inclusion dependency* (rind) is defined as was inclusion dependency, except that repeats are permitted in the attribute sequences on both the left- and right-hand sides.

(a) Show that if Σ is a set of ind's, σ a rind, and $\Sigma \models_{\text{unr}} \sigma$, then σ is equivalent to an ind.

(b) Exhibit a set Σ of ind's and fd's such that $\Sigma \models_{\text{unr}} R[AB] \subseteq S[CC]$. Do the same for $R[AA] \subseteq R[BC]$.

♠ (c) [Mit83a] Consider the rules

IND4: If $R[A_1A_2] \subseteq S[BB]$ and $R[\vec{C}] \subseteq T[\vec{D}]$, then $R[\vec{C}'] \subseteq T[\vec{D}]$, where $\vec{C}'$ is obtained from $\vec{C}$ by replacing one or more occurrences of A_2 by A_1.

IND5: If $R[A_1A_2] \subseteq S[BB]$ and $T[\vec{C}] \subseteq R[\vec{D}]$, then $T[\vec{C}] \subseteq R[\vec{D}']$, where $\vec{D}'$ is obtained from $\vec{D}$ by replacing one or more occurrences of A_2 by A_1.

Prove that the inference rules {IND1, IND2, IND3, IND4, IND5} are sound and complete for finite implication of sets of rind's.

(d) Prove that unrestricted and finite implication coincide for rind's.

(e) A *left-repeats-permitted inclusion dependency* (l-rind) is a rind for which there are no repeats on the right-hand side. Given a set $\Sigma \cup \{\sigma\}$ of l-rind's over $\mathbf{R}$, describe how to construct a schema $\mathbf{R}'$ and ind's $\Sigma' \cup \{\sigma'\}$ over $\mathbf{R}'$ such that $\Sigma \models \sigma$ iff $\Sigma' \models \sigma'$ and $\Sigma \models_{\text{fin}} \sigma$ iff $\Sigma' \models_{\text{fin}} \sigma'$.

(f) Do the same as in part (e), except for arbitrary rind's.

Exercise 9.5 [CV85] Prove part (b) of Theorem 9.2.4. *Hint:* In the proof of part (a), extend the schema of R to include new attributes $A_{\alpha'}$, $A_{\beta'}$, and $A_{y'}$; add dependencies $A_y \to A_{y'}$, $R[A_\alpha, A_{\alpha'}] \subseteq R[A_y, A_{y'}]$, $R[A_\beta, A_{\beta'}] \subseteq R[A_y, A_{y'}]$; and use $A_{\alpha'} \to A_{\beta'}$ as σ.

Exercise 9.6

(a) Develop an alternative proof of Theorem 9.3.3 in which δ is an fd rather than an ind.

(b) In the proof of Theorem 9.3.3 for finite implication, the dependency σ used is an ind. Using the same set Σ, find an fd that can be used in place of σ in the proof.

Exercise 9.7 Prove that there is no k for which there is a k-ary sound and complete axiomatization for finite implication of fd's, jd's, and ind's.

★ **Exercise 9.8** [SW82] Prove that there is no k-ary sound and complete set of inference rules for finite implication of emvd's.

Exercise 9.9 Recall the notion of sort-set dependency (ssd) from Exercise 8.32.

(a) Prove that finite and unrestricted implication coincide for fd's and ssd's considered together. Conclude that implication for fd's and ssd's is decidable.

★(b) [GH86] Prove that there is no k-ary sound and complete set of inference rules for finite implication of fd's (key dependencies) and ssd's taken together.

Exercise 9.10

(a) [CFP84] A set of ind's is *bounded* by k if each ind in the set has at most k attributes on the left-hand side and on the right-hand side. Show that logical implication for bounded sets of ind's is decidable in polynomial time.

(b) [CV83] An ind is *typed* if it has the form $R[\vec{A}\,] \subseteq S[\vec{A}\,]$. Exhibit a polynomial time algorithm for deciding logical implication between typed ind's.

Exercise 9.11 Suppose that some attribute domains may be finite.

(a) Show that {IND1, IND2, IND3} remains sound in the framework.

(b) Show that if one-element domains are permitted, then {IND1, IND2, IND3} is not complete.

(c) Show for each $n > 0$ that if all domains are required to have at least n elements, then {IND1, IND2, IND3} is not complete.

Exercise 9.12 Suppose that no restrictions are put on the order of application of ind rules in chasing sequences.

(a) Exhibit a tableau query q and a set Σ of ind's and two terminal chasing sequences of q by Σ that end with nonisomorphic tableau queries.

(b) Exhibit a tableau query q and a set Σ of ind's, a terminal chasing sequence of q by Σ, and an infinite chasing sequence of q by Σ.

(c) Exhibit a tableau query q and a set Σ of ind's such that q has no finite terminal chasing sequence by Σ.

♠**Exercise 9.13** [JK84b] Recall that for tableau queries q and q' and a set Σ of fd's and jd's over R, $q \subseteq_\Sigma q'$ if for each instance I that satisfies Σ, $q(I) \subseteq q'(I)$. In the context of ind's, this containment relationship may depend on whether infinite instances are permitted or not. For tableau queries q, q' and a set Σ of dependencies over $\mathbf{R}$, we write $q \subseteq_{\Sigma,\text{fin}} q'$ ($q \subseteq_{\Sigma,\text{unr}} q'$) if $q(\mathbf{I}) \subseteq q'(\mathbf{I})$ for each finite (unrestricted) instance $\mathbf{I}$ that satisfies Σ.

(a) Show that if Σ is a set of fd's and jd's, then $\subseteq_{\Sigma,\text{fin}}$ and $\subseteq_{\Sigma,\text{unr}}$ coincide.

(b) Exhibit a set Σ of fd's and ind's and tableau queries q, q' such that $q \subseteq_{\Sigma,\text{fin}} q'$ but $q \not\subseteq_{\Sigma,\text{unr}} q'$.

Exercise 9.14

(a) Prove Proposition 9.4.3.

(b) Prove Theorem 9.4.4.

(c) Let q be a tableau query and Σ a set of fd's, jd's, and ind's over $\mathbf{R}$, where the set of ind's in Σ is acyclic; and suppose that q', q'' are the final tableaux of two terminal chasing sequences of q by Σ (where the order of rule application is not restricted). Prove that $q \equiv q'$.

(d) Prove Theorem 9.4.5.

(e) Prove Corollary 9.4.6.

Exercise 9.15

(a) Exhibit an acyclic set Σ of ind's and a tableau query q such that $chase(q, \Sigma)$ is exponential in the size of Σ and q.

(b) [CK86] Prove that implication of an ind by an acyclic set of ind's is NP-complete. *Hint:* Use a reduction from the problem of Permutation Generation [GJ79].

(c) [CK86] Recall from Exercise 9.10(b) that an ind is *typed* if it has the form $R[\vec{A}] \subseteq S[\vec{A}]$. Prove that implication of an ind by a set of fd's and an acyclic set of typed ind's is NP-hard. *Hint:* Use a reduction from 3-SAT.

♠ **Exercise 9.16** [CKV90] In this exercise you will prove Theorem 9.4.7. The exercise begins by focusing on the unirelational case; for notational convenience we omit the relation name from uind's in this context.

Given a set Σ of fd's and uind's over R, define $G(\Sigma)$ to be a multigraph with node set R and two colors of edges: a red edge from A to B if $A \to B \in \Sigma$, and a black edge from A to B is $B \subseteq A \in \Sigma$. If A and B have red (black) edges in both directions, replace them with an undirected red (black) edge.

(a) Suppose that Σ is closed under the inference rules. Prove that $G(\Sigma)$ has the following properties:
   1. Nodes have red (black) self-loops, and the red (black) subgraph of $G(\Sigma)$ is transitively closed.
   2. The subgraphs induced by the strongly connected components of $G(\Sigma)$ contain only undirected edges.
   3. In each strongly connected component, the red (black) subset of edges forms a collection of node disjoint cliques (the red and black partitions of nodes could be different).
   4. If $A_1 \ldots A_m \to B$ is an fd in Σ and $A_1, \ldots, A_m$ have common ancestor A in the red subgraph of $G(\Sigma)$, then $G(\Sigma)$ contains a red edge from A to B.

(b) Given a set Σ of fd's and uind's closed under the inference rules, use $G(\Sigma)$ to build counterexample instances that demonstrate that $\Sigma \not\vdash \sigma$ implies $\Sigma \not\models_{\text{fin}} \sigma$ for fd or uind σ.

(c) Use the rules to develop a polynomial time algorithm for inferring finite implication for a set of fd's and uind's.

(d) Generalize the preceding development to arbitrary database schemas.

Exercise 9.17

(a) Let $k > 1$ be an integer. Prove that there is a database schema $\mathbf{R}$ with at least one unary relation $R \in \mathbf{R}$, and a set Σ of fd's and ind's such that
   (i) for each $\mathbf{I} \models \Sigma$, $|\mathbf{I}(R)| = 0$ or $|\mathbf{I}(R)| = 1$ or $|\mathbf{I}(R)| \geq k$.
   (ii) for each $l \geq k$ there is an instance $\mathbf{I}_l \models \Sigma$ with $|\mathbf{I}(R)| = l$.

(b) Prove that this result cannot be strengthened so that condition (i) reads
   (i) (i′) for each $\mathbf{I} \models \Sigma$, $|\mathbf{I}(R)| = 0$ or $|\mathbf{I}(R)| = 1$ or $|\mathbf{I}(R)| = k$.

♠ **Exercise 9.18** [CKV90]

(a) Show that the set of inference rules containing {FD1, FD2, FD3, UIND1, UIND2} and
   FD-UIND1: If $\emptyset \to A$ and $R[B] \subseteq R[A]$, then $\emptyset \to B$.
   FD-UIND1: If $\emptyset \to A$ and $R[B] \subseteq R[A]$, then $R[A] \subseteq R[B]$.

is sound and complete for unrestricted logical implication of fd's and uind's over a single relation schema R.

(b) Generalize this result to arbitrary database schemas, under the assumption that in all instances, each relation is nonempty.

10 A Larger Perspective

Alice: *fd's, jd's, mvd's, ejd's, emvd's, ind's—it's all getting very confusing.*
Vittorio: *Wait! We'll use logic to unify it all.*
Sergio: *Yes! Logic will make everything crystal clear.*
Riccardo: *And we'll get a better understanding of dependencies that make sense.*

The dependencies studied in the previous chapters have a strong practical motivation and provide a good setting for studying two of the fundamental issues in dependency theory: deciding logical implication and constructing axiomatizations.

Several new dependencies were introduced in the late 1970s and early 1980s, sometimes motivated by practical examples and later motivated by a desire to understand fundamental theoretical properties of unirelational dependencies or to find axiomatizations for known classes of dependencies. This process culminated with a rather general perspective on dependencies stemming from mathematical logic: Almost all dependencies that have been introduced in the literature can be described as logical sentences having a simple structure, and further syntactic restrictions on that structure yield natural subclasses of dependencies. The purpose of this chapter is to introduce this general class of dependencies and its natural subclasses and to present important results and techniques obtained for them.

The general perspective is given in the first section, along with a simple application of logic to obtain the decidability of implication for a large class of dependencies. It turns out that the chase is an invaluable tool for analyzing implication; this is studied in the second section. Axiomatizations for important subclasses have been developed, again using the chase; this is the topic of the third section. We conclude the chapter with a provocative alternative view of dependencies stemming from relational algebra.

The classes of dependencies studied in this chapter include complex dependencies that would not generally arise in practice. Even if they did arise, they are so intricate that they would probably be unusable—it is unlikely that database administrators would bother to write them down or that software would be developed to use or enforce them. Nevertheless, it is important to repeat that the perspective and results discussed in this chapter have served the important function of providing a unified understanding of virtually all dependencies raised in the literature and, in particular, of providing insight into the boundaries between tractable and intractable problems in the area.

10.1 A Unifying Framework

The fundamental property of all of the dependencies introduced so far is that they essentially say, "The presence of some tuples in the instance implies the presence of certain other tuples in the instance, or implies that certain tuple components are equal." In the case of jd's and mvd's, the *new* tuples can be completely specified in terms of the *old* tuples, but for ind's this is not the case. In any case, all of the dependencies discussed so far can be expressed using first-order logic sentences of the form

$$(*) \qquad \forall x_1 \ldots \forall x_n \, [\, \varphi(x_1, \ldots, x_n) \rightarrow \exists z_1 \ldots \exists z_k \psi(y_1, \ldots, y_m) \,],$$

where $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_m\} - \{x_1, \ldots, x_n\}$, and where φ is a (possibly empty) conjunction of atoms and ψ a nonempty conjunction. In both φ and ψ, one finds *relation atoms* of the form $R(w_1, \ldots, w_l)$ and *equality atoms* of the form $w = w'$, where each of the $w, w', w_1, \ldots, w_l$ is a variable.

Because we generally focus on sets of dependencies, we make several simplifying assumptions before continuing (see Exercise 10.1a). These include that (1) we may eliminate equality atoms from φ without losing expressive power; and (2) we can also assume without loss of generality that no existentially quantified variable participates in an equality atom in ψ. Thus we define an *(embedded) dependency* to be a sentence of the foregoing form, where

1. φ is a conjunction of relation atoms using all of the variables $x_1, \ldots, x_n$;
2. ψ is a conjunction of atoms using all of the variables $z_1, \ldots, z_k$; and
3. there are no equality atoms in ψ involving existentially quantified variables.

A dependency is *unirelational* if at most one relation name is used, and it is *multirelational* otherwise. To simplify the presentation, the focus in this chapter is almost exclusively on unirelational dependencies. Thus, unless otherwise indicated, the dependencies considered here are unirelational.

We now present three fundamental classifications of dependencies.

Full versus embedded: A *full* dependency is a dependency that has no existential quantifiers.

Tuple generating versus equality generating: A *tuple-generating dependency* (tgd) is a dependency in which no equality atoms occur; an *equality-generating dependency* (egd) is a dependency for which the right-hand formula is a *single* equality atom.

Typed versus untyped: A dependency is *typed* if there is an assignment of variables to column positions such that (1) variables in relation atoms occur only in their assigned position, and (2) each equality atom involves a pair of variables assigned to the same position.

It is sometimes important to distinguish dependencies with a single atom in the right-hand formula. A dependency is *single head* if the right-hand formula involves a single atom; it is *multi-head* otherwise.

The following result is easily verified (Exercise 10.1b).

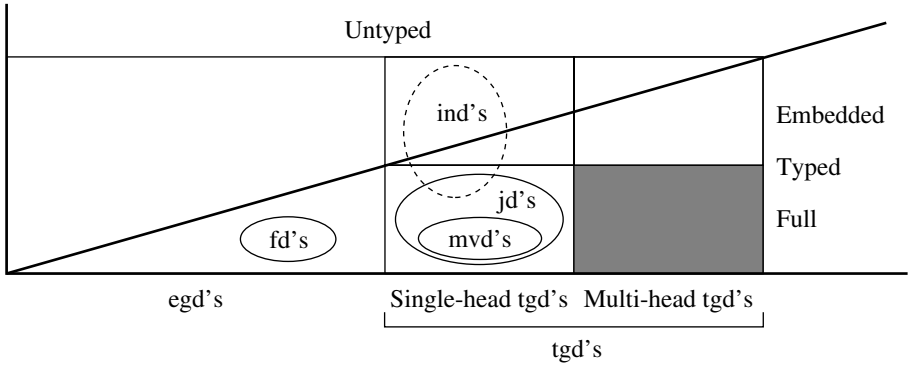


Figure 10.1: Dependencies

Proposition 10.1.1 Each (typed) dependency is equivalent to a set of (typed) egd's and tgd's.

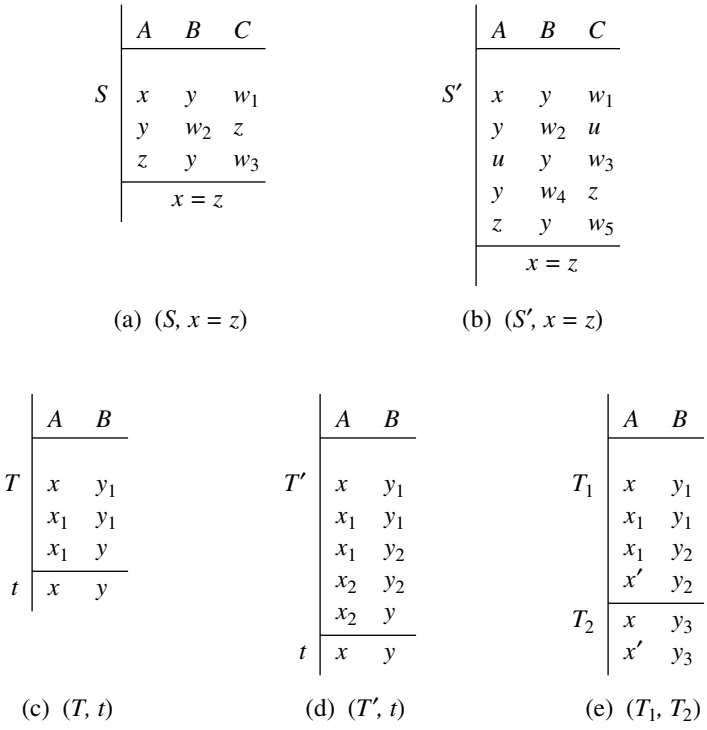
It is easy to classify the fd's, jd's, mvd's, ejd's, emvd's and ind's studied in Chapters 8 and 9 according to the aforementioned dimensions. All except the last are typed. During the late 1970s and early 1980s the class of typed dependencies was studied in depth. In many cases, the results obtained for dependencies and for typed dependencies are equivalent. However, for negative results the typed case sometimes requires more sophisticated proof techniques because it imposes more restrictions.

A classification of dependencies along the three axes is given in Fig. 10.1. The gray square at the lower right indicates that each full multihead tgd is equivalent to a set of single-head tgd's. The intersection of ind's and jd's stems from trivial dependencies. For example, $R[AB] \subseteq R[AB]$ and $\bowtie[AB]$ over relation $R(AB)$ are equivalent [and are syntactically the same when written in the form of $(*)$].

There is a strong relationship between dependencies and tableaux. Tableaux provide a convenient notation for expressing and working with dependencies. (As will be seen in Section 10.4, the family of typed dependencies can also be represented using a formalism based on algebraic expressions.) The tableau representation of two untyped egd's is shown in Figs. 10.2(a) and 10.2(b). These two egd's are equivalent. Note that all egd's can be expressed as a pair $(T, x = y)$, where T is a tableau and $x, y \in var(T)$. If $(T, x = y)$ is typed, unirelational, and x, y are in the A column of T, then this is referred to as an *A-egd*.

Parts (c) and (d) of Fig. 10.2 show two full tgd's that are equivalent. This is especially interesting because, considered as tableau queries, (T', t) properly contains (T, t) (see Exercise 10.4). As suggested earlier, each full tgd is equivalent to some set of full single-head tgd's. In the following, when considering full tgd's, we will assume that they are single head.

Part (e) of Fig. 10.2 shows a typed tgd that is not single head. To represent these within



(a) $(S, x = z)$ (b) $(S', x = z)$

(c) (T, t) (d) (T', t) (e) (T_1, T_2)

Figure 10.2: Five dependencies

the tableau notation, we use an ordered pair (T_1, T_2), where both T_1 and T_2 are tableaux. This tgd is not equivalent to any set of single-head tgd's (see Exercise 10.6b).

Finite versus Unrestricted Implication Revisited

We now reexamine the issues of finite versus unrestricted implication using the logical perspective on dependencies. Because all of these lie within first-order logic, $\models_{\text{fin}}$ is co-r.e. and $\models_{\text{unr}}$ is r.e. (see Chapter 2). Suppose that $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is a set of dependencies and $\{\sigma\}$ a dependency. Then $\Sigma \models_{\text{unr}} \sigma$ ($\Sigma \models_{\text{fin}} \sigma$) iff there is no unrestricted (finite) model of $\sigma_1 \wedge \cdots \wedge \sigma_n \wedge \neg\sigma$. If these are all full dependencies, then they can be rewritten in prenex normal form, where the quantifier prefix has the form $\exists^*\forall^*$. (Here each of the σ_i is universally quantified, and $\neg\sigma$ contributes the existential quantifier.) The family of sentences that have a quantifier prefix of this form (and no function symbols) is called the *initially extended Bernays-Schönfinkel class*, and it has been studied in the logic community since the 1920s. It is easily verified that finite and unrestricted satisfiability coincide for sentences in this class (Exercise 10.3). It follows that finite and unrestricted implication coincide for full dependencies and, as discussed in Chapter 9, it follows that implication is decidable.

On the other hand, because fd's and uind's are dependencies, we know from Theorem 9.2.4 that the two forms of implication do not coincide for (embedded) dependencies, and both are nonrecursive. Although not demonstrated here, these results have been extended to the family of embedded multivalued dependencies (emvd's).

To summarize:

THEOREM 10.1.2

1. For full dependencies, finite and unrestricted implication coincide and are decidable.
2. For (typed) dependencies, finite and unrestricted implication do not coincide and are both undecidable. In fact, this is true for embedded multivalued dependencies. In particular, finite implication is not r.e., and unrestricted implication is not co-r.e.

10.2 The Chase Revisited

As suggested by the close connection between dependencies and tableaux, chasing is an invaluable tool for characterizing logical implication for dependencies. In this section we first use chasing to develop a test for logical implication of arbitrary dependencies by full dependencies. We also present an application of the chase for determining how full dependencies are propagated to views. We conclude by extending the chase to work with embedded dependencies. In this discussion we focus almost entirely on typed dependencies, but it will be clear that the arguments can be modified to the untyped case.

Chasing with Full Dependencies

We first state without proof the natural generalization of chasing by fd's and jd's (Theorem 8.4.12) to full dependencies (see Exercise 10.8). In this context we begin either with a tableau T, or with an arbitrary tgd (T, T') or egd $(T, x = y)$. The notion of *applying* a full dependency to this is defined in the natural manner. Lemma 8.4.17 and the notation developed for it generalize naturally to this context, as does the following analog of Theorem 8.4.18:

THEOREM 10.2.1 If Σ is a set of full dependencies and T is a tableau (τ a dependency), then chasing $T(\tau)$ by Σ yields a unique finite result, denoted $chase(T, \Sigma)$ ($chase(\tau, \Sigma)$).

Logical implication of (full or embedded) dependencies by sets of full dependencies will now be characterized by a straightforward application of the techniques developed in Section 8.4 (see Exercise 10.8). A dependency τ is *trivial* if

(a) τ is an egd $(T, x = x)$; or

(b) τ is a tgd (T, T') and there is a substitution θ for T' such that $\theta(T') \subseteq T$ and θ is the identity on $var(T) \cap var(T')$.

Note that if τ is a *full* tgd, then (b) simply says that $T' \subseteq T$.

A dependency τ is a *tautology* for finite (unrestricted) instances if each finite (unrestricted) instance of appropriate type satisfies τ—that is, if $\emptyset \models_{\text{fin}} \tau$ ($\emptyset \models_{\text{unr}} \tau$). It is easily verified that a dependency is a tautology iff it is trivial.

The following now provides a simple test for implication by full typed dependencies:

THEOREM 10.2.2 Let Σ be a set of full typed dependencies and τ a typed dependency. Then $\Sigma \models \tau$ iff $chase(\tau, \Sigma)$ is trivial.

Recall that the chase relies on a total order $\leq$ on **var**. For egd $(T, x = y)$ we assume that $x < y$ and that these are the least and second to least variables appearing in the tableau; and for full tgd (T, t), $t(A)$ is least in $T(A)$ for each attribute A. Using this convention, we can obtain the following:

COROLLARY 10.2.3 Let Σ be a set of full typed dependencies.

(a) If $\tau = (T, x = y)$ is a typed egd, then $\Sigma \models \tau$ iff x and y are identical or $y \notin var(chase(T, \Sigma))$.

(b) If $\tau = (T, t)$ is a full typed tgd, then $\Sigma \models \tau$ iff $t \in chase(T, \Sigma)$.

Using the preceding results, it is straightforward to develop a deterministic exponential time algorithm for testing implication of full dependencies. It is also known that for both the typed and untyped cases, implication is complete in EXPTIME. (Note that, in contrast, logical implication for arbitrary sets of initially extended Bernays-Schöfinkel sentences is known to be complete in nondeterministic EXPTIME.)

Dependencies and Views

On a bit of a tangent, we now apply the chase to characterize the interaction of full dependencies and user views. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be a database schema, where R_j has associated set Σ_j of full dependencies for $j \in [1, n]$. Set $\Sigma = \{R_i : \sigma \mid \sigma \in \Sigma_i\}$. Note that the elements of Σ are tagged by the relation name they refer to. Suppose that a view is defined by algebraic expression $E : \mathbf{R} \to S[V]$. It is natural to ask what dependencies will hold in the view. Formally, we say that $\mathbf{R} : \Sigma$ *implies* $E : \sigma$, denoted $\mathbf{R} : \Sigma \models E : \sigma$, if $E(\mathbf{I})$ satisfies σ for each $\mathbf{I}$ that satisfies Σ. The notion of $\mathbf{R} : \Sigma \models E : \Gamma$ for a set Γ is defined in the natural manner.

To illustrate these notions in a simple setting, we state the following easily verified result (see Exercise 10.10).

PROPOSITION 10.2.4 Let $(R[U], \Sigma)$ be a relation schema where Σ is a set of fd's and mvd's, and let $V \subseteq U$. Then

(a) $R : \Sigma \models [\pi_V(R)] : X \to A$ iff $\Sigma \models X \to A$ and $XA \subseteq V$.

(b) $R : \Sigma \models [\pi_V(R)] : X \twoheadrightarrow Y$ iff $\Sigma \models X \twoheadrightarrow Z$ for some $X \subseteq V$ and $Y = Z \cap V$.

Given a database schema $\mathbf{R}$, a family Σ of tagged full dependencies over $\mathbf{R}$, a view

expression E mapping $\mathbf{R}$ to $S[V]$, and a full dependency γ, is it decidable whether $\mathbf{R} : \Sigma \models E : \gamma$? If E ranges over the full relational algebra, the answer is no, even if the only dependencies considered are fd's.

THEOREM 10.2.5 It is undecidable, given database schema $\mathbf{R}$, tagged fd's Σ, algebra expression $E : \mathbf{R} \to S$ and fd σ over S, whether $\mathbf{R} : \Sigma \models E : \sigma$.

Proof Let $\mathbf{R} = \{R[U], S[U]\}$, $\sigma = R : \emptyset \to U$ and $\Sigma = \{\sigma\}$. Given two algebra expressions $E_1, E_2 : S \to R$, consider

$$E = R \cup [E_1(S) - E_2(S)] \cup [E_2(S) - E_1(S)]$$

Then $\mathbf{R} : \Sigma \models E : \sigma$ iff $E_1 \equiv E_2$. This is undecidable by Corollary 6.3.2. ■

In contrast, we now present a decision procedure, based on the chase, for inferring view dependencies when the view is defined using the SPCU algebra.

THEOREM 10.2.6 It is decidable whether $\mathbf{R} : \Sigma \models E : \gamma$, if E is an SPCU query and $\Sigma \cup \{\gamma\}$ is a set of (tagged) full dependencies.

Crux We prove the result for SPC queries that do not involve constants, and leave the extension to include union and constants for the reader (Exercise 10.12).

Let $E : \mathbf{R} \to S[V]$ be an SPC expression, where $S \notin \mathbf{R}$. Recall from Chapter 4 (Theorem 4.4.8; see also Exercise 4.18) that for each such expression E there is a tableau mapping $\tau_E = (\mathbf{T}, t)$ equivalent to E.

Assume now that Σ is a set of full dependencies and γ a full tgd. (The case where γ is an egd is left for the reader.) Let the tgd γ over S be expressed as the tableau (W, w). Create a new free instance $\mathbf{Z}$ out of $(\mathbf{T}, t)$ and W as follows: For each tuple $u \in W$, set $\mathbf{T}_u = \nu(\mathbf{T})$ where valuation ν maps t to u, and maps all other variables in $\mathbf{T}$ to new distinct variables. Set $\mathbf{Z} = \cup_{u \in W} \mathbf{T}_u$. It can now be verified that $\mathbf{R} : \Sigma \models E : \gamma$ iff $w \in E(chase(\mathbf{Z}, \Sigma))$. ■

In the case where $\Sigma \cup \{\gamma\}$ is a set of fd's and mvd's and the view is defined by an SPCU expression, testing the implication of a view dependency can be done in polynomial time, if jd's are involved the problem is NP-complete, and if full dependencies are considered the problem is EXPTIME-complete.

Recall from Section 8.4 that a *satisfaction family* is a family $sat(\mathbf{R}, \Sigma)$ for some set Σ of dependencies. Suppose now that SPC expression $E : R[U] \to S[V]$ is given, and that Σ is a set of full dependencies over R. Theorem 10.2.6, suitably generalized, shows that the family Γ of full dependencies implied by Σ for view E is recursive. This raises the natural question: Does $E(sat(R, \Sigma)) = sat(\Gamma)$, that is, does Γ completely characterize the image of $sat(R, \Sigma)$ under E? The affirmative answer to this question is stated next. This result follows from the proof of Theorem 10.2.6 (see Exercise 10.13).

THEOREM 10.2.7 If Σ is a set of full dependencies over $\mathbf{R}$ and $E : \mathbf{R} \to S$ is an SPC expression without constants, then there is a set Γ of full dependencies over S such that $E(sat(\mathbf{R}, \Sigma)) = sat(S, \Gamma)$.

Suppose now that $E : R[U] \to S[V]$ is given, and Σ is a *finite set* of dependencies. Can a *finite* set Γ be found such that $E(sat(R, \Sigma)) = sat(S, \Gamma)$? Even in the case where E is a simple projection and Σ is a set of fd's, the answer to this question is sometimes negative (Exercise 10.11c).

Chasing with Embedded Dependencies

We now turn to the case of (embedded) dependencies. From Theorem 10.1.2(b), it is apparent that we cannot hope to generalize Theorem 10.2.2 to obtain a decision procedure for (finite or unrestricted) implication of dependencies. As initially discussed in Chapter 9, the chase need not terminate if dependencies are used. All is not lost, however, because we are able to use the chase to obtain a proof procedure for testing unrestricted implication of a dependency by a set of dependencies.

For nonfull tgd's, we shall use the following rule. We present the rule as it applies to tableaux, but it can also be used on dependencies.

tgd rule: Let T be a tableau, and let $\sigma = (S, S')$ be a tgd. Suppose that there is a valuation θ for S that embeds S into T, but no extension θ' to $var(S) \cup var(S')$ of θ such that $\theta'(S') \subseteq T$. In this case σ can be *applied* to T.

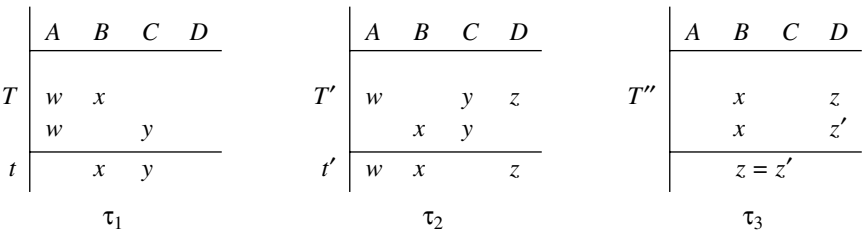
Let $\theta_1, \ldots, \theta_n$ be a list of all valuations having this property. For each $i \in [1, n]$, (nondeterministically) choose a *distinct extension*, i.e., an extension θ_i' to $var(S) \cup var(S')$ of θ_i such that each variable in $var(S') - var(S)$ is assigned a distinct new variable greater than all variables in T. (The same variable is not chosen in two extensions $\theta_i', \theta_j', i \neq j$.)

The *result of applying* σ to T is $T \cup \{\theta_i'(S') \mid i \in [1, n]\}$.

This rule is nondeterministic because variables not occurring in T are chosen for the existentially quantified variables of σ. We assume that some fixed mechanism is used for selecting these variables when given T, (S, S'), and θ.

The notion of a chasing sequence $T = T_1, T_2, \ldots$ of a tableau (or dependency) by a set of dependencies is now defined in the obvious manner. Clearly, this sequence may be infinite.

EXAMPLE 10.2.8 Let $\Sigma = \{\tau_1, \tau_2, \tau_3\}$, where



A	B	C	D	
x_1	x_2	x_3	x_4	
x_1	x_5	x_6	x_7	
x_{10}	x_2	x_6	x_{12}	application
x_{11}	x_5	x_3	x_{13}	of τ_1

(a)

A	B	C	D	
x_1	x_2	x_3	x_4	
x_1	x_5	x_6	x_7	
x_{10}	x_2	x_6	x_4	application
x_{11}	x_5	x_3	x_7	of τ_3

(b)

A	B	C	D	
x_1	x_2	x_3	x_4	
x_1	x_5	x_6	x_7	
x_{10}	x_2	x_6	x_4	
x_{11}	x_5	x_3	x_7	
x_1	x_5	x_{20}	x_4	
x_{11}	x_2	x_{21}	x_7	
x_1	x_2	x_{22}	x_7	application
x_{10}	x_5	x_{23}	x_4	of τ_2

(c)

A	B	C	D	
x_1	x_2	x_3	x_4	
x_1	x_5	x_6	x_4	
x_{10}	x_2	x_6	x_4	
x_{11}	x_5	x_3	x_4	
x_1	x_5	x_{20}	x_4	
x_{11}	x_2	x_{21}	x_4	
x_1	x_2	x_{22}	x_4	application
x_{10}	x_5	x_{23}	x_4	of τ_3

(d)

Figure 10.3: Parts of a chasing sequence

We show here only the relevant variables of τ_1, τ_2, and τ_3; all other variables are assumed to be distinct. Here $\tau_3 \equiv B \to D$.

In Fig. 10.3, we show some stages of a chasing sequence that demonstrates that $\Sigma \models_{\text{unr}} A \to D$. To do that, the chase begins with the tableau $\{\langle x_1, x_2, x_3, x_4\rangle, \langle x_1, x_5, x_6, x_7\rangle\}$. Figure 10.3 shows the results of applying $\tau_1, \tau_3, \tau_2, \tau_3$ in turn (left to right). This sequence implies that $\Sigma \models_{\text{unr}} A \to D$, because variables x_4 and x_7 are identified.

Consider now the typed tgd's:

	A	B	C	D
T'''	w	x		z
	w		y	
t''		x	y	z

τ_4

	A	B	C	D
T''''	w	x		
	w		y	
t'''	w	x	y	

τ_5

The chasing sequence of Fig. 10.3 also implies that $\Sigma \models_{\text{unr}} \tau_4$, because (x_{10}, x_2, x_6, x_4) is in the second tableau. On the other hand, we now argue that $\Sigma \not\models_{\text{unr}} \tau_5$. Consider the chasing sequence beginning as the one shown in Fig. 10.3, and continuing by applying the sequence $\tau_1, \tau_3, \tau_2, \tau_3$ repeatedly. It can be shown that this chasing sequence will not terminate and that (x_1, x_2, x_6, v) does not occur in the resulting infinite sequence for any variable v (see Exercise 10.16). It follows that $\Sigma \not\models_{\text{unr}} \tau_5$; in particular, the infinite result of the chasing sequence is a counterexample to this implication. On the other hand, this chasing sequence does not alone provide any information about whether $\Sigma \models_{\text{fin}} \tau_5$. It can be shown that this also fails.

To ensure that all relevant dependencies have a chance to influence a chasing sequence, we focus on chasing sequences that satisfy the following conditions:

(1) Whenever an egd is applied, it is applied repeatedly until it is no longer applicable.

(2) No dependency is "starved" (i.e., each dependency that is applicable infinitely often is applied infinitely often).

Even if these conditions are satisfied, it is possible to have two chasing sequences of a tableau T by typed dependencies, where one is finite and the other infinite (see Exercise 10.14).

Now consider an infinite chasing sequence $T_1 = T, T_2, \ldots$. Let us denote it by $\overline{T, \Sigma}$. Because egd's may be applied arbitrarily late in $\overline{T, \Sigma}$, for each n, tuples of T_n may be modified as the result of later applications of egd's. Thus we cannot simply take the union of some tail $T_n, T_{n+1}, \ldots$ to obtain the result of the chase. As an alternative, for the chasing sequence $\overline{T, \Sigma} = T_1, T_2, \ldots$, we define

$$chase(\overline{T, \Sigma}) = \{u \mid \exists n \, \forall m > n (u \in T_m)\}.$$

This is nonempty because (1) the "new" variables introduced by the tgd rule are always greater than variables already present; and (2) when the egd rule is applied, the newer variable is replaced by the older one.

By generalizing the techniques developed, it is easily seen that the (possibly infinite) resulting tableau satisfies all dependencies in Σ. More generally, let Σ be a set of dependencies and σ a dependency. Then one can show that $\Sigma \models_{\text{unr}} \sigma$ iff for some chasing sequence $\overline{\sigma, \Sigma}$ of σ using Σ, $chase(\overline{\sigma, \Sigma})$ is trivial. Furthermore, it can be shown that

- if for some chasing sequence $\overline{\sigma, \Sigma}$ of σ using Σ, $chase(\overline{\sigma, \Sigma})$ is trivial, then it is so for *all* chasing sequences of σ using Σ; and
- for each chasing sequence $\overline{\sigma, \Sigma} = T_1, \ldots, T_n, \ldots$ of σ using Σ, $chase(\overline{\sigma, \Sigma})$ is trivial iff T_i is trivial for some i.

This shows that, for practical purposes, it suffices to generate some chasing sequence of σ using Σ and stop as soon as some tableau in the sequence becomes trivial.

10.3 Axiomatization

A variety of axiomatizations have been developed for the family of dependencies and for subclasses such as the full typed tgd's. In view of Theorem 10.1.2, sound and complete recursively enumerable axiomatizations do not exist for finite implication of dependencies. This section presents an axiomatization for the family of full typed tgd's and typed egd's (which is sound and complete for both finite and unrestricted implication). A generalization to the embedded case (for unrestricted implication) has also been developed (see Exercise 10.21). The axiomatization presented here is closely related to the chase. In the next section, a very different kind of axiomatization for typed dependencies is discussed.

We now focus on the full typed dependencies (i.e., on typed egd's and full typed tgd's). The development begins with the introduction of a technical tool for forming the composition of tableaux queries. The axiomatization then follows.

Composition of Typed Tableaux

Suppose that $\tau = (T, t)$ and $\sigma = (S, s)$ are two full typed tableau queries over relation schema R. It is natural to ask whether there is a tableau query $\tau \bullet \sigma$ corresponding to the composition of τ followed by σ—that is, with the property that for each instance I over R,

$$(\tau \bullet \sigma)(I) = \sigma(\tau(I))$$

and, if so, whether there is a simple way to construct it. We now provide an affirmative answer to both questions. The syntactic composition of full typed tableau mappings will be a valuable tool for combining pairs of full typed tgd's in the axiomatization presented shortly.

Let $T = \{t_1, \ldots, t_n\}$ and $S = \{s_1, \ldots, s_m\}$. Suppose that tuple w is in $\sigma(\tau(I))$. Then there is an embedding ν of $s_1, \ldots, s_m$ into $\tau(I)$ such that $\nu(s) = w$. It follows that for each $j \in [1, m]$ there is an embedding μ_j of T into I, with $\mu_j(t) = \nu(s_j)$. This suggests that the tableau of $\tau \bullet \sigma$ should have mn tuples, with a block of n tuples for each s_j.

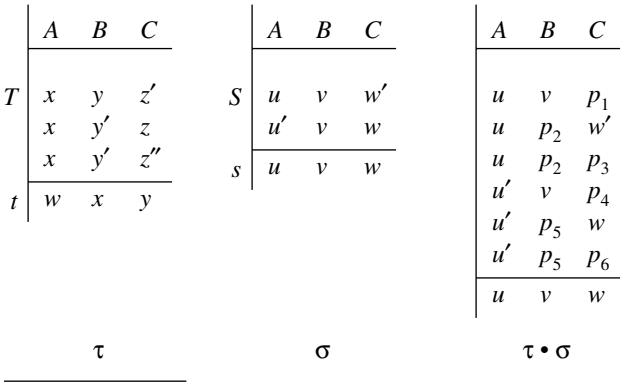
To be more precise, for each $j \in [1, m]$, let T_{s_j} be $\theta_j(T)$, where θ_j is a substitution that maps $t(A)$ to $s_j(A)$ for each attribute A of R and maps each other variable of T to a new, distinct variable not used elsewhere in the construction. Now set

$$[S](T, t) \equiv \cup\{T_{s_j} \mid j \in [1, m]\} \quad \text{and} \quad \tau \bullet \sigma \equiv ([S](T, t), s).$$

The following is now easily verified (see Exercise 10.18):

PROPOSITION 10.3.1 For full typed tableau queries τ and σ over R, and for each instance I of R, $\tau \bullet \sigma(I) = \sigma(\tau(I))$.

EXAMPLE 10.3.2 The following table shows two full typed tableau queries and their composition.



It is straightforward to verify that the syntactic operation of composition is associative.

Suppose that τ and σ are full typed tableau queries. It can be shown by simple chasing arguments that $\{\tau, \sigma\}$ and $\{\tau \bullet \sigma\}$ are equivalent as sets of dependencies. It follows that full typed tgd's are closed under finite conjunction, in the sense that each finite set of full typed tgd's over a relation schema R is equivalent to a single full typed tgd. This property does not hold in the embedded case (see Exercise 10.20).

An Axiomatization for Full Typed Dependencies

For full typed tgd's, $\tau = (T, t)$ and $\sigma = (S, s)$, we say that τ *embeds into* σ denoted $\tau \hookrightarrow \sigma$, if there is a substitution ν such that $\nu(T) \subseteq S$ and $\nu(t) = s$. Recall from Chapter 4 that $\tau \supseteq \sigma$ (considered as tableau queries) iff $\tau \hookrightarrow \sigma$. As a result we have that if $\tau \hookrightarrow \sigma$, then $\tau \models \sigma$, although the converse does not necessarily hold. Analogously, for A-egd's $\tau = (T, x = y)$ and $\sigma = (S, v = w)$, we define $\tau \hookrightarrow \sigma$ if there is a substitution ν such that $\nu(T) \subseteq S$, and $\nu(\{x, y\}) = \{v, w\}$. Again, if $\tau \hookrightarrow \sigma$, then $\tau \models \sigma$.

We now list the axioms for full typed tgd's:

FTtgd1: (triviality) For each free tuple t without constants, $(\{t\}, t)$.

FTtgd2: (embedding) If τ and $\tau \hookrightarrow \sigma$, then σ.

FTtgd3: (composition) If τ and σ, then $\tau \bullet \sigma$.

The following rules focus exclusively on typed egd's:

Tegd1: (triviality) If $x \in var(T)$, then $(T, x = x)$.

Tegd2: (embedding) If τ and $\tau \hookrightarrow \sigma$, then σ.

The final rules combining egd's and full typed tgd's use the following notation. Let $R[U]$ be a relation schema. For $A \in U$, $\overline{A}$ denotes $U - \{A\}$. Given typed A-egd $\tau = (T, x = y)$ over R, define free tuples u_x, u_y such that $u_x(A) = x$, $u_y(A) = y$ and $u_x[\overline{A}] = u_y[\overline{A}]$ consists of distinct variables not occurring in T. Define two full typed tgd's $\tau_x = (T \cup \{u_y\}, u_x)$ and $\tau_y = (T \cup \{u_x\}, u_y)$.

FTD1: (conversion) If $\tau = (T, x = y)$, then τ_x and τ_y.

FTD2: (composition) If (T, t) and $(S, x = y)$, then $([S](T, t), x = y)$.

We now have the following:

THEOREM 10.3.3 The set {FTtgd1, FTtgd2, FTtgd3, Tegd1, Tegd2, FTD1, FTD2} is sound and complete for (finite and unrestricted) logical implication of full typed dependencies.

Crux Soundness is easily verified. We illustrate completeness by showing that the FTtgd rules are complete for tgd's. Suppose that $\Sigma \models \tau = (T, t)$, where Σ is a set of full typed tgd's and (T, t) is full and typed. By Theorem 10.2.2 there is a chasing sequence of T by Σ yielding T' with $t \in T'$. Let $\sigma_1, \ldots, \sigma_n$ $(n \geq 0)$ be the sequence of elements of Σ used in the chasing sequence. It follows that $t \in \sigma_n(\ldots(\sigma_1(T)\ldots)$, and by Proposition 10.3.1, $t \in (\sigma_1 \bullet \cdots \bullet \sigma_n)(T)$. This implies that $(\sigma_1 \bullet \cdots \bullet \sigma_n) \hookrightarrow (T, t)$. A proof of τ from Σ is now obtained by starting with σ_1 (or $(\{s\}, s)$ if $n = 0$), followed by $n - 1$ applications of FTtgd3 and one application of FTtgd2 (see Exercise (10.18b). ■

The preceding techniques and the chase can be used to develop an axiomatization of unrestricted implication for the family of all typed dependencies.

10.4 An Algebraic Perspective

This section develops a very different paradigm for specifying dependencies based on the use of algebraic expressions. Surprisingly, the class of dependencies formed is equivalent to the class of typed dependencies. We also present an axiomatization that is rooted primarily in algebraic properties rather than chasing and tableau manipulations.

We begin with examples that motivate and illustrate this approach.

EXAMPLE 10.4.1 Let $R[ABCD]$ be a relation schema. Consider the tgd τ of Fig. 10.4 and the algebraic expression

$$\pi_{AC}(\pi_{AB}(R) \bowtie \pi_{BC}(R)) \subseteq \pi_{AC}(R).$$

It is straightforward to verify that for each instance I over $ABCD$,

$$I \models \tau \text{ iff } \pi_{AC}(\pi_{AB}(I) \bowtie \pi_{BC}(I)) \subseteq \pi_{AC}(I).$$

Now consider dependency σ. One can similarly verify that for each instance I over $ABCD$,

$$I \models \sigma \text{ iff } \pi_{AC}(\pi_{AB}(I) \bowtie \pi_{BC}(I)) \subseteq \pi_{AC}(\pi_{AD}(I) \bowtie \pi_{CD}(I)).$$

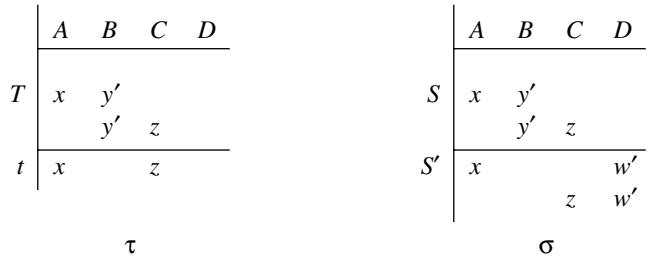


Figure 10.4: Dependencies of Example 10.4.1

The observation of this example can be generalized in the following way. A *project-join* (PJ) expression is an algebraic expression over a single relation schema using only projection and natural join. We describe next a natural recursive algorithm for translating PJ expressions into tableau queries (see Exercise 10.23). (This algorithm is also implicit in the equivalence proofs of Chapter 4.)

ALGORITHM 10.4.2

Input: a PJ expression E over relation schema $R[A_1, \ldots, A_n]$

Output: a tableau query (T, t) equivalent to E

Basis: If E is simply R, then return $(\{\langle x_1, \ldots, x_n\rangle\}, \langle x_1, \ldots, x_n\rangle)$.

Inductive steps:

1. If E is $\pi_X(q)$ and the tableau query of q is (T, t), then return $(T, \pi_X(t))$.
2. Suppose E is $q_1 \bowtie q_2$ and the tableau query of q_i is (T_i, t_i) for $i \in [1, 2]$.
   Let X be the intersection of the output sorts of q_1 and q_2. Assume without loss of generality that the two tableaux use distinct variables except that $t_1(A) = t_2(A)$ for $A \in X$. Then return $(T_1 \cup T_2, t_1 \bowtie t_2)$.

Suppose now that (T, T') is a typed dependency with the property that for some free tuple t, (T, t) is the tableau associated by this algorithm with PJ expression E, and (T', t) is the tableau associated with PJ expression E'. Suppose also that the only variables common to T and T' are those in t. Then for each instance I, $I \models (T, T')$ iff $E(I) \subseteq E'(I)$.

This raises three natural questions: (1) Is the family of PJ inclusions equivalent to the set of typed tgd's? (2) If not, can this paradigm be extended to capture all typed tgd's? (3) Can this paradigm be extended to capture typed egd's as well as tgd's?

The answer to the first question is no (see Exercise 10.24).

The answer to the second and third questions is yes. This relies on the notion of *extended relations* and *extended project-join expressions*. Let $R[A_1, \ldots, A_n]$ be a relation schema. For each $i \in [1, n]$, we suppose that there is an infinite set of attributes $A_i^1, A_i^2, \ldots$, called *copies* of A_i. The *extended schema* of R is the schema $\overline{R}[A_1^1, \ldots, A_n^1, A_1^2, \ldots, A_n^2, \ldots]$. For an instance I of R, the *extended instance* of $\overline{R}$ corresponding to I, denoted $\overline{I}$, has one "tuple" $\overline{u}$ for each tuple $u \in I$, where $\overline{u}(A_i^j) = u(A_i)$ for each $i \in [1, n]$ and $j > 0$.

An *extended project-join expression* over R is a PJ expression over $\overline{R}$ such that a

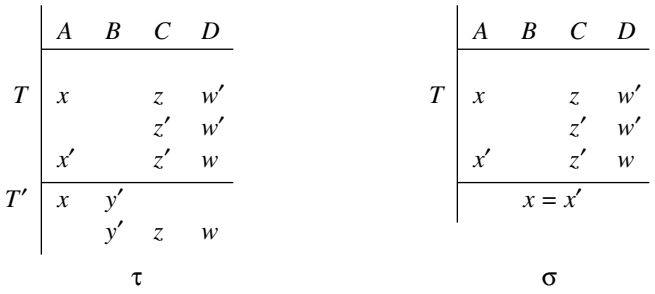


Figure 10.5: tgd and egd of Example 10.4.3

projection operator is applied first to each occurrence of $\overline{R}$. (This ensures that the evaluation and the result of such expressions involve only finite objects.) Given two extended PJ expressions E and E' with the same target sort, and instance I over R, $E(I) \subseteq_e E'(I)$ denotes $E(\overline{I}) \subseteq E'(\overline{I})$.

An *algebraic dependency* is a syntactic expression of the form $E \subseteq_e E'$, where E and E' are extended PJ expressions over a relation schema R with the same target sort. An instance I over R *satisfies* $E \subseteq_e E'$ if $E(I) \subseteq_e E'(I)$—that is, if $E(\overline{I}) \subseteq E'(\overline{I})$.

This is illustrated next.

Example 10.4.3 Consider the dependency τ of Fig. 10.5. Let

$$E = \pi_{ACD^1}(\overline{R}) \bowtie \pi_{C^1D^1}(\overline{R}) \bowtie \pi_{A^1C^1D}(\overline{R}).$$

Here we use A, $A^1, \ldots$ to denote different copies the attribute A, etc.

It can be shown that, for each instance I over $ABCD$, $I \models \tau$ iff $E_1(\overline{I}) \subseteq_e E_2(\overline{I})$, where

$$E_1 = \pi_{ACD}(E)$$
$$E_2 = \pi_{ACD}(\pi_{AB^1}(\overline{R}) \bowtie \pi_{B^1CD}(\overline{R})).$$

(See Exercise 10.25).

Consider now the functional dependency $A \to BC$ over $ABCD$. This is equivalent to $\pi_{ABC}(\overline{R}) \bowtie \pi_{AB^1C^1}(\overline{R}) \subseteq_e \pi_{ABCB^1C^1}(\overline{R})$.

Finally, consider σ of Fig. 10.5. This is equivalent to $F_1 \subseteq_e F_2$, where

$$F_1 = \pi_{AA^1}(E)$$
$$F_2 = \pi_{AA^1}(\overline{R}).$$

We next see that algebraic dependencies correspond precisely to typed dependencies.

Theorem 10.4.4 For each algebraic dependency, there is an equivalent typed dependency, and for each typed dependency, there is an equivalent algebraic dependency.

Crux Let $R[A_1, \ldots, A_n]$ be a relation schema, and let $E \subseteq_e E'$ be an algebraic dependency over R, where E and E' have target sort X. Without loss of generality, we can assume that there is k such that the sets of attributes involved in E and E' are contained in $\widehat{U} = \{A_1^1, \ldots, A_n^1, \ldots, A_1^k, \ldots, A_n^k\}$. Using Algorithm 10.4.2, construct tableau queries $\tau = (T, t)$ and $\tau' = (T', t')$ over $\widehat{U}$ corresponding to E and E'. We assume without loss of generality that τ and τ' do not share any variables except that $t(A) = t'(A)$ for each $A \in X$.

Consider T (over $\widehat{U}$). For each tuple $s \in T$ and $j \in [1, k]$,

- construct an atom $R(x_1, \ldots, x_n)$, where $x_i = s(A_i^j)$ for each $i \in [1, n]$;
- construct atoms $s(A_i^j) = s(A_i^{j'})$ for each $i \in [1, n]$ and j, j' satisfying $1 \leq j < j' \leq k$.

Let $\varphi(x_1, \ldots, x_p)$ be the conjunction of all atoms obtained from τ in this manner. Let $\psi(y_1, \ldots, y_q)$ be constructed analogously from τ'. It can now be shown (Exercise 10.26) that $E \subseteq_e E'$ is equivalent to the typed dependency

$$\forall x_1 \ldots x_p(\varphi(x_1, \ldots, x_p) \rightarrow \exists z_1 \ldots z_r \psi(y_1, \ldots, y_q)),$$

where $z_1, \ldots, z_r$ is the set of variables in $\{y_1, \ldots, y_q\} - \{x_1, \ldots, x_p\}$.

For the converse, we generalize the technique used in Example 10.4.3. For each attribute A, one distinct copy of A is used for each variable occurring in the A column. ■

An Axiomatization for Algebraic Dependencies

Figure 10.6 shows a family of inference rules for algebraic dependencies. Each of these rules stems from an algebraic property of join and project, and only the last explicitly uses a property of extended instances. (It is assumed here that all expressions are well formed.)

The use of these rules to infer dependencies is considered in Exercises 10.31, and 10.32.

It can be shown that:

Theorem 10.4.5 The family {AD1, ..., AD8} is sound and complete for inferring unrestricted implication of algebraic dependencies.

To conclude this discussion of the algebraic perspective on dependencies, we consider a new operation, direct product, and the important notion of faithfulness.

Faithfulness and Armstrong Relations

We show now that sets of typed dependencies have Armstrong relations,[1] although these may sometimes be infinite. To accomplish this, we first introduce a new way to combine instances and an important property of it.

[1] Recall that given a set Σ of dependencies over some schema R, an Armstrong relation for Σ is an instance I over R that satisfies Σ and violates every dependency not implied by Σ.

AD1: (Idempotency of Projection)
(a) $\pi_X(\pi_Y E) =_e \pi_X E$
(b) $\pi_{sort(E)} E =_e E$

AD2: (Idempotency of Join)
(a) $E \bowtie \pi_X E =_e E$
(b) $\pi_{sort(E)}(E \bowtie E') \subseteq_e E$

AD3: (Monotonicity of Projection)
If $E \subseteq_e E'$ then $\pi_X E \subseteq_e \pi_X E'$

AD4: (Monotonicity of Join)
If $E \subseteq_e E'$, then $E \bowtie E'' \subseteq_e E' \bowtie E''$

AD5: (Commutativity of Join)
$E \bowtie E' =_e E' \bowtie E$

AD6: (Associativity of Join)
$(E \bowtie E') \bowtie E'' =_e E \bowtie (E' \bowtie E'')$

AD7: (Distributivity of Projection over Join)
Suppose that $X \subseteq sort(E)$ and $Y \subseteq sort(E')$. Then
(a) $\pi_{X \cup Y}(E \bowtie E') \subseteq_e \pi_{X \cup Y}(E \bowtie \pi_Y E')$.
(b) If $sort(E) \cap sort(E') \subseteq Y$, then equality holds in (a).

AD8: (Extension)
If $X \subseteq sort(\overline{R})$ and A, A' are copies of the same attribute, then
$\pi_{AA'}\overline{R} \bowtie \pi_{AX}\overline{R} =_e \pi_{AA'X}\overline{R}$.

Figure 10.6: Algebraic dependency axioms

Let R be a relation schema of arity n. We blur our notation and use elements of **dom** $\times$ **dom** as if they were elements of **dom**. Given tuples $u = \langle x_1, \ldots, x_n \rangle$ and $v = \langle y_1, \ldots, y_n \rangle$, we define the *direct product* of u and v to be

$$u \otimes v = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle.$$

The *direct product* of two instances I, J over R is

$$I \otimes J = \{u \otimes v \mid u \in I, v \in J\}.$$

This is generalized to form k-ary direct product instances for each finite k. Furthermore, if $\mathcal{J}$ is a (finite or infinite) index set and $\{I_j \mid j \in \mathcal{J}\}$ is a family of instances over R, then $\otimes\{I_j \mid j \in \mathcal{J}\}$ denotes the (possibly infinite) direct product of this family of instances.

A dependency σ is *faithful* if for each family $\{I_j \mid j \in \mathcal{J}\}$ of nonempty instances,

$$\otimes\{I_j \mid j \in \mathcal{J}\} \models \sigma \text{ if and only if } \forall j \in \mathcal{J},\ I_j \models \sigma.$$

(The restriction that the instances be nonempty is important—if this were omitted then no nontrivial dependency would be faithful.)

The following holds because the $\otimes$ operator commutes with project, join, and "extension" (see Exercise 10.29).

PROPOSITION 10.4.6 The family of typed dependencies is faithful.

We can now prove that each set of typed dependencies has an Armstrong relation.

THEOREM 10.4.7 Let Σ be a set of typed dependencies over relation R. Then there is a (possibly infinite) instance I_Σ such that for each typed dependency σ over R, $I_\Sigma \models \sigma$ iff $\Sigma \models_{\text{unr}} \sigma$.

Proof Let Γ be the set of typed dependencies over R not in Σ^*. For each $\gamma \in \Gamma$, let I_γ be a nonempty instance that satisfies Σ but not γ. Then $\otimes\{I_\gamma \mid \gamma \in \Gamma\}$ is the desired relation. ■

This result cannot be strengthened to yield finite Armstrong relations because one can exhibit a finite set of typed tgd's with no finite Armstrong relation.

Bibliographic Notes

The papers [FV86, Kan91, Var87] all provide excellent surveys on the motivations and history of research into relational dependencies; these have greatly influenced our treatment of the subject here.

Because readers could be overwhelmed by the great number of dependency theory terms we have used a subset of the terminology. For instance, the typed single-head tgd's (that were studied in depth) are called *template* dependencies. In addition, the typed unirelational dependencies that are considered here were historically called *embedded implicational dependencies* (eid's); and their full counterparts were called *implicational dependencies* (id's). We use this terminology in the following notes.

After the introduction of fd's and mvd's, there was a flurry of research into special classes of dependencies, including jd's and ind's. *Embedded* dependencies were first introduced in [Fag77b], which defined embedded multivalued dependencies (emvd's); these are mvd's that hold in a projection of a relation. Embedded jd's are defined in the analogous fashion. This is distinct from *projected* jd's [MUV84]—these are template dependencies that correspond to join dependencies, except that some of the variables in the summary row may be distinct variables not occurring elsewhere in the dependency. Several other specialized dependencies were introduced. These include *subset dependencies* [SW82], which generalize mvd's; *mutual dependencies* [Nic78], which say that a relation is a 3-ary join; *generalized mutual dependencies* [MM79]; *transitive dependencies* [Par79], which generalize fd's and mvd's; *extended transitive dependencies* [PPG80], which generalize mutual dependencies and transitive dependencies; and *implied dependencies* [GZ82], which form a specialized class of egd's. In many cases these classes of dependencies were introduced in attempts to provide axiomatizations for the emvd's, jd's, or superclasses of them. Although most of the theoretical work studies dependencies in an abstract setting, [Sci81, Sci83] study families of mvd's and ind's as they arise in practical situations.

The proliferation of dependencies spawned interest in the development of a unifying framework that subsumed essentially all of them. Nicolas [Nic78] is credited with first observing that fd's, mvd's, and others have a natural representation in first-order logic. At

roughly the same time, several researchers reached essentially the same generalized class of dependencies that was studied in this chapter. [BV81a] introduced the class of tgd's and egd's, defined using the paradigm of tableaux. Chasing was studied in connection with both full and embedded dependencies in [BV84c]. Reference [Fag82b] introduced the class of typed dependencies, essentially the same family of dependencies but presented in the paradigm of first-order logic. Simultaneously, [YP82] introduced the algebraic dependencies, which present the same class in algebraic terms. A generalization of algebraic dependencies to the untyped case is presented in [Abi83].

Related general classes of dependencies introduced at this time are the *general dependencies* [PJ81], which are equivalent to the full typed tgd's, and *generalized dependency constraints* [GJ82], which are the full dependencies.

Importantly, several kinds of constraints that lie outside the dependencies described in this chapter have been studied in the literature. Research on the use of arbitrary first-order logic sentences as constraints includes [GM78, Nic78, Var82b]. A different extension of dependencies based on partitioning relationships, which are not expressible in first-order logic, is studied in [Cos87]. Another kind of dependency is the *afunctional dependency* of [BP83], which, as the name suggests, focuses on the portions of an instance that violate an fd. The *partition dependencies* [CK86] are not first-order expressible and are powerful; interestingly, finite and unrestricted implication coincide for this class of dependencies and are decidable in PTIME. *Order* [GH83] and *sort-set dependencies* [GH86] address properties of instances defined in terms of orderings on the underlying domain elements. There is provably no finite axiomatization for order dependencies, or for sort-set dependencies and fd's considered together (Exercise 9.8).

Another broad class of constraints not included in the dependencies discussed in this chapter is *dynamic* constraints, which focus on how data change over time [CF84, Su92, Via87, Via88]; see Section 22.6.

As suggested by the development of this chapter, one of the most significant theoretical directions addressed in connection with dependencies has been the issue of decidability of implication. The separation of finite and unrestricted implication, and the undecidability of the implication problem, were shown independently for typed dependencies in [BV81a, CLM81]. Subsequently, these results were independently strengthened to projected jd's in [GL82, Var84, YP82]. Then, after nearly a decade had elapsed, this result was strengthened to include emvd's [Her92].

On the other hand, the equivalence of finite and unrestricted implication for full dependencies was observed in [BV81a]. That deciding implication for full typed dependencies is complete in EXPTIME is due to [CLM81]. See also [BV84c, FUMY83], which present numerous results on full and embedded typed dependencies. The special case of deciding implication of a typed dependency by ind's has been shown to be PSPACE-complete [JK84b].

The issue of inferring view dependencies was first studied in [Klu80], where Theorem 10.2.5 was presented. Reference [KP82] developed Theorem 10.2.6.

The issue of attempting to characterize view images of a satisfaction family as a satisfaction family was first raised in [GZ82], where Exercise 10.11b was shown. Theorem 10.2.7 is due to [Fag82b], although a different proof technique was used there. Reference [Hul84] demonstrates that some projections of satisfaction families defined by fd's

cannot be characterized by any finite set of full dependencies (see Exercise 10.11c,d). That investigation is extended in [Hul85], where it is shown that if Σ is a family of fd's over U and $V \subseteq U$, and if $\pi_V(sat(U, \Sigma)) \neq sat(V, \Gamma)$ for any set Γ of fd's, then $\pi_V(sat(U, \Sigma)) \neq sat(V, \Gamma)$ for any finite set Γ of full dependencies.

Another primary thrust in the study of dependencies has been the search for axiomatizations for various classes of dependencies. The axiomatization presented here for full typed dependencies is due to [BV84a], which also provides an axiomatization for the embedded case. The axiomatization for algebraic dependencies is from [YP82]. An axiomatization for template dependencies is given in [SU82] (see Exercise 10.22). Research on axiomatizations for jd's is described in the Bibliographic Notes of Chapter 8.

The direct product construction is from [Fag82b]. Proposition 10.4.6 is due to [Fag82b], and the proof presented here is from [YP82]. A finite set of tgd's with no finite Armstrong relation is exhibited in [FUMY83]. The direct product has also been used in connection with tableau mappings and dependencies [FUMY83] (see Exercise 10.19). The direct product has been studied in mathematical logic; the notion of (upward) faithful presented here (see Exercise 10.28) is equivalent to the notion of "preservation under direct product" found there (see, e.g., [CK73]); and the notion of downward faithful is related to, but distinct from, the notion of "preservation under direct factors."

Reference [MV86] extends the work on direct product by characterizing the expressive power of different families of dependencies in terms of algebraic properties satisfied by families of instances definable using them.

Exercises

Exercise 10.1

(a) Show that for each first-order sentence of the form $(*)$ of Section 10.1, there exists an equivalent finite set of dependencies.

(b) Show that each dependency is equivalent to a finite set of egd's and tgd's.

Exercise 10.2 Consider the tableaux in Example 10.3.2. Give $\sigma \bullet \sigma$. Compare it (as a mapping) to σ. Give $\sigma \bullet \tau$. Compare it (as a mapping) to $\tau \bullet \sigma$.

Exercise 10.3 [DG79] Let φ be a first-order sentence with equality but no function symbols that is in prenex normal form and has quantifier structure $\exists^*\forall^*$. Prove that φ has an unrestricted model iff it has a finite model.

Exercise 10.4 This exercise concerns the dependencies of Fig. 10.2.

(a) Show that $(S, x = z)$ and $(S', x = z)$ are equivalent.

(b) Show that (T, t) and (T', t) are equivalent, but that $(T, t) \subset (T', t)$ as tableau queries.

Exercise 10.5 Let $R[ABC]$ be a relation scheme. We construct a family of egd's over R as follows. For $n \geq 0$, let

$$T_n = \{\langle x_i, y_i, z_{2i}\rangle, \langle x_i, y_{i+1}, z_{2i+1}\rangle \mid i \in [0, n]\}$$

and set $\tau_n = (T_n, z_0 = z_{2n+1})$. Note that $\tau_0 \equiv A \to C$.

(a) Prove that as egd's, $\tau_i \equiv \tau_j$ for all $i, j > 0$.

(b) Prove that $\tau_0 \models \tau_1$, but not vice versa.

Exercise 10.6

(a) [FUMY83] Prove that there are exactly three distinct (up to equivalence) full typed single-head tgd's over a binary relation. *Hint:* See Exercise 10.4.

(b) Prove that there is no set of single-head tgd's that is equivalent to the typed tgd (T_1, T_2) of Fig. 10.2.

(c) Exhibit an infinite chain $\tau_1, \tau_2, \ldots$ of typed tgd's over a binary relation where each is strictly weaker than the previous (i.e., such that $\tau_i \models \tau_{i+1}$ but $\tau_{i+1} \not\models \tau_i$ for each $i \geq 1$).

★ **Exercise 10.7** [FUMY83] Let $U = \{A_1, \ldots, A_n\}$ be a set of attributes.

(a) Consider the full typed single-head tgd (full *template dependency*) $\tau_{strongest} = (\{t_1, \ldots, t_n\}, t)$, where $t_i(A_i) = t(A_i)$ for $i \in [1, n]$, and all other variables used are distinct. Prove that $\tau_{strongest}$ is the "strongest" template dependency for U, in the sense that for each (not necessarily full) template dependency τ over U, $\tau_{strongest} \models \tau$.

(b) Let $\tau_{weakest}$ be the template dependency (S, s), where $s(A_i) = x_i$ for $i \in [1, n]$ and where S includes all tuples s' over U that satisfy (1) $s'(A_i) = x_i$ or y_i for $i \in [1, n]$, and (2) $s'(A_i) \neq x_i$ for at least one $i \in [1, n]$. Prove that $\tau_{weakest}$ is the "weakest" full template dependency U, in the sense that for each nontrivial full template dependency τ over U, $\tau \models \tau_{weakest}$.

(c) For $V \subseteq U$, a template dependency over U is *V-partial* if it can be expressed as a tableau (T, t), where t is over V. For $V \subseteq U$ exhibit a "weakest" V-partial template dependency.

Exercise 10.8 [BV84c] Prove Theorems 10.2.1 and 10.2.2.

Exercise 10.9 Prove that the triviality problem for typed tgd's is NP-complete. *Hint:* Use a reduction from tableau containment (Theorem 6.2.3).

Exercise 10.10

(a) Prove Proposition 10.2.4.

(b) Develop an analogous result for the binary natural join.

Exercise 10.11 Let $R[ABCDE]$ and $S[ABCD]$ be relation schemas, and let $V = ABCD$. Consider $\Sigma = \{A \to E, B \to E, CE \to D\}$.

(a) Describe the set Γ of fd's implied by Σ on $\pi_V(R)$.

(b) [GZ82] Show that $sat(\pi_V(R, \Sigma)) \neq sat(S, \Gamma)$. *Hint:* Consider the instance $J = \{\langle a, b_1, c, d_1\rangle, \langle a, b, c_1, d_2\rangle, \langle a_1, b, c, d_3\rangle\}$ over S.

★ (c) [Hul84] Show that there is no finite set Υ of full dependencies over S such that $\pi_V(sat(R, \Sigma)) = sat(S, \Upsilon)$ *Hint:* Say that a satisfaction family $\mathcal{F}$ over R has rank n if $\mathcal{F} = sat(R, \Gamma)$ for some Γ where the tableau in each dependency of Γ has $\leq n$ elements. Suppose that $\pi_V(sat(R, \Sigma))$ has rank n. Exhibit an instance J over V with $n + 1$ elements such that (a) $J \notin \pi_V(sat(R, \Sigma))$, and (b) J satisfies each dependency σ that is implied for $\pi_V(R)$ by Σ, and that has $\leq n$ elements in its tableau. Conclude that $J \in sat(V, \Gamma)$, a contradiction.

★(d) [Hul84] Develop a result for mvd's analogous to part (c).

Exercise 10.12 [KP82] Complete the proof of Theorem 10.2.6 for the case where Σ is a set of full dependencies and γ is a full tgd. Show how to extend that proof (a) to the case where γ is an egd; (b) to include union; and (c) to permit constants in the expression E. *Hint:* For (a), use the technique of Theorem 8.4.12; for (b) use union of tableaux, but permitting multiple output rows; and for (c) recall Exercise 8.27b.

Exercise 10.13 [Fag82b] Prove Theorem 10.2.7.

Exercise 10.14 Exhibit a typed tgd τ and a set Σ of typed dependencies such that $\Sigma \models \tau$, and there are two chasing sequences of τ by Σ, both of which satisfy conditions (1) and (2), in the definition of chasing for embedded dependencies in Section 10.2, where one sequence is finite and the other is infinite.

Exercise 10.15 Consider these dependencies:

A	B	C
x	y	
x		z
	y	z

τ_1

A	B	C
x		z
	y	z
x	y	

τ_2

$AC \to B$

τ_3

(a) Starting with input $T = \{\langle 1, 2, 3\rangle, \langle 1, 4, 5\rangle\}$, perform four steps of the chase using these dependencies.

(b) Prove that $\{\tau_1, \tau_2, \tau_3\} \not\models_{\text{unr}} A \to B$.

★ **Exercise 10.16**

(a) Prove that the chasing sequence of Example 10.2.8 does not terminate; then use this sequence to verify that $\Sigma \not\models_{\text{unr}} \tau_5$.

(b) Show that $\Sigma \not\models_{\text{fin}} \tau_5$.

(c) Exhibit a set Σ' of dependencies and a dependency σ' such that the chasing sequence of σ' with Σ' is infinite, and such that $\Sigma' \not\models_{\text{unr}} \sigma'$ but $\Sigma' \models_{\text{fin}} \sigma'$.

♠ **Exercise 10.17** [BV84c] Suppose that $\overline{T, \Sigma}$ is a chasing sequence. Prove that $chase(\overline{T, \Sigma})$ satisfies Σ.

Exercise 10.18 [BV84a] (a) Prove Proposition 10.3.1. (b) Complete the proof of Theorem 10.3.3.

Exercise 10.19 [FUMY83] This exercise uses the direct product construction for combining full typed tableau mappings. Let R be a fixed relation schema of arity n. The direct product of free tuples and tableaux is defined as for tuples and instances. Given two full typed tgd's $\tau = (T, t)$ and $\tau' = (T', t')$ over relation schema R, their *direct product* is

$$\tau \otimes \tau' = (T \otimes T', t \otimes t').$$

(a) Let τ, σ be full typed single-head tgd's over R. Prove that $\tau \otimes \sigma$ is equivalent to $\{\tau, \sigma\}$.

(b) Are $\tau \otimes \sigma$ and $\tau \bullet \sigma$ comparable as tableau queries under $\subseteq$, and, if so, how?

(c) Show that the family of typed egd's that have equality atoms referring to the same column of R is closed under finite conjunction.

Exercise 10.20 [FUMY83]

(a) Let τ and τ' be typed tgd's. Prove that $\tau \models_{\text{unr}} \tau'$ iff $\tau \models_{\text{fin}} \tau'$. *Hint:* Show that chasing will terminate in this case.

(b) Prove that there is a pair τ, τ' of typed tgd's for which there is no typed tgd τ'' equivalent to $\{\tau, \tau'\}$. *Hint:* Assume that typed tgd's were closed under conjunction in this way. Use part (a).

★ **Exercise 10.21** [BV84a] State and prove an axiomatization theorem for the family of typed dependencies.

Exercise 10.22 [SU82] Exhibit a set of axioms for template dependencies (i.e., typed single-head tgd's), and prove that it is sound and complete for unrestricted logical implication.

Exercise 10.23 Prove that Algorithm 10.4.2 is correct. (See Exercise 4.18a).

Exercise 10.24

(a) Consider the full typed tgd

$$\tau = (\{\langle x, y'\rangle, \langle x', y'\rangle, \langle x', y\rangle\}, \langle x, y\rangle).$$

Prove that there is no pair E, E' of (nonextended) PJ expressions such that τ is equivalent to $E \subseteq E'$ [i.e., such that $I \models \tau$ iff $E(I) \subseteq E'(I)$].

(b) Let τ be as in Fig. 10.5. Prove that there is no pair E, E' of (nonextended) PJ expressions such that τ is equivalent to $E \subseteq E'$.

Exercise 10.25 In connection with Example 10.4.3,

(a) Prove that τ is equivalent to $E_1 \subseteq_e E_2$.

(b) Prove that $A \to BC$ is equivalent to $\pi_{ABC}(\overline{R}) \bowtie \pi_{AB^1C^1}(\overline{R}) \subseteq_e \pi_{ABCB^1C^1}(\overline{R})$.

(c) Prove that σ is equivalent to $F_1 \subseteq_e F_2$.

★ **Exercise 10.26** Complete the proof of Theorem 10.4.4.

Exercise 10.27 An extended PJ expression E is *shallow* if it has the form $\pi_X(\overline{R})$ or the form $\pi_X(\pi_{Y_1}(\overline{R}) \bowtie \cdots \bowtie \pi_{Y_n}(\overline{R}))$. An algebraic dependency $E \subseteq_e E'$ is *shallow* if E and E' are shallow. Prove that every algebraic dependency is equivalent to a shallow one.

Exercise 10.28 [Fag82b] A dependency σ is *upward faithful* (with respect to direct products) if, for each family of nonempty instances $\{I_j \mid j \in \mathcal{J}\}$,

$$\forall j \in \mathcal{J}, I_j \models \sigma \text{ implies } \otimes\{I_j \mid j \in \mathcal{J}\} \models \sigma.$$

Analogously, σ is *downward faithful* if

$$\otimes\{I_j \mid j \in \mathcal{J}\} \models \sigma \text{ implies } \forall j \in \mathcal{J}, I_j \models \sigma.$$

(a) Show that the constraint

$$\forall x, y, y', z, z'(R(x, y, z) \wedge R(x, y', z') \rightarrow (y = y' \vee z = z'))$$

is downward faithful but not upward faithful.

(b) Show that the constraint

$$\forall x, y, z(R(x, y) \wedge R(y, z) \rightarrow R(x, z))$$

is upward faithful but not downward faithful.

Exercise 10.29 [Fag82b, YP82] Prove Proposition 10.4.6.

Exercise 10.30 [Fag82b] The direct product operator $\otimes$ is extended to instances of database schema $\mathbf{R} = \{R_1, \ldots, R_n\}$ by forming, for each $i \in [1, n]$, a direct product of the relation instances associated with R_i. Let $\mathbf{R} = \{P[A], Q[A]\}$ be a database schema. Show that the empty set of typed dependencies over $\mathbf{R}$ has no Armstrong relation. *Hint:* Find typed dependencies σ_1, σ_2 over $\mathbf{R}$ such that $\emptyset \models (\sigma_1 \vee \sigma_2)$ but $\emptyset \not\models \sigma_1$ and $\emptyset \not\models \sigma_2$.

★ **Exercise 10.31** [YP82] Let $R[ABCD]$ be a relation schema. The pseudo-transitivity rule for multivalued dependencies (Chapter 8) implies, given $A \twoheadrightarrow B$ and $B \twoheadrightarrow C$, that $A \twoheadrightarrow C$. Express this axiom in the paradigm of algebraic dependencies. Prove it using axioms {AD1, ..., AD7} (without using extended relations).

★ **Exercise 10.32** Infer the three axioms for fd's from rules {A1, ..., A8}.

Exercise 10.33 [YP82] Prove that {A1, ..., A8} is sound.

11 Design and Dependencies

When the only tool you have is a hammer,
everything begins to look like a nail.
—Anonymous

Alice: *Will we use a hammer for schema design?*
Riccardo: *Sure: decomposition, semantic modeling, . . .*
Vittorio: *And each provides nails to which the data must fit.*
Sergio: *The more intricate the hammer, the more intricate the nail.*

We have discussed earlier applications of dependencies in connection with query optimization (Section 8.4) and user views (Section 10.2). In this chapter, we briefly consider how dependencies are used in connection with the design of relational database schemas.

The problem of designing database schemas is complex and spans the areas of cognitive science, knowledge representation, software practices, implementation issues, and theoretical considerations. Due to the interaction of these many aspects (some of them integrally related to how people think and perceive the world), we can only expect a relatively narrow and somewhat simplistic contribution from theoretical techniques. As a result, the primary focus of this chapter is to introduce the kinds of formal tools that are used in the design process; a broader discussion of how to use these tools in practice is not attempted. The interested reader is referred to the Bibliographic Notes, which indicate where more broad-based treatments of relational schema design can be found.

In the following discussion, designing a relational schema means coming up with a "good" way of grouping the attributes of interest into tables, yielding a database schema. The choice of a schema is guided by semantic information about the application data provided by the designer. There are two main ways to do this, and each leads to a different approach to schema design.

Semantic data model: In this approach (Section 11.1), the application data is first described using a model with richer semantic constructs than relations. Such models are called "semantic data models." The schema in the richer model is then translated into a relational schema. The hope is that the use of semantic constructs will naturally lead to specifying good schemas.

Refinement of relational schema: This approach (Section 11.2) starts by specifying an initial relational schema, augmented with dependencies (typically fd's and mvd's). The design process uses the dependencies to improve the schema. But what is it that makes

one schema better than another? This is captured by the notion of "normal form" for relational schemas, a central notion in design theory.

Both of these approaches focus on the transformation of a schema $\mathbf{S}_1$ into a relational schema $\mathbf{S}_2$. Speaking in broad terms, three criteria are used to evaluate the result of this transformation:

(1) Preservation of data;

(2) Desirable properties of $\mathbf{S}_2$, typically described using normal forms; and

(3) Preservation of "meta-data" (i.e., information captured by schema and dependencies).

Condition (1) requires that information not be lost when instances of $\mathbf{S}_1$ are represented in $\mathbf{S}_2$. This is usually formalized by requiring that there be a "natural" mapping $\tau : \mathit{Inst}(\mathbf{S}_1) \rightarrow \mathit{Inst}(\mathbf{S}_2)$ that is one-to-one. As we shall see, the notion of "natural" can vary, depending on the data model used for $\mathbf{S}_1$.

Criterion (2) has been the focus of considerable research, especially in connection with the approach based on refining relational schemas. In this context, the notion of relational schema is generalized to incorporate dependencies, as follows: A *relation schema* is a pair (R, Σ), where R is a relation name and Σ is a set of dependencies over R. Similarly, a *database schema* is a pair $(\mathbf{R}, \Sigma)$, where $\mathbf{R}$ is a database schema as before, and Σ is a set of dependencies over $\mathbf{R}$. Some of these may be *tagged* by a single relation (i.e., have the form $R_j : \sigma$, where σ is a dependency over $R_j \in \mathbf{R}$). Others, such as ind's, may involve pairs of relations. More generally, some dependencies might range over the full set of attributes occurring in $\mathbf{R}$. (This requires a generalization of the notion of dependency satisfaction, which is discussed in Section 11.3.)

With this notation established, we return to criterion (2). In determining whether one relational schema is better than another, the main factors that have been considered are redundancy in the representation of data and update anomalies. Recall that these were illustrated in Section 8.1, using the relations *Movies* and *Showings*. We concluded there that certain schemas yielded undesirable behavior. This resulted from the nature of the information contained in the database, as specified by a set of dependencies.

Although the dependencies are in some sense the cause of the problems, they also suggest ways to eliminate them. For example, the fd

$$\mathit{Movies}: \mathit{Title} \rightarrow \mathit{Director}$$

suggests that the attribute *Director* is a characteristic of *Title*, so the two attributes belong together and can safely be represented in isolation from the other data. It should be clear that one always needs some form of semantic information to guide schema design; in the absence of such information, one cannot distinguish "good" schemas from "bad" ones (except for trivial cases). As will be seen, the notion of normal form captures some characteristics of "good" schemas by guaranteeing that certain kinds of redundancies and update anomalies will not occur. It will also be seen that the semantic data model approach to schema design can lead to relational schemas in normal form.

In broad terms, the intuition behind criterion (3) is that properties of data captured by schema $\mathbf{S}_1$ (e.g., functional or inclusion relationships) should also be captured by schema $\mathbf{S}_2$. In the context of refining relational schemas, a precise meaning will be given for this criterion in terms of "preservation" of dependencies. We shall see that there is a kind of trade-off between criteria (2) and (3).

The approach of refining relational schemas typically makes a simplifying assumption called the "pure universal relation assumption" (pure URA). Intuitively, this states that the input schema $\mathbf{S}_1$ consists of a single relation schema, possibly with some dependencies. Section 11.3 briefly considers this assumption in a more general light. In addition, the "weak" URA is introduced, and the notions of dependency satisfaction and query interpretation are extended to this context.

This chapter is more in the form of a survey than the previous chapters, for several reasons. As noted earlier, more broad-based treatments of relational schema design may be found elsewhere and require a variety of tools complementary to formal analysis. The tools presented here can at best provide only part of the skeleton of a design methodology for relational schemas. Normal forms and the universal relation assumption were active research topics in the 1970s and early 1980s and generated a large body of results. Some of that work is now considered somewhat unfashionable, primarily due to the emergence of new data models. However, we mention these topics briefly because (1) they lead to interesting theoretical issues, and (2) we are never secure from a change of fashion.

11.1 Semantic Data Models

In this section we introduce semantic data models and describe how they are used in relational database design. Semantic data models provide a framework for specifying database schemas that is considerably richer than the relational model. In particular, semantic models are arguably closer than the relational model to ways that humans organize information in their own thinking. The semantic data models are precursors of the recently emerging *object-oriented database models* (presented in a more formal fashion in Chapter 21) and are thus of interest in their own right.

As a vehicle for our discussion, we present a semantic data model, called loosely the *generic semantic model* (GSM). (This is essentially a subset of the IFO model, one of the first semantic models defined in a formal fashion.) We then illustrate how schemas from this model can be translated into relational schemas. Our primary intention is to present the basic flavor of the semantic data model approach to relational schema design and some formal results that can be obtained. The presentation itself is somewhat informal so that the notation does not become overly burdensome.

In many practical contexts, the semantic model used is the *Entity-Relationship model* (ER model) or one of its many variants. The ER model is arguably the first semantic data model that appeared in the literature. We use the GSM because it incorporates several features of the semantic modeling literature not present in the ER model, and because the GSM presents a style closer to object-oriented database models.

GSM Schemas

Figure 11.1 shows the schema **CINEMA-SEM** from the GSM, which can be used to represent information on movies and theaters. The major building blocks of such schemas are abstract classes, attributes, complex value classes, and the ISA hierarchy; these will be considered briefly in turn.

The schema of Fig. 11.1 shows five classes that hold *abstract* objects: *Person*, *Director*, *Actor*, *Movie*, and *Theater*. These correspond to collections of similar objects in the world. There are two kinds of abstract class: *primary* classes, shown using diamonds, and *subclasses* shown using circles. This distinction will be clarified further when ISA relationships are discussed.

Instances of semantic schemas are constructed from the usual *printable* classes (e.g., **string**, **integer**, **float**, etc.) and "abstract" classes. The printable classes correspond to (subsets of) the domain **dom** used in the relational model. The printable classes are indicated using squares; in Fig. 11.1 we have labeled these to indicate the kind of values that populate them. Conceptually, the elements of an abstract class such as *Person* are actual persons in the world; in the formal model internal representations for persons are used. These internal representations have come to be known as *object identifiers* (OIDs). Because they are internal, it is usually assumed that OIDs cannot be presented explicitly to users, although programming and query languages can use variables that hold OIDs. The notion of instance will be defined more completely later and is illustrated in Example 11.1.1 and Fig. 11.2.

Attributes provide one mechanism for representing relationships between objects and other objects or printable values; they are drawn using arrows. For example, the *Person* class has attributes *name* and *citizenship*, which associate strings with each person object. These are examples of *single-valued* attributes. (In this schema, all attributes are assumed to be total.) *Multivalued* attributes are also allowed; these map each object to a set of objects or printable values and are denoted using arrows with double heads. For example, *acts_in* maps actors to the movies that they have acted in. It is common to permit *inverse* constraints between pairs of attributes. For example, consider the relationship between actors and movies. It can be represented using the multivalued attribute *acts_in* on *Actor* or the multivalued attribute *actors* on *Movie*. In this schema, we assume that the attributes *acts_in* and *actors* are constrained to be inverses of each other, in the sense that $m \in acts\_in(a)$ iff $a \in actor(m)$. A similar constraint is assumed between the attributes associating movies with directors.

In the schema **CINEMA-SEM**, the *Pariscope* node is an example of a *complex value class*. Members of the underlying class are triples whose coordinates are from the classes *Theater*, Time, and *Movie*, respectively. In the GSM, each complex value is the result of one application of the tuple construct. This is indicated using a node of the form $\otimes$, with components indicated using dashed arrows. The components of each complex value can be printable, abstract, or complex values. However, there cannot be a directed cycle in the set of edges used to define the complex values. As suggested by the attribute *price*, a complex value class may have attributes. Complex value classes can also serve as the range of an attribute, as illustrated by the class *Award*.

Complex values are of independent interest and are discussed in some depth in Chapter 20. Complex values generally include hierarchical structures built from a handful of

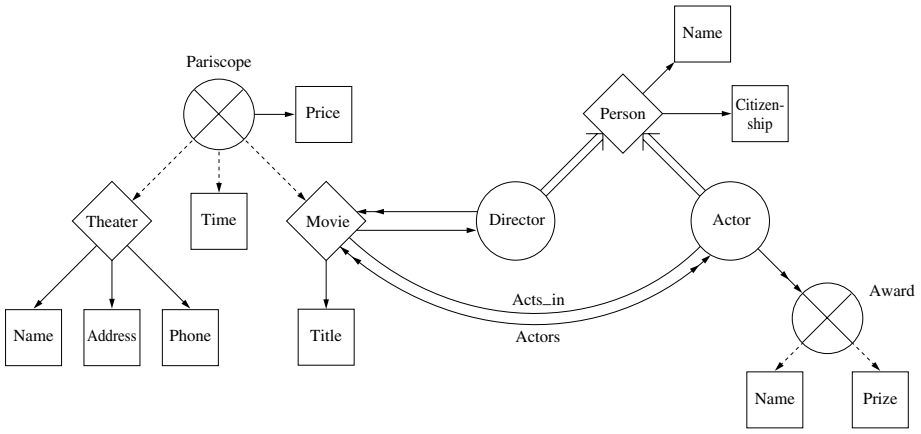


Figure 11.1: The schema **CINEMA-SEM** in the Generic Semantic Model

basic constructors, including tuple (as shown here) set, and sometimes others such as bag and list. Rich complex value models are generally incorporated into object-oriented data models and into some semantic data models. Some constructs for complex values, such as set, cannot be simulated directly using the pure relational model (see Exercise 11.24).

The final building block of the GSM is the *ISA* relationship, which represents set inclusion. In the example schema of Fig. 11.1, the ISA relationships are depicted by double-shafted arrows and indicate that the set of *Director* is a subset of *Person*, and likewise that *Actor* is a subset of *Person*. In addition to indicating set inclusion, ISA relationships indicate a form of subtyping relationship, or *inheritance*. Specifically, if class B ISA class A, then each attribute of A is also relevant (and defined for) elements of class B. In the context of semantic models, this should be no surprise because the elements of B are elements of A.

In the GSM, the graph induced by ISA relationships is a directed acyclic graph (DAG). The root nodes are primary abstract classes (represented with diamonds), and all other nodes are subclass nodes (represented with circles). Each subclass node has exactly one primary node above it. Complex value classes cannot participate in ISA relationships.

In the GSM, the tuple and multivalued attribute constructs are somewhat redundant: A multivalued attribute is easily simulated using a tuple construct. Such redundancy is typical of semantic models: The emphasis is on allowing schemas that correspond closely to the way that users think about an application. On a bit of a tangent, we also note that the tuple construct of GSM is close to the relationship construct of the ER model.

GSM Instances

Let **S** be a GSM schema. It is assumed that a fixed (finite or infinite) *domain* is associated to each printable class in **S**. We also assume a countably infinite set **obj** of OIDs.

An *instance* of **S** is a function **I** whose domain is the set of primary, subclass, and complex value classes of **S** and the set of attributes of **S**. For primary class C, $\mathbf{I}(C)$ is a finite set of OIDs, disjoint from $\mathbf{I}(C')$ for each other primary class C'. For each subclass D, $\mathbf{I}(D)$ is a set of OIDs, such that the inclusions indicated by the ISA relationships of **S** are satisfied. For complex value class C with components $D_1, \ldots, D_n$, $\mathbf{I}(C)$ is a finite set of tuples $\langle d_1, \ldots, d_n \rangle$, where $d_i \in \mathbf{I}(D_i)$ if D_i is an abstract or complex value class, and d_i is in the domain of D_i if D_i is a printable class. For a single-valued attribute f from C to C', $\mathbf{I}(f)$ is a function from $\mathbf{I}(C)$ to $\mathbf{I}(C')$ (or to the domain of C', if C' is printable). For a multivalued attribute f from C to C', $\mathbf{I}(f)$ is a function from $\mathbf{I}(C)$ to finite subsets of $\mathbf{I}(C')$ (or the domain of C', if C' is printable). Given instance **I**, attribute f from C to C', and object o in $\mathbf{I}(C)$, we often write $f(o)$ to denote $[\mathbf{I}(f)](o)$.

Example 11.1.1 Part of a very small instance $\mathbf{I}_1$ of **CINEMA-SEM** is shown in Fig. 11.2. The values of complex value *Award*, the attributes *award*, *address*, and *phone* are not shown. The symbols o_1, o_2, etc., denote OIDs.

Consider an instance $\mathbf{I}'$ that is identical to $\mathbf{I}_1$, except that o_2 is replaced by o_8 everywhere. Because OIDs serve only as internal representations that cannot be accessed

$\mathbf{I}_1(Person) =$ $\{o_1, o_2, o_3\}$	$name(o_1) = Alice$	$citizenship(o_1) = Great\ Britain$
	$name(o_2) = Allen$	$citizenship(o_2) = United\ States$
	$name(o_3) = Keaton$	$citizenship(o_3) = United\ States$
$\mathbf{I}_1(Director) = \{o_2\}$	$directed(o_2) = \{o_4, o_5\}$	
$\mathbf{I}_1(Actor) = \{o_2, o_3\}$	$acts\_in(o_2) = \{o_4, o_5\}$	
	$acts\_in(o_3) = \{o_5\}$	
$\mathbf{I}_1(Movie) = \{o_4, o_5\}$	$title(o_4) = Take\ the\ Money$ $and\ Run$	
	$title(o_5) = Annie\ Hall$	
	$director(o_4) = o_2$	$actors(o_4) = \{o_2\}$
	$director(o_5) = o_2$	$actors(o_5) = \{o_2, o_3\}$
$\mathbf{I}_1(Theater) = \{o_6\}$	$name(o_6) = Le\ Champo$	
$\mathbf{I}_1(Pariscope) =$ $\{\langle o_6, 20{:}00, o_4\rangle\}$	$price(\langle o_6, 20{:}00, o_4\rangle) = 30\text{FF}$	

Figure 11.2: Part of an instance $\mathbf{I}_1$ of **CINEMA-SEM**

explicitly, $\mathbf{I}_1$ and $\mathbf{I}'$ are considered to be identical in terms of the information that they represent.

Let **S** be a GSM schema. An *OID isomorphism* is a function μ that is a permutation on the set **obj** of OIDs and leaves all printables fixed. Such functions are extended to $Inst(\mathbf{S})$ in the natural fashion. Two instances $\mathbf{I}$ and $\mathbf{I}'$ are *OID equivalent*, denoted $\mathbf{I} \equiv_{OID} \mathbf{I}'$, if there is an OID isomorphism μ such that $\mu(\mathbf{I}) = \mathbf{I}'$. This is clearly an equivalence relation. As suggested by the preceding example, if two instances are OID equivalent, then they represent the same information. The formalism of OID equivalence will be used later when we discuss the relational simulation of GSM.

The GSM is a very basic semantic data model, and many variations on the semantic constructs included in the GSM have been explored in the literature. For example, a variety of simple constraints can be incorporated, such as cardinality constraints on attributes and disjointness between subclasses (e.g., that *Director* and *Actor* are disjoint). Another variation is to require that a class be "dependent" on an attribute (e.g., that each *Award* object must occur in the image of some *Actor*) or on a complex value class. More complex constraints based on first-order sentences have also been explored. Some semantic models support different kinds of ISA relationships, and some provide "derived data" (i.e., a form of user view incorporated into the base schema).

Translating into the Relational Model

We now describe an approach for translating semantic schemas into relational database schemas. As we shall see, the semantics associated with the semantic schema will yield dependencies of various forms in the relational schema.

A minor problem to be surmounted is that in a semantic model, real-world objects such as persons can be represented using OIDs, but printable classes must be used in the pure relational model. To resolve this, we assume that each primary abstract class has a *key*, that is, a set $\{k_1, \ldots, k_n\}$ of one or more attributes with printable range such that for each instance $\mathbf{I}$ and pair o, o' of objects in the class, $o = o'$ iff $k_1(o) = k_1(o')$ and ... and $k_n(o) = k_n(o')$. (Although more than one key might exist for a primary class, we assume that a single key is chosen.) In the schema **CINEMA-SEM**, we assume that (*person_*) *name* is the key for *Person*, that *title* is the key for *Movie*, and that (*theater_*)*name* is the key for *Theater*. (Generalizations of this approach permit the composition of attributes to serve as part of a key; e.g., including in the key for *Movie* the composition *director* $\circ$ *name*, which would give the name of the director of the movie.)

An alternative to the use of keys as just described is to permit the use of surrogates. Informally, a *surrogate* of an object is a unique, unchanging printable value that is associated with the object. Many real-world objects have natural surrogates (e.g., Social Security number for persons in the United States or France; or Invoice Number for invoices in a commercial enterprise). In other cases, abstract surrogates can be used.

The kernel of the translation of GSM schemas into relational ones concerns how objects in GSM instances can be represented using (tuples of) printables. For each class C occurring in the GSM schema, we associate a set of relational attributes, called the *representation* of C, and denoted $rep(C)$. For a printable class C, $rep(C)$ is a single attribute having this sort. For abstract class C, $rep(C)$ is a set of attributes corresponding to the key attributes of the primary class above C. For a complex value class $C = [C_1, \ldots, C_m]$, $rep(C)$ consists of (disjoint copies of) all of the attributes occurring in $rep(C_1), \ldots, rep(C_m)$.

Translation of a GSM schema into a relation schema is illustrated in the following example.

Example 11.1.2 One way to simulate schema **CINEMA-SEM** in the relational model is to use the schema **CINEMA-REL**, which has the following schema:

Person	[*name*, *citizenship*]
Director	[*name*]
Actor	[*name*]
Acts_in	[*name*, *title*]
Award	[*prize*, *year*]
Has_Award	[*name*, *prize*, *year*]
Movie	[*title*, *director_name*]
Theater	[*theater_name*, *address*, *phone*]
Pariscope	[*theater_name*, *time*, *title*, *price*]

Person	*name*	*citizenship*
	Alice	*Great Britain*
	Allen	*United States*
	Keaton	*United States*

Movie	*title*	*director_name*
	Take the Money and Run	*Allen*
	Annie Hall	*Allen*

Pariscope	*theater_name*	*time*	*title*	*price*
	Le Champo	20:00	*Take the Money and Run*	30FF

Figure 11.3: Part of a relational instance $\mathbf{I}_2$ that simulates $\mathbf{I}_1$

Figure 11.3 shows three relations in the relational simulation $\mathbf{I}_2$ of the instance $\mathbf{I}_1$ of Fig. 11.2.

In schema **CINEMA-REL**, both *Actor* and *Acts_in* are included in case there are one or more actors that did not act in any movie. For similar reasons, *Acts_in* and *Has_Award* are separated.

In contrast, we have assumed that each person has a citizenship (i.e., that citizenship is a total function). If not, then two relations would be needed in place of *Person*. Analogous remarks hold for directors, movies, theaters, and *Pariscope* objects.

In schema **CINEMA-REL**, we have not explicitly provided relations to represent the attributes *directed* of *Director* or *actors* of *Movie*. This is because both of these are inverses of other attributes, which are represented explicitly (by *Movie* and *Acts_in*, respectively).

If we were to consider the complex value class *Awards* of **CINEMA-SEM** to be dependent on the attribute *award*, then the relation *Award* could be omitted.

Suppose that $\mathbf{I}$ is an instance of **CINEMA-SEM** and that $\mathbf{I}'$ is the simulation of $\mathbf{I}$. The semantics of **CINEMA-SEM**, along with the assumed keys, imply that $\mathbf{I}'$ will satisfy several dependencies. This includes the following fd's (in fact, key dependencies):

$$
\begin{array}{lcl}
\mathit{Person} & : & \mathit{name} \rightarrow \mathit{citizenship} \\
\mathit{Movie} & : & \mathit{title} \rightarrow \mathit{director\_name} \\
\mathit{Theater} & : & \mathit{theater\_name} \rightarrow \mathit{address}, \mathit{phone} \\
\mathit{Pariscope} & : & \mathit{theater\_name}, \mathit{time}, \mathit{title} \rightarrow \mathit{price}
\end{array}
$$

A number of ind's are also implied:

$$
\begin{array}{lcl}
\mathit{Director}[\mathit{name}] & \subseteq & \mathit{Person}[\mathit{name}] \\
\mathit{Actor}[\mathit{name}] & \subseteq & \mathit{Person}[\mathit{name}]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Movie}[\mathit{director\_name}] & \subseteq & \mathit{Director}[\mathit{name}] \\
\mathit{Acts\_in}[\mathit{name}] & \subseteq & \mathit{Actor}[\mathit{name}] \\
\mathit{Acts\_in}[\mathit{title}] & \subseteq & \mathit{Movie}[\mathit{title}] \\
\mathit{Has\_Award}[\mathit{name}] & \subseteq & \mathit{Actor}[\mathit{name}]
\end{array}
$$

$$\begin{array}{lcl} \mathit{Has\_Award}[\mathit{prize}, \mathit{year}] & \subseteq & \mathit{Award}[\mathit{prize}, \mathit{year}] \\ \\ \mathit{Pariscope}[\mathit{theater\_name}] & \subseteq & \mathit{Theater}[\mathit{theater\_name}] \\ \mathit{Pariscope}[\mathit{title}] & \subseteq & \mathit{Movie}[\mathit{title}] \end{array}$$

The first group of ind's follows from ISA relationships; the second from restrictions on attribute ranges; and the third from restrictions on the components of complex values. All but one of the ind's here are unary, because all of the keys, except the key for *Award*, are based on a single attribute.

Preservation of Data

Suppose that $\mathbf{S}$ is a GSM schema with keys for primary classes, and $(\mathbf{R}, \Sigma \cup \Gamma)$ is a relational schema that simulates it, constructed in the fashion illustrated in Example 11.1.2, where Σ is the set of fd's and Γ is the set of ind's. As noted in criterion (1) at the beginning of this chapter, it is desirable that there be a natural one-to-one mapping τ from instances of $\mathbf{S}$ to instances of $(\mathbf{R}, \Sigma \cup \Gamma)$. To formalize this, two obstacles need to be overcome. First, we have not developed a query language for the GSM. (In fact, no query language has become widely accepted for any of the semantic data models. In contrast, some query languages for object-oriented database models are now gaining wide acceptance.) We shall overcome this obstacle by developing a rather abstract notion of "natural" for this context.

The second obstacle stems from the fact that OID-equivalent GSM instances hold essentially the same information. Thus we would expect OID-equivalent instances to map to the same relational instance.[1] To refine criterion (1) for this context, we are searching for a one-to-one mapping from $\mathit{Inst}(\mathbf{S})/\equiv_{OID}$ into $\mathit{Inst}(\mathbf{R}, \Sigma \cup \Gamma)$.

A mapping $\tau : \mathit{Inst}(\mathbf{S}) \to \mathit{Inst}(\mathbf{R}, \Sigma \cup \Gamma)$ is *OID consistent* if $\mathbf{I} \equiv_{OID} \mathbf{I}'$ implies $\tau(\mathbf{I}) = \tau(\mathbf{I}')$. In this case, we can view τ as a mapping with domain $\mathit{Inst}(\mathbf{S})/\equiv_{OID}$. The mapping τ *preserves the active domain* if for each $\mathbf{I} \in \mathit{Inst}(\mathbf{S})$, $\mathit{adom}(\tau(\mathbf{I})) = \mathit{adom}(\mathbf{I})$. [The *active domain* of a GSM instance $\mathbf{I}$, denoted $\mathit{adom}(\mathbf{I})$, is the set of all printables that occur in $\mathbf{I}$.]

The following can be verified (see Exercise 11.3):

Theorem 11.1.3 (Informal) Let $\mathbf{S}$ be a GSM schema with keys for primary classes, and let $(\mathbf{R}, \Sigma \cup \Gamma)$ be a relational simulation of $\mathbf{S}$. Then there is a function $\tau : \mathit{Inst}(\mathbf{S}) \to \mathit{Inst}(\mathbf{R}, \Sigma \cup \Gamma)$ such that τ is OID consistent and preserves the active domain, and such that $\tau : \mathit{Inst}(\mathbf{S})/\equiv_{OID} \to \mathit{Inst}(\mathbf{R}, \Sigma \cup \Gamma)$ is one-to-one and onto.

Properties of the Relational Schema

We now consider criteria (2) and (3) to highlight desirable properties of relational schemas that simulate GSM schemas.

[1] When artificial surrogates are used to represent OIDs in the relational database, one might have to use a notion of an "equivalent" relational database instances as well.

Criterion (2) for schema transformations concerns desirable properties of the target schema. We now describe three such properties resulting from the transformation of GSM schemas into relational ones.

Suppose again that $\mathbf{S}$ is a GSM schema with keys, and $(\mathbf{R}, \Sigma \cup \Gamma)$ is a relational simulation of it. We assume as before that no constraints hold for $\mathbf{S}$, aside from those implied by the constructs in $\mathbf{S}$ and the keys.

The three properties are as follows:

1. First, Σ is equivalent to a family of key dependencies; in the terminology of the next section, this means that each of the relation schemas obtained is in Boyce-Codd Normal Form (BCNF). Furthermore, the only mvd's satisfied by relations in $\mathbf{R}$ are implied by Σ, and so the relation schemas are in fourth normal form (4NF).
2. Second, the family Γ of ind's is acyclic (see Chapter 9). That is, there is no sequence $R_1[X_1] \subseteq R_2[Y_1]$, $R_2[X_2] \subseteq R_3[Y_2]$, $\ldots$, $R_n[X_n] \subseteq R_1[Y_n]$ of ind's in the set. By Theorem 9.4.5, this implies that logical implication can be decided for $(\Sigma \cup \Gamma)$ and that finite and unrestricted implication coincide.
3. Finally, each ind $R[X] \subseteq S[Y]$ in Γ *key based*. That is, Y is a (minimal) key of S under Σ.

Together these properties present a number of desirable features. In particular, dependency implication is easy to check. Given a database schema $\mathbf{R}$ and sets Σ of fd's and Γ of ind's over $\mathbf{R}$, Σ and Γ are *independent* if (1) for each fd σ over $\mathbf{R}$, $(\Sigma \cup \Gamma) \models \sigma$ implies $\Sigma \models \sigma$, and (2) for each ind γ over $\mathbf{R}$, $(\Sigma \cup \Gamma) \models \gamma$ implies $\Gamma \models \gamma$. Suppose that $\mathbf{S}$ is a GSM schema and that $(\mathbf{R}, \Sigma \cup \Gamma)$ is a relational simulation of $\mathbf{S}$. It can be shown that the three aforementioned properties imply that Σ and Γ are independent (see Exercise 11.4).

To conclude this section, we consider criterion (3). This criterion concerns the preservation of meta-data. We do not attempt to formalize this criterion for this context, but it should be clear that there is a close correspondence between the dependencies in $\Sigma \cup \Gamma$ and the constructs used in $\mathbf{S}$. In other words, the semantics of the application as expressed by $\mathbf{S}$ is also captured, in the relational representation, by the dependencies $\Sigma \cup \Gamma$.

The preceding discussion assumes that no dependency holds for $\mathbf{S}$, aside from those implied by the keys and the constructs in $\mathbf{S}$. However, in many cases constraints will be incorporated into $\mathbf{S}$ that are not directly implied by the structure of $\mathbf{S}$. For instance, recall Example 11.1.2, and suppose that the fd *Pariscope* : *theater_name*, *time* $\rightarrow$ *price* is true for the underlying data. The relational simulation will have to include this dependency and, as a result, the resulting relational schema may be missing some of the desirable features (e.g., the family of fd's is not equivalent to a set of keys and the schema is no longer in BCNF). This suggests that a semantic model might be used to obtain a coarse relational schema, which might be refined further using the techniques for improving relational schemas developed in the next section.

11.2 Normal Forms

In this section, we consider schema design based on the refinement of relational schemas and normal forms, which provide the basis for this approach. The articulation of these normal forms is arguably the main contribution of relational database theory to the realm of schema design. We begin the discussion by presenting two of the most prominent normal forms and a design strategy based on "decomposition." We then develop another normal form that overcomes certain technical problems of the first two, and describe an associated design strategy based on "synthesis." We conclude with brief comments on the relationship of ind's with decomposition.

When all the dependencies in a relational schema $(\mathbf{R}, \Sigma)$ are considered to be tagged, one can view the *database schema* as a set $\{(R_1, \Sigma_1), \ldots, (R_n, \Sigma_n)\}$, where each (R_j, Σ_j) is a relation schema and the R_j's are distinct. In particular, an *fd schema* is a relation schema (R, Σ) or database schema $(\mathbf{R}, \Sigma)$, where Σ is a set of tagged fd's; this is extended in the natural fashion to other classes of dependencies. Much of the work on refinement of relational schemas has focused on fd schemas and (fd + mvd) schemas. This is what we consider here. (The impact of the ind's is briefly considered at the end of this section.)

A normal form restricts the set of dependencies that are allowed to hold in a relation schema. The main purpose of the normal forms is to eliminate at least some of the redundancies and update anomalies that might otherwise arise. Intuitively, schemas in normal form are "good" schemas.

We introduce next two kinds of normal forms, namely BCNF and 4NF. (We will consider a third one, 3NF, later.) We then consider techniques to transform a schema into such desirable normal forms.

BCNF: Do Not Represent the Same Fact Twice

Recall the schema $(Movies[T(itle), D(irector), A(actor)], \{T \rightarrow D\})$ from Section 8.1. As discussed there, the *Movies* relation suffers from various anomalies, primarily because there is only one *Director* associated with each *Title* but possibly several *Actors.* Suppose that $(R[U], \Sigma)$ is a relation schema, $\Sigma \models X \rightarrow Y$, $Y \not\subseteq X$ and $\Sigma \not\models X \rightarrow U$. It is not hard to see that anomalies analogous to those of *Movies* can arise in R. Boyce-Codd normal form prohibits this kind of situation.

Definition 11.2.1 A relation schema $(R[U], \Sigma)$ is in *Boyce-Codd normal form* (BCNF) if $\Sigma \models X \rightarrow U$ whenever $\Sigma \models X \rightarrow Y$ for some $Y \not\subseteq X$. An fd schema $(\mathbf{R}, \Sigma)$ is in BCNF if each of its relation schemas is.

BCNF is most often discussed in cases where Σ involves only functional dependencies. In such cases, if (R, Σ) is in BCNF, the anomalies of Section 8.1 do not arise. An essential intuition underlying BCNF is, "Do not represent the same fact twice."

The question now arises: What does one do with a relation schema (R, Σ) that is not in BCNF? In many cases, it is possible to decompose this schema into subschemas $(R_1, \Sigma_1), \ldots, (R_n, \Sigma_n)$ without information loss. As a simple example, *Movies* can be decomposed into

$$\left\{ \begin{array}{l} (Movie\_director[TD], \{T \rightarrow D\}), \\ (Movie\_actors[TA], \emptyset) \end{array} \right\}$$

A general framework for decomposition is presented shortly.

4NF: Do Not Store Unrelated Information in the Same Relation

Consider the relation schema ($Studios[N(ame), D(irector), L(ocation)]$, $\{N \twoheadrightarrow D|L\}$). A tuple $\langle n, d, l \rangle$ is in *Studios* if director d is employed by the studio with name n and if this studio has an office in location l. Only trivial fd's are satisfied by all instances of this schema, and so it is in BCNF. However, update anomalies can still arise, essentially because the D and L values are independent from each other. This gives rise to the following generalization of BCNF[2]:

Definition 11.2.2 A relation schema $(R[U], \Sigma)$ is in *fourth normal form* (*4NF*) if

(a) whenever $\Sigma \models X \rightarrow Y$ and $Y \not\subseteq X$, then $\Sigma \models X \rightarrow U$

(b) whenever $\Sigma \models X \twoheadrightarrow Y$ and $Y \not\subseteq X$, then $\Sigma \models X \rightarrow U$.

An (fd + mvd) schema $(\mathbf{R}, \Sigma)$ is in 4NF if each of its relation schemas is.

It is clear that if a relation schema is in 4NF, then it is in BCNF. It is easily seen that *Studios* can be decomposed into two 4NF relations, without loss of information and that the resulting relation schemas do not have the update anomalies mentioned earlier. An essential intuition underlying 4NF is, "Do not store unrelated information in the same relation."

The General Framework of Decomposition

One approach to refining relational schemas is *decomposition*. In this approach, it is usually assumed that the original schema consists of a single wide relation containing all attributes of interest. This is referred to as the *pure universal relation assumption,* or *pure URA*. A relaxation of the pure URA, called the "weak URA," is considered briefly in Section 11.3.

The pure URA is a simplifying assumption, because in practice the original schema is likely to consist of several tables, each with its own dependencies. In that case, the design process described for the pure URA is applied separately to each table. We adopt the pure URA here. In this context, the schema transformation produced by the design process consists of decomposing the original table into smaller tables by using the projection operator. (In an alternative approach, selection is used to yield so-called *horizontal decompositions*.)

We now establish the basic framework of decompositions. Let $(U[Z], \Sigma)$ be a relation schema. A *decomposition* of $(U[Z], \Sigma)$ is a database schema $\mathbf{R} = \{R_1[X_1], \ldots, R_n[X_n]\}$ with dependencies Γ, where $\cup\{X_j \mid j \in [1, n]\} = Z$. (The relation name '$U$' is used to suggest that it is a "universal" relation.) In the sequel, we often use relation names U (R_i) and attribute sets Z (X_i), interchangeably if ambiguity does not arise.

[2] The motivation behind the names of several of the normal forms is largely historical; see the Bibliographic Notes.

We now consider the three criteria for schema transformation in the context of decomposition. As already suggested, criterion (2) is evaluated in terms of the normal forms. With regard to the preservation of data (1), the "natural" mapping from R to $\mathbf{R}$ is obtained by projection: The *decomposition mapping* of $\mathbf{R}$ is the function $\pi_{\mathbf{R}} : Inst(U) \to Inst(\mathbf{R})$ such that for $I \in inst(U)$, we have $\pi_{\mathbf{R}}(I)(R_j) = \pi_{R_j}(I)$. Criterion (1) says that the decomposition should not lose information when I is replaced by its projections (i.e., it should be one-to-one).

A natural property implying that a decomposition is one-to-one is that the original instance can be obtained by joining the component relations. Formally, a decomposition is said to have the *lossless join* property if for each instance $\mathbf{I}$ of (U, Σ) the join of the projections is the original instance, i.e., $\bowtie(\pi_{\mathbf{R}}(\mathbf{I})) = \mathbf{I}$. It is easy to test if a decomposition $\mathbf{R} = \{R_1, \ldots, R_n\}$ of (U, Σ) has the lossless join property. Consider the query $q(I) = \pi_{R_1}(I) \bowtie \cdots \bowtie \pi_{R_n}(I)$. The lossless join property means that $q(I) = I$ for every instance I over (U, Σ). But $q(I) = I$ simply says that I satisfies the jd $\bowtie[\mathbf{R}]$. Thus we have the following:

Theorem 11.2.3 Let (U, Σ) be a (full dependencies) schema and $\mathbf{R}$ a decomposition for (U, Σ). Then $\mathbf{R}$ has the lossless join property iff $\Sigma \models \bowtie[\mathbf{R}]$.

The preceding implication can be tested using the chase (see Chapter 8), as illustrated next.

Example 11.2.4 Recall the schema $(Movies[TDA], \{T \to D\})$. As suggested earlier, a decomposition into BCNF is $\mathbf{R} = \{TD, TA\}$. This decomposition has the lossless join property. The tableau associated with the jd $\sigma = \bowtie[TD, TA]$ is as follows:

	T	D	A
T_σ	t	d	a_1
	t	d_1	a
t_σ	t	d	a

Consider the chase of $\langle T_\sigma, t_\sigma \rangle$ with $\{T \to D\}$. Because the two first tuples agree on the T column, d and d_1 are merged because of the fd. Thus $\langle t, d, a \rangle \in chase(T_\sigma, t_\sigma, \{T \to D\})$. Hence $T \to D$ implies the jd σ, so $\mathbf{R}$ has the lossless join property. (See also Exercise 11.9.)

Referring to the preceding example, note that it is possible to represent information in $\mathbf{R}$ that cannot be directly represented in *Movies*. Specifically, in the decomposed schema we can represent a movie with a director but no actors and a movie with an actor but no director. This indicates, intuitively, that a decomposed schema may have more information capacity

than the original (see Exercise 11.23). In practice, this additional capacity is exploited; in fact, it provides part of the solution of so-called deletion anomalies.

Remark 11.2.5 In the preceding example, we used the natural join operator to reconstruct decompositions. Interestingly, there are cases in which the natural join does not suffice. To show that a decomposition is one-to-one, it suffices to exhibit an inverse to the projection, called a *reconstruction mapping*. If Σ is permitted to include very general constraints expressed in first-order logic that may not be dependencies per se, then there are one-to-one decompositions whose reconstruction mappings are *not* the natural join (see Exercise 11.20).

We now consider criterion (3), the preservation of meta-data. In the context of decomposition, this is formalized in terms of "dependency preservation": Given schema (U, Σ), which is replaced by a decomposition $\mathbf{R} = \{R_1, \ldots, R_n\}$, we would like to find for each j a family Γ_j of dependencies over R_j such that $\cup_j \Gamma_j$ is equivalent to the original Σ. In the case where Σ is a set of fd's, we can make this much more precise. For $V \subseteq U$, let

$$\pi_V(\Sigma) = \{X \to A \mid XA \subseteq V \text{ and } \Sigma \models X \to A\},$$

let $\Gamma_j = \pi_{X_j}(\Sigma)$, and let $\Gamma = \cup_j \Gamma_j$. Obviously, $\Sigma \models \Gamma$. (See Proposition 10.2.4.) Intuitively, Γ consists of the dependencies in Σ^* that are local to the relations in the decomposition $\mathbf{R}$. The decomposition $\mathbf{R}$ is said to be *dependency preserving* iff $\Gamma \equiv \Sigma$. In other words, Σ can be enforced by the dependencies local in the decomposition. It is easy to see that the decomposition of Example 11.2.4 is dependency preserving.

Given an fd schema (U, Σ) and $V \subseteq U$, $\pi_V(\Sigma)$ has size exponential in V, simply because of trivial fd's. But perhaps there is a smaller set of fd's that is equivalent to $\pi_V(\Sigma)$. A *cover* of a set Γ of fd's is a set Γ' of fd's such that $\Gamma' \equiv \Gamma$. Unfortunately, in some cases the smallest cover for a projection $\pi_V(\Sigma)$ is exponential in the size of Σ (see Exercise 11.11).

What about projections of sets of mvd's? Suppose that Σ is a set of fd's and mvd's over U. Let $V \subseteq U$ and

$$\pi_V^{mvd}(\Sigma) = \{[X \twoheadrightarrow (Y \cap V) | (Z \cap V)] \mid [X \twoheadrightarrow Y | Z] \in \Sigma^* \text{ and } X \subseteq V\}.$$

Consider a decomposition $\mathbf{R}$ of (U, Σ). Viewed as constraints on U, the sets $\pi_{R_j}^{mvd}(\Sigma)$ are now embedded mvd's. As we saw in Chapter 10, testing implication for embedded mvd's is undecidable. However, the issue of testing for dependency preservation in the context of decompositions involving fd's and mvd's is rather specialized and remains open.

Fd's and Decomposition into BCNF

We now present a simple algorithm for decomposing an fd schema (U, Σ) into BCNF relations. The decomposition produced by the algorithm has the lossless join property but is not guaranteed to be dependency preserving.

We begin with a simple example.

EXAMPLE 11.2.6 Consider the schema (U, Σ), where U has attributes

TITLE *D_NAME* *TIME* *PRICE*
TH_NAME *ADDRESS* *PHONE*

and Σ contains

$FD1$: $TH\_NAME \rightarrow ADDRESS, PHONE$
$FD2$: $TH\_NAME, TIME, TITLE \rightarrow PRICE$
$FD3$: $TITLE \rightarrow D\_NAME$

Intuitively, schema (U, Σ) represents a fragment of the real-world situation represented by the semantic schema **CINEMA-SEM**.

A first step toward transforming this into a BCNF schema is to decompose using $FD1$, to obtain the database schema

$$\left\{ \begin{array}{l} (\{TH\_NAME, ADDRESS, PHONE\}, \{FD1\}), \\ (\{TH\_NAME, TITLE, TIME, PRICE, D\_NAME\}, \{FD2, FD3\}) \end{array} \right\}$$

Next $FD3$ can be used to split the second relation, obtaining

$$\left\{ \begin{array}{l} (\{TH\_NAME, ADDRESS, PHONE\}, \{FD1\}) \\ (\{TITLE, D\_NAME\}, \{FD3\}) \\ (\{TH\_NAME, TITLE, TIME, PRICE\}, \{FD2\}) \end{array} \right\}$$

which is in BCNF. It is easy to see that this decomposition has the lossless join property and is dependency preserving. In fact, in this case, we obtain the same relational schema as would result from starting with a semantic schema.

We now present the following:

ALGORITHM 11.2.7 (BCNF Decomposition)

Input: A relation schema (U, Σ), where Σ is a set of fd's.

Output: A database schema $(\mathbf{R}, \Gamma)$ in BCNF

1. Set $(\mathbf{R}, \Gamma) := \{(U, \Sigma)\}$.
2. Repeat until $(\mathbf{R}, \Gamma)$ is in BCNF:
   (a) Choose a relation schema $(S[V], \Omega) \in \mathbf{R}$ that is not in BCNF.
   (b) Choose nonempty, disjoint $X, Y, Z \subset V$ such that
      (i) $XYZ = V$;
      (ii) $\Omega \models X \rightarrow Y$; and
      (iii) $\Omega \not\models X \rightarrow A$ for each $A \in Z$.
   (c) Replace $(S[V], \Omega)$ in $\mathbf{R}$ by $(S_1[XY], \pi_{XY}(\Omega))$ and $(S_2[XZ], \pi_{XZ}(\Omega))$.
   (d) If there are $(S[V], \Omega)$, $(S'[V'], \Omega')$ in $\mathbf{R}$ with $V \subseteq V'$, then remove $S([V], \Omega)$ from $\mathbf{R}$.

It is easily seen that the preceding algorithm terminates [each iteration of the loop eliminates at least one violation of BCNF among finitely many possible ones]. The following is easily verified (see Exercise 11.10):

THEOREM 11.2.8 The BCNF Decomposition Algorithm yields a BCNF schema and a decomposition that has the lossless join property.

What is the complexity of running the BCNF Decomposition Algorithm? The main expenses are (1) examining subschemas $(S[V], \Omega)$ to see if they are in BCNF and, if not, finding a way to decompose them; and (2) computing the projections of Ω. (1) is polynomial, but (2) is inherently exponential (see Exercise 11.11). This suggests a modification to the algorithm, in which only the relational schemas $S[V]$ are computed at each stage, but $\Omega = \pi_V(\Sigma)$ is not. However, the problem of determining, given fd schema (U, Σ) and $V \subseteq U$, whether $(V, \pi_V(\Sigma))$ is in BCNF is co-NP-complete (see Exercise 11.12). Interestingly, a polynomial time algorithm does exist for finding *some* BCNF decomposition of an input schema (U, Σ) (see Exercise 11.13).

When applying BCNF decomposition to the schema of Example 11.2.6, the same result is achieved regardless of the order in which the dependencies are applied. This is not always the case, as illustrated next.

EXAMPLE 11.2.9 Consider $(ABC, \{A \to B, B \to C\})$. This has two BCNF decompositions

$$\mathbf{R}_1 = \{(AB, \{A \to B\}), (BC, \{B \to C\})\}$$
$$\mathbf{R}_2 = \{(AB, \{A \to B\}), (AC, \emptyset)\}.$$

Note that $\mathbf{R}_1$ is dependency preserving, but $\mathbf{R}_2$ is not.

Fd's, Dependency Preservation, and 3NF

It is easy to check that the schemas in Examples 11.2.4, 11.2.6, and 11.2.9 have dependency-preserving decompositions into BCNF. However, this is not always achievable, as shown by the following example.

EXAMPLE 11.2.10 Consider a schema *Lectures*[*C*(*ourse*), *P*(*rofessor*), *H*(*our*)], where tuple $\langle c, p, h\rangle$ indicates that course c is taught by professor p at hour h. We assume that *Hour* ranges over weekday-time pairs (e.g., Tuesday at 4PM) and that a given course may have lectures during several hours each week. Assume that the following two dependencies are to hold:

$$\Sigma = \left\{ \begin{array}{c} C \to P \\ PH \to C \end{array} \right\}.$$

In other words, each course is taught by only one professor, and a professor can teach only one course at a given hour.

The schema (*Lectures*, Σ) is not in BCNF because $\Sigma \models C \to P$, but $\Sigma \not\models C \to H$. Applying the BCNF Decomposition Algorithm yields $\mathbf{R} = \{(CP, \{C \to P\}), (CH, \emptyset)\}$.

It is easily seen that $\{CP : C \to P\} \not\models \Sigma$, and so this decomposition does not preserve dependencies. A simple case analysis shows that there is no BCNF decomposition of *Lectures* that preserves dependencies.

This raises the question: Is there a less restrictive normal form for fd's so that a lossless join decomposition that preserves dependencies can always be found? The affirmative answer is based on "third normal form" (3NF). To define it, we need some auxiliary notions. Suppose that $(R[U], \Sigma)$ is an fd schema. A *superkey* of R is a set $X \subseteq U$ such that $\Sigma \models X \to U$. A *key* of R is a minimal superkey. A *key attribute* is an attribute $A \in U$ that is in some key of R. We now have the following:

Definition 11.2.11 An fd schema (U, Σ) is in *third normal form* (*3NF*) if whenever $X \to A$ is a nontrivial fd implied by Σ, then either X is a superkey or A is a key attribute. An fd schema $(\mathbf{R}, \Sigma)$ is in 3NF if each of its components is.

Example 11.2.12 Recall the schema (*Lectures*,$\{C \to P, PH \to C\}$) described in Example 11.2.10. Here PH is a key, so P is a key attribute. Thus the schema is in 3NF.

A 3NF Decomposition Algorithm can be defined in analogy to the BCNF Decomposition Algorithm. We present an alternative approach, generally referred to as "synthesis."

Given a set Σ of fd's, a *minimal cover* of Σ is a set Σ' of fd's such that

(a) each dependency in Σ' has the form $X \to A$, where A is an attribute;

(b) $\Sigma' \equiv \Sigma$;

(c) no proper subset of Σ' implies Σ; and

(d) for each dependency $X \to A$ in Σ', there is no $Y \subset X$ such that $\Sigma \models Y \to A$.

A minimal cover can be viewed as a reduced representative for a set of fd's. It is straightforward to develop a polynomial time algorithm for producing a minimal cover of a set of fd's (see Exercise 11.16).

We now have the following:

Algorithm 11.2.13 (3NF Synthesis)

Input: A relation schema (U, Σ), where Σ is a set of fd's that is a minimal cover. We assume that each attribute of U occurs in at least one fd of Σ.

Output: An fd schema $(\mathbf{R}, \Gamma)$ in 3NF

1. If there is an fd $X \to A$ in Σ, where $XA = U$, then output (U, Σ).
2. Otherwise
   (a) for each fd $X \to A$ in Σ, include the relational schema $(XA, \{X \to A\})$ in the output schema $(\mathbf{R}, \Gamma)$; and

(b) choose a key X of U under Σ, and include $(X, \emptyset)$ in the output.

A central aspect of this algorithm is to form a relation XA for each fd $X \to A$ in Σ. Intuitively, then, the output relations result from combining or "synthesizing" attributes rather than decomposing the full attribute set.

The following is easily verified (see Exercise 11.17):

THEOREM 11.2.14 The 3NF Synthesis Algorithm decomposes a relation schema into a database schema in 3NF that has the lossless join property and preserves dependencies.

Several improvements to the basic 3NF Synthesis Algorithm can be made easily. For example, different schemas obtained in step (2.a) can be merged if they come from fd's with the same left-hand side. Step (2.b) is not needed if step (2.a) already produced a schema whose set of attributes is a superkey for (U, Σ). In many practical situations, it may be appropriate to omit step (2.b) of the algorithm. In that case, the decomposition preserves dependencies but does not necessarily satisfy the lossless join property.

In the preceding algorithm, it was assumed that each attribute of U occurs in at least one fd of Σ. Obviously, this may not always be the case, for example, the attribute *A_NAME* in Example 11.2.15b does not participate in fd's. One approach to remedy this situation is to introduce symbolic fd's. For instance, in that example one might include the fd *TITLE, A_NAME* $\to \omega_1$, where ω_1 is a new attribute. One relation produced by the algorithm will be $\{TITLE, A\_NAME, \omega_1\}$. As a last step, attributes such as ω_1 are removed.

In Example 11.2.9 we saw that the output of a BCNF decomposition may depend on the order in which fd's are applied. In the case of the preceding algorithm for 3NF, the minimal cover chosen greatly impacts the final result.

Mvd's and Decomposition into 4NF

A fundamental problem with BCNF decomposition and 3NF synthesis as just presented is that they do not take into account the impact of mvd's.

EXAMPLE 11.2.15 (a) The schema (*Studios*[*N*(*ame*), *D*(*irector*), *L*(*ocation*)], $\{N \twoheadrightarrow D|L\}$) is in BCNF and 3NF but has update anomalies. The mvd suggests a decomposition into ($\{Name, Director\}$, $\{Name, Location\}$).

(b) A related issue is that BCNF decompositions may not separate attributes that intuitively should be separated. For example, consider again the schema of Example 11.2.6, but suppose that the attribute *A_NAME* is included to denote actor names. Following the same decomposition steps as before, we obtain the schema

$$\left\{ \begin{array}{l} (\{TH\_NAME, ADDRESS, PHONE\}, \{FD1\}), \\ \\ (\{TITLE, D\_NAME\}, \{FD3\}), \\ \\ (\{TH\_NAME, TITLE, TIME, PRICE, A\_NAME\}, \{FD2\}) \end{array} \right\}$$

which can be further decomposed to

$$\left\{\begin{array}{l}(\{TH\_NAME, ADDRESS, PHONE\}, \{FD1\}),\\ (\{TITLE, D\_NAME\}, \{FD3\}),\\ (\{TH\_NAME, TITLE, TIME, PRICE\}, \{FD2\}),\\ (\{TH\_NAME, TITLE, TIME, A\_NAME\}, \emptyset)\end{array}\right\}$$

Although there is a connection in the underlying data between *TITLE* and *A_NAME*, the last relation here is unnatural. If we assume that the mvd $TITLE \twoheadrightarrow A\_NAME$ is incorporated into the original schema, we can further decompose the last relation and apply a step analogous to (2d) of the BCNF Decomposition Algorithm to obtain

$$\left\{\begin{array}{l}(\{TH\_NAME, ADDRESS, PHONE\}, \{FD1\}),\\ (\{TITLE, D\_NAME\}, \{FD3\}),\\ (\{TH\_NAME, TITLE, TIME, PRICE\}, \{FD2\}),\\ (\{TITLE, A\_NAME\}, \emptyset)\end{array}\right\}$$

Fourth normal form (4NF) was originally developed to address these kinds of situations. As suggested by the preceding example, an algorithm yielding 4NF decompositions can be developed along the lines of the BCNF Decomposition Algorithm. As with BCNF, the output of 4NF decomposition is a lossless join decomposition that is not necessarily dependency preserving.

A Note on Ind's

In relational schema design starting with a semantic data model, numerous ind's are typically generated. In contrast, the decomposition and synthesis approaches for refining relational schemas as presented earlier do not take ind's into account. It is possible to incorporate ind's into these approaches, but the specific choice of ind's is dependent on the intended semantics of the target schema.

Example 11.2.16 Recall the schema $(Movies[TDA], \{T \rightarrow D\})$ and decomposition into $(R_1[TD], \{T \rightarrow D\})$ and $(R_2[TA], \emptyset)$.

(a) If all movies must have a director and at least one actor, then both $R_1[T] \subseteq R_2[T]$ and $R_2[T] \subseteq R_1[T]$ should be included. In this case, the mapping from *Movies* to its decomposed representation is one-to-one and onto.

(b) If the fd $T \rightarrow D$ is understood to mean that there is a *total* function from movies to directors, but movies without actors are permitted, then the ind $R_2[T] \subseteq R_1[T]$ should be included.

(c) Finally, suppose the fd $T \rightarrow D$ is understood to mean that each movie has at most one director (i.e., it is a partial function), and suppose that a movie can have no actor. Then an additional relation $R_3[T]$ should be added to hold the titles of all movies, along with ind's $R_1[T] \subseteq R_3[T]$ and $R_2[T] \subseteq R_3[T]$.

More generally, what if one is to refine a relational schema $(\mathbf{R}, \Sigma \cup \Gamma)$, where Σ is a set of tagged fd's and mvd's and Γ is a set of ind's? It may occur that there is an ind $R_i[X] \subseteq R_j[Y]$, and either X or Y is to be "split" as the result of a decomposition step. The desired semantics of the target schema can be used to select between a variety of heuristic approaches to preserving the semantics of this ind. If Γ consists of unary ind's, such splitting cannot occur. Speaking intuitively, if the ind's of Γ are key based, then the chances of such splitting are reduced.

11.3 Universal Relation Assumption

In the preceding section, we saw that the decomposition and synthesis approaches to relational schema design assume the pure URA. This section begins by articulating some of the implications that underly the pure URA. It then presents the "weak URA," which provides an intuitively natural mechanism for viewing a relational database instance **I** as if it were a universal relation.

Underlying Assumptions

Suppose that an fd schema $(U[Z], \Sigma)$ is given and that decomposition or synthesis will be applied. One of several different database schemas might be produced, but presumably all of them carry roughly the same semantics. This suggests that the attributes in Z can be grouped into relation schemas in several different ways, without substantially affecting their underlying semantics. Intuitively, then, it is the attributes themselves (along with the dependencies in Σ), rather than the attributes as they occur in different relation schemas, that carry the bulk of the semantics in the schema. The notion that the attributes can represent a substantial portion of the semantics of an application is central to schema design based on the pure URA.

When decomposition and synthesis were first introduced, the underlying implications of this notion were not well understood. Several intuitive assumptions were articulated that attempted to capture these implications. We describe here two of the most important assumptions. Any approach to relational schema design based on the pure URA should also abide by these two assumptions.

Universal Relation Scheme Assumption: This states that if an attribute name appears in two or more places in a database schema, then it refers to the same entity set in each place. For example, an attribute name *Number* should not be used for both serial numbers and employee numbers; rather two distinct attribute names *Serial#* and *Employee#* should be used.

Unique Role Assumption: This states that for each set of attributes there is a unique rela-

tionship between them. This is sometimes weakened to say that there may be several relationships, but one is deemed primary. This is illustrated in the following example.

Example 11.3.1 (a) Recall in Example 11.2.15(b) that *D_NAME* and *A_NAME* were used for director and actor names, respectively. This is because there were two possible relationships between movies and persons.

(b) For a more complicated example, consider a schema for bank branches that includes attributes for *B*(*ranch*), *L*(*oan*), (*checking*) *A*(*ccount*), and *C*(*ustomer*). Suppose there are four relations

BL, which holds data about branches and loans they have given

BA, which holds data about branches and checking accounts they provide

CL, which holds data about customers and loans they have

CA, which holds data about customers and checking accounts they have.

This design does not satisfy the unique role assumption, mainly because of the cycle in the schema. For example, consider the relationship between branches and customers. In fact, there are two relationships—via loans and via accounts. Thus a request for "the" data in the relationship between banks and customers is somewhat ambiguous, because it could mean tuples stemming from either of the two relationships or from the intersection or union of both of them.

One solution to this ambiguity is to "break" the cycle. For example, we could replace the *Customer* attribute by the two attributes *L-C*(*ustomer*) and *A-C*(*ustomer*). Now the user can specify the desired relationship by using the appropriate attribute.

The Weak Universal Relation Assumption

Suppose that schema (U, Σ) has decomposition $(\mathbf{R}, \Gamma)$ (with $\mathbf{R} = \{R_1, \ldots, R_n\}$). When studying decomposition, we focused primarily on instances $\mathbf{I}$ of $(\mathbf{R}, \Gamma)$ that were the image of some instance I of (U, Σ) under the decomposition mapping $\pi_{\mathbf{R}}$. In particular, such instances $\mathbf{I}$ are *globally consistent*. [Recall from Chapter 6 that instance $\mathbf{I}$ is *globally consistent* if for each $j \in [1, n]$, $\pi_{R_j}(\bowtie \mathbf{I}) = \mathbf{I}(R_j)$; i.e., no tuple of $\mathbf{I}(R_j)$ is dangling relative to the full join.] However, in many practical situations it might be useful to use the decomposed schema $\mathbf{R}$ to store instances $\mathbf{I}$ that are not globally consistent.

Example 11.3.2 Recall the schema (*Movies*[*TDA*], $\{T \rightarrow D\}$) from Example 11.2.4 and its decomposition $\{TD, TA\}$. Suppose that for some movie the director is known, but no actors are known. As mentioned previously, this information is easily stored in the decomposed database, but not in the original. The impossibility of representing this information in the original schema was one of the anomalies that motivated the decomposition in the first place.

Suppose that fd schema (U, Σ) has decomposition $(\mathbf{R}, \Gamma) = \{(R_1, \Gamma_1), \ldots, (R_n, \Gamma_n)\}$. Suppose also that $\mathbf{I}$ is an instance of $\mathbf{R}$ such that (1) $\mathbf{I}(R_j) \models \Gamma_j$ for each j, but (2) $\mathbf{I}$ is

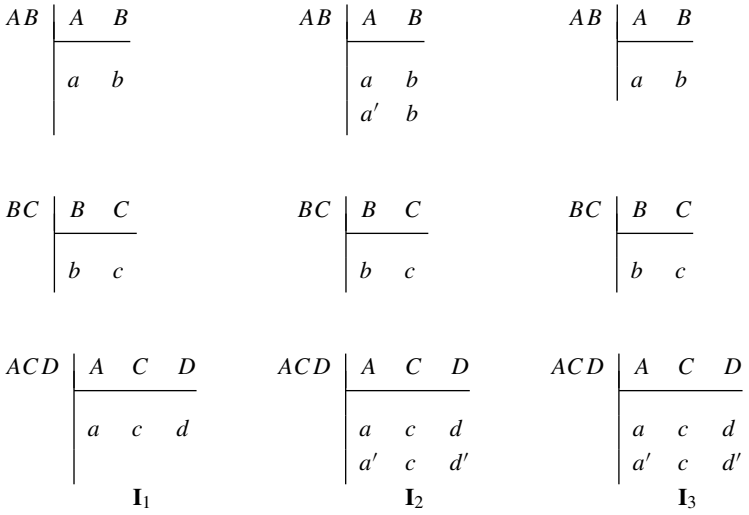


Figure 11.4: Instances illustrating weak URA

not necessarily globally consistent. Should $\mathbf{I}$ be considered a "valid" instance of schema $(\mathbf{R}, \Gamma)$? More generally, given a schema (U, Σ), a decomposition $\mathbf{R}$ of U, and a (not necessarily globally consistent) instance $\mathbf{I}$ over $\mathbf{R}$, how should we define the notion of "satisfaction" of Σ by $\mathbf{I}$?

The *weak universal relation assumption* (*weak URA*) provides one approach for answering this question. Under the weak URA, we say that $\mathbf{I}$ *satisfies* Σ if there is *some* instance $J \in sat(U, \Sigma)$ such that $\mathbf{I}(R_j) \subseteq \pi_{R_j}(J)$ for each $j \in [1, n]$. In this case, J is called a *weak instance* for $\mathbf{I}$.

Example 11.3.3 Let $U = \{ABCD\}$, $\Sigma = \{A \to B, BC \to D\}$, and $\mathbf{R} = \{AB, BC, ACD\}$. Consider the three instances of $\mathbf{R}$ shown in Fig. 11.4. The instance $\mathbf{I}_1$ satisfies Σ under the weak URA, because $J_1 = \{\langle a, b, c, d\rangle\}$ is a weak instance.

On the other hand, $\mathbf{I}_2$, which contains $\mathbf{I}_1$, does not satisfy Σ under the weak URA. To see this, suppose that J_2 is a weak instance for $\mathbf{I}_2$. Then J_2 must contain the following (not necessarily distinct) tuples:

$$t_1 = \langle a, b, c_1, d_1\rangle$$
$$t_2 = \langle a', b, c_2, d_2\rangle$$
$$t_3 = \langle a_3, b, c, d_3\rangle$$
$$t_4 = \langle a, b_4, c, d\rangle$$
$$t_5 = \langle a', b_5, c, d'\rangle$$

where the subscripted constants may be new. Because $J_2 \models A \to B$, by considering the

pairs $\langle t_1, t_4\rangle$ and $\langle t_2, t_5\rangle$, we see that $b_4 = b_5 = b$. Next, because $J_2 \models BC \rightarrow D$, and by considering the pair $\langle t_4, t_5\rangle$, we have that $d = d'$, a contradiction.

Finally, $\mathbf{I}_3$ does satisfy Σ under the weak URA.

As suggested by the preceding example, testing whether an instance $\mathbf{I}$ over $\mathbf{R}$ is a weak instance of (U, Σ) for a set of fd's Σ can be performed using the chase. To do that, it suffices to construct a table over U by padding the tuples from each R_j with distinct new variables. The resulting table is chased with the dependencies in Σ. If the chase fails, there is no weak instance for $\mathbf{I}$. On the other hand, a successful chase provides a weak instance for $\mathbf{I}$ by simply replacing each remaining variable with a distinct new constant.

This yields the following (see Exercise 11.27):

THEOREM 11.3.4 Let Σ be a set of fd's over U and $\mathbf{R}$ a decomposition of U. Testing whether $\mathbf{I}$ over $\mathbf{R}$ satisfies Σ under the weak URA can be performed in polynomial time.

Of course, the chasing technique can be extended to arbitrary egd's, although the complexity jumps to EXPTIME-complete.

What about full tgd's? Recall that full tgd's can always be satisfied by adding new tuples to an instance. Let Σ be a set of full dependencies. It is easy to see that $\mathbf{I}$ satisfies Σ under the weak URA iff $\mathbf{I}$ satisfies $\Sigma^* \cap \{\sigma \mid \sigma \text{ is an egd}\}$ under the weak URA.

Querying under the Weak URA

Let (U, Σ) be a schema, where Σ is a set of full dependencies, and let $\mathbf{R}$ be a decomposition of U. Let us assume the weak URA, and suppose that database instance $\mathbf{I}$ over $\mathbf{R}$ satisfies Σ. How should queries against $\mathbf{I}$ be answered? One approach is to consider the query against all weak instances for $\mathbf{I}$ and then take the intersection of the answers. That is,

$$q_{weak}(\mathbf{I}) = \cap\{q(I) \mid I \text{ is a weak instance of } \mathbf{I}\}.$$

We develop now a constructive method for computing q_{weak}.

Given instance $\mathbf{I}$ of $\mathbf{R}$, the *representative instance* of $\mathbf{I}$ is defined as follows: For each component I_j of $\mathbf{I}$, let I'_j be the result of extending I_j to be a free instance over U by padding tuples with distinct variables. Set $I' = \cup\{I'_j \mid j \in [1, n]\}$. Now apply the chase using Σ to obtain the representative instance $rep(\mathbf{I}, \Sigma)$ (or the empty instance, if two distinct constants are to be identified). Note that some elements of $rep(\mathbf{I}, \Sigma)$ may have variables occurring in them.

For $X \subseteq U$, let $\pi{\downarrow}_X(rep(\mathbf{I}, \Sigma))$ denote the set of tuples (i.e., with no variables present) in $\pi_X(rep(\mathbf{I}, \Sigma))$. The following can now be verified (see Exercise 11.28).

PROPOSITION 11.3.5 Let (U, Σ), $\mathbf{R}$ and $\mathbf{I}$ be as above, and let $X \subseteq U$. Then

(a) $[\pi_X]_{weak}(\mathbf{I}) = \pi{\downarrow}_X(rep(\mathbf{I}, \Sigma))$.

(b) If Σ is a set of fd's, then $[\pi_X]_{weak}(\mathbf{I})$ can be computed in PTIME.

This proposition provides the basis of a constructive method for evaluating an arbitrary algebra query q under the weak URA. Furthermore, if Σ is a set of fd's, then evaluating q will take time at most polynomial in the size of the input instance. This approach can be generalized to the case where Σ is a set of full dependencies but computing the projection is EXPTIME-complete.

Bibliographic Notes

The recent book [MR92] provides an in-depth coverage of relational schema design, including both the theoretical underpinnings and other, less formal factors that go into good design. Extensive treatments of the topic are also found in [Dat86, Fv89, Ull88, Vos91]. References [Ken78, Ken79, Ken89] illustrate the many difficulties that arise in schema design, primarily with a host of intriguing examples that show how skilled the human mind is at organizing diverse information and how woefully limiting data models are.

Surveys of semantic data models include [Bor85, HK87, PM88], and the book [TL82]; [Vos91] includes a chapter on this topic. Prominent early semantic data models include the Entity-Relationship (ER) model [Che76] (see also [BLN86, MR92, TYF86]), the Functional Data Model [Shi81, HK81], the Semantic Data Model [HM81], and the Semantic Binary Data Model [Abr74]. An early attempt to incorporate semantic data modeling constructs into the relational model is RM/T [Cod79]; more recently there have been various extensions of the relational model to incorporate object-oriented data modeling features (e.g., [SJGP90]). Many commercial systems support "tuple IDs," which can be viewed as a form of OID. Galileo [ACO85], Taxis [MBW80], and FQL [BFN82] are programming languages that support constructs stemming from semantic data models. The IFO [AH87] model is a relatively simple, formal semantic data model that subsumes the structural components of the aforementioned semantic models and several others. Reference [AH87] clarifies issues concerning ISA hierarchies in semantic schemas (see also [BLN86, Cod79, DH84] and studies the propagation of updates.

Reference [Che76] describes a translation of the ER model into the relational model, so that the resulting schema is in BCNF. From a practical perspective, this has become widely accepted as *the* method of choice for designing relational schemas; [TYF86] provides a subsequent perspective on this approach. There has also been considerable work on understanding the properties of relational schemas resulting from ER schemas and mapping relational schemas into ER ones. Reference [MR92] provides an in-depth discussion of this area.

Reference [LV87] presents a translation from a semantic to the relational model and studies the constraints implied for the relational schema, including cardinality constraints. The logical implication of constraints within a semantic model schema is studied in [CL94]. References [Lie80, Lie82] study the relationship of schemas from the network and relational models.

At a fundamental level, an important aspect of schema design is to replace one schema with another that can hold essentially the same information. This raises the issue of developing formal methods for comparing the *relative information capacity* of different schemas. Early work in this direction for the relational model includes [AABM82] and [BMSU81] (see Exercise 11.22). More abstract work is found in [HY84, Hul86] (see Exercises 11.23 and 11.24), which forms the basis for Theorem 11.1.3. Reference [MS92]

provides justification for translations from the Entity-Relationship model into the relational model using notions of relative information capacity. Formal notions of relative information capacity have also been applied in the context of schema integration and translation [MIR93] and heterogeneous databases [MIR94]. A very abstract framework for comparing schemas from different data models is proposed in [AT93].

The area of normal forms and relational database design was studied intensively in the 1970s and early 1980s. Much more complete coverage of this topic than presented here may be found in [Dat86, Mai83, Ull88, Vos91]. We mention some of the most important papers in this area. First normal form [Cod70] is actually fundamental to the relational model: A relation is in *first normal form* (1NF) if each column contains atomic values. In Chapter 20 this restriction shall be relaxed to permit relations some of whose columns themselves hold relations (which again may not be in first normal form). References [Cod71, Cod72a] raised the issue of update anomalies and initiated the search for normal forms that prevent them by introducing second and third normal forms. The definition of 3NF used here is from [Zan82]. (Second normal form is less restrictive than third normal form.) Boyce-Codd normal form (BCNF) was introduced in [Cod74] to provide a normal form simpler than 3NF. Another improvement of 3NF is proposed in [LTK81]. Fourth normal form was introduced in [Fag77b]; Example 11.2.15 is inspired from that reference. Even richer normal forms include project-join normal form (PJ/NF) [Fag79] and domain-key normal form [Fag81].

In addition to introducing second and third normal form, [Cod72a] initiated the search for normalization algorithms by proposing the first decomposition algorithms. This spawned other research on decomposition [DC72, RD75, PJ81] and synthesis [BST75, Ber76b, WW75]. The fact that these two criteria are not equivalent was stressed in [Ris77], where it is proposed that both be attempted. Early surveys on these approaches to relational design include [BBG78, Fag77a, Ris78]. Algorithms for synthesis into 3NF include [Ber76b, BDB79], for decomposition into BCNF include [TF82], and for decomposition into 4NF include [Fag77b]. Computational issues raised by decompositions are studied in [LO78, BB79, FJT83, TF82] and elsewhere. Reference [Got87] presents a good heuristic for finding covers of the projection of a set of fd's. The 3NF Synthesis Algorithm presented in this chapter begins with a *minimal cover* of a set of fd's; [Mai80] shows that minimal covers can be found in polynomial time.

The more formal study of decompositions and their properties was initiated in [Ris77], which considered decompositions into two-element sets and proposed the notion of independent components; and [AC78], which studied decompositions with lossless joins and dependency preservation. This was extended independently to arbitrary decompositions over fd's by [BR80] and [MMSU80]. Lossless join was further investigated in [Var82b] (see Exercise 11.20).

The notion that not all integrity constraints specified in a schema should be considered for the design process was implicit in various works on semantic data modeling (e.g., [Che76, Lie80, Lie82]). It was stated explicitly in connection with relational schema design in [FMU82, Sci81]. An extensive application of this approach to develop an approach to schema design that incorporates both fd's and mvd's is [BK86].

A very different form of decomposition, called *horizontal decomposition*, is introduced in [DP84]. This involves splitting a relation into pieces, each of which satisfies a given set of fd's.

The universal relation assumption has a long history; the reader is directed to [AA93, MUV84, Ull89b] for a much more complete coverage of this topic than found in this chapter. The URA was implicit in much of the early work on normal forms and decompositions; this was articulated more formally in [FMU82, MUV84]. The weak URA was studied in connection with query processing in [Sag81, Sag83], and in connection with fd satisfaction in [Hon82]. Proposition 11.3.5(a) is due to [MUV84] and part (b) is due to [Hon82]; the extension to full dependencies is due to [GMV86]. Reference [Sci86] presents an interesting comparison of the relational model with inclusion dependencies to a variant of the universal relation model and shows an equivalence when certain natural restrictions are imposed.

A topic related to the URA is that of *universal relation interfaces* (URI); these attempt to present a user view of a relational database in the form of a universal relation. An excellent survey of research on this topic is found in [MRW86]; see also [AA93, Osb79, Ull89b].

Exercises

Exercise 11.1

(a) Extend the instance of Example 11.1.1 for **CINEMA-SEM** so that it has at least two objects in each class.

(b) Let **CINEMA-SEM**′ be the same as **CINEMA-SEM**, except that a complex value class *Movie_Actor* is used in **CINEMA-SEM** in place of the attributes *acted_in* and *has_actors*. How would the instance you constructed for part (a) be represented in **CINEMA-SEM**′?

Exercise 11.2

(a) Suppose that in **CINEMA-SEM** some theaters do not have phones. Describe how the simulation **CINEMA-REL** can be changed to reflect this (without using null values). What dependencies are satisfied?

(b) Do the same for the case where some persons may have more than one citizenship.

Exercise 11.3

(a) Describe a general algorithm for translating GSM schemas with keys into relational ones.

(b) Verify Theorem 11.1.3.

(c) Verify that the relational schema resulting from a GSM schema is in 4NF and has acyclic and key-based ind's.

♠ **Exercise 11.4** [MR88, MR92] Let $\mathbf{R}$ be a relational database schema, Σ a set of tagged fd's for $\mathbf{R}$, and Γ a set of ind's for $\mathbf{R}$. Assume that $(\mathbf{R}, \Sigma)$ is in BCNF and that Γ is acyclic and consists of key-based ind's (as will arise if $\mathbf{R}$ is the simulation of a GSM schema). Prove that Σ and Γ are independent. *Hint:* Show that if $\mathbf{I}$ is an instance of $\mathbf{R}$ satisfying Σ, then no fd can be applied during chasing of $\mathbf{I}$ by $(\Sigma \cup \Gamma)$. Now apply Theorem 9.4.5.

Exercise 11.5 [Fag79] Let (R, Σ) be a relation schema, and let Σ' be the set of key dependencies implied by Σ. Show that R is in 4NF iff each nontrivial mvd implied by Σ is implied by Σ'.

Exercise 11.6 [DF92] A key dependency $X \to U$ is *simple* if X is a singleton.

(a) Suppose that (R, Σ) is in BCNF, where Σ may involve both fd's and mvd's. Suppose further that (R, Σ) has at least one simple key. Prove that (R, Σ) is in 4NF.

(b) Suppose that (R, Σ) is in 3NF and that each key of Σ is simple. Prove that (R, Σ) is in BCNF.

A schema (R, Σ) is in *project-join normal form* (*PJ/NF*) if each JD σ implied by Σ is implied by the key dependencies implied by Σ.

(a) Show that if (R, Σ) is in 3NF and each key of Σ is simple, then (R, Σ) is in PJ/NF.

Exercise 11.7 Let (U, Σ) be a schema, where Σ contains possibly fd's, mvd's, and jd's. Show that (a) (U, Σ) is in BCNF implies (U, Σ) is in 3NF; (b) (U, Σ) is in 4NF implies (U, Σ) is in BCNF; (c) (U, Σ) is in PJ/NF implies (U, Σ) is in 4NF.

Exercise 11.8 [BR80, MMSU80] Prove Theorem 11.2.3.

Exercise 11.9 Recall the schema (*Movies*[*TDA*],$\{T \to D\}$). Consider the decomposition $\mathbf{R}_1 = \{(TD, \{T \to D\}), (DA, \emptyset)\}$.

(a) Show that this does not have the lossless join property.

★(b) Show that this decomposition is not one-to-one. That is, exhibit two distinct instances I, I' of (*Movies*, $\{T \to D\}$) such that $\pi_{\mathbf{R}_1}(I) = \pi_{\mathbf{R}_1}(I')$.

Exercise 11.10 Verify Theorem 11.2.8. *Hint:* To prove the lossless join property, use repeated applications of Proposition 8.2.2.

Exercise 11.11 [FJT83] For each $n \geq 0$, describe an fd schema (U, Σ) and $V \subseteq U$, such that Σ has $\leq 2n + 1$ dependencies but the smallest cover for $\pi_V(\Sigma)$ has at least 2^n elements.

Exercise 11.12

(a) Let $(U[Z], \Gamma)$ be an fd schema. Give a polynomial time algorithm for determining whether this relation schema is in BCNF. (In fact, there is a linear time algorithm.)

(b) [BB79] Show that the following problem is co-NP-complete. Given fd schema $(R[U], \Sigma)$ and $V \subseteq U$, determine whether $(V, \pi_V(\Sigma))$ is in BCNF. *Hint:* Reduce to the hitting set problem [GJ79].

★**Exercise 11.13** [TF82] Develop a polynomial time algorithm for finding BCNF decompositions. *Hint:* First show that each two-attribute fd schema is in BCNF. Then show that if $(S[V], \Omega)$ is not in BCNF, then there are $A, B \in V$ such that $(V - AB) \to A$.

Exercise 11.14 Recall the schema *Showings*[*Th*(*eater*), *Sc*(*reen*), *Ti*(*tle*), *Sn*(*ack*)] of Section 8.1, which satisfies the fd *Th*,*Sc* $\to$ *Ti* and the mvd *Th* $\twoheadrightarrow$ *Sc*,*Ti* | *Sn*. Consider the two decompositions

$$\mathbf{R}_1 = \{\{Th, Sc, Ti\}, \{Th, Sn\}\}$$
$$\mathbf{R}_2 = \{\{Th, Sc, Ti\}, \{Th, Sc, Sn\}\}.$$

Are they one-to-one? dependency preserving? Describe anomalies that can arise if either of these decompositions is used.

Exercise 11.15 [BB79] Verify that the schema of Example 11.2.10 has no BCNF decomposition that preserves dependencies.

Exercise 11.16 [Mai80] Develop a polynomial time algorithm that finds a minimal cover of a set of fd's.

Exercise 11.17 Prove Theorem 11.2.14.

Exercise 11.18 [Mai83] Show that a schema $(R[U], \Sigma)$ with $2n$ attributes and $2n$ fd's can have as many as 2^n keys.

Exercise 11.19 [LO78] Let $(S[V], \Omega)$ be an fd schema. Show that the following problem is NP-complete: Given $A \in V$, is there a nontrivial fd $Y \to A$ implied by Ω, where Y is not a superkey and A is not a key attribute?

★ **Exercise 11.20** [Var82b] For this exercise, you will exhibit an example of a schema (R, Σ), where Σ consists of dependencies expressed in first-order logic (which may not be embedded dependencies) and a decomposition $\mathbf{R}$ of R such that $\mathbf{R}$ is one-to-one but does not have the lossless join property.

Consider the schema $R[ABCD]$. Given $t \in I \in inst(R)$, $t[A]$ is a *key element* for AB in I if there is no $s \in I$ with $t[A] = s[A]$ and $t[B] \neq s[B]$. The notion of $t[C]$ being a *key element* for CD is defined analogously. Let Σ consist of the constraints

(i) $\exists t \in I$ such that both $t[A]$ and $t[C]$ are key elements.

(ii) If $t \in I$, then $t[A]$ is a key element or $t[C]$ is a key element.

(iii) If $s, t \in I$ and $s[A]$ or $t[C]$ is a key element, then the tuple u is in I, where $u[AB] = s[AB]$ and $u[CD] = t[CD]$.

Let $\mathbf{R} = \{R_1[AB], R_2[CD]\}$ be a decomposition of (R, Σ).

(a) Show that the decomposition $\mathbf{R}$ for (R, Σ) is one-to-one.

(b) Exhibit a reconstruction mapping for $\mathbf{R}$. (The natural join will not work.)

Exercise 11.21 This and the following exercise provide one kind of characterization of the relative information capacity of decompositions of relation schemas. Let U be a set of attributes, let $\alpha = \{X_1, \ldots, X_n\}$ be a nonempty family of subsets of U, and let $X = \cup_{i=1}^n X_i$. The *project-join* mapping determined by α, denoted PJ_α, is a mapping from instances over U to instances over $\cup_{i=1}^n X_i$ defined by $PJ_\alpha(I) = \bowtie_{i=1}^n (\pi_{X_i}(I))$. α is *full* if $\cup_{i=1}^n = U$, in which case PJ_α is a *full* project-join mapping.

Prove the following for instances I and J over U:

(a) $\pi_X(I) \subseteq PJ_\alpha(I)$

(b) $PJ_\alpha(PJ_\alpha(I)) = PJ_\alpha(I)$

(c) if $I \subseteq J$ then $PJ_\alpha(I) \subseteq PJ_\alpha(J)$.

★ **Exercise 11.22** [BMSU81] Let U be a set of attributes. If $\alpha = \{X_1, \ldots, X_n\}$ is a nonempty full family of subsets of U, then $Fixpt(\alpha)$ denotes $\{I$ over $U \mid PJ_\alpha(I) = I\}$ (see the preceding exercise). For α and β nonempty full families of subsets of U, β *covers* α, denoted $\alpha \preceq \beta$, if for each set $X \in \alpha$ there is a set $Y \in \beta$ such that $X \subseteq Y$. Prove for nonempty full families α, β of subsets of U that the following are equivalent:

(a) $\alpha \preceq \beta$

(b) $PJ_\alpha(I) \supseteq PJ_\beta(I)$ for each instance I over U

(c) $Fixpt(\alpha) \subseteq Fixpt(\beta)$.

Exercise 11.23 Given relational database schemas $\mathbf{S}$ and $\mathbf{S}'$, we say that $\mathbf{S}'$ *dominates* $\mathbf{S}$ using the calculus, denoted $\mathbf{S} \preceq_{\text{calc}} \mathbf{S}'$, if there are calculus queries $q : Inst(\mathbf{S}) \to Inst(\mathbf{S}')$ and $q' : Inst(\mathbf{S}') \to Inst(\mathbf{S})$ such that $q \circ q'$ is the identity on $Inst(\mathbf{S})$. Let schema $R = (ABC, \{A \to B\})$ and the decomposition $\mathbf{R} = \{(AB, \{A \to B\}), (AC, \emptyset)\}$. (a) Verify that $R \preceq_{\text{calc}} \mathbf{R}$. (b) Show that $\mathbf{R} \not\preceq_{calc} R$. *Hint:* For schemas $\mathbf{S}$ and $\mathbf{S}'$, $\mathbf{S}'$ *dominates* $\mathbf{S}$ *absolutely*, denoted $\mathbf{S} \preceq_{abs} \mathbf{S}'$, if there is some $n \geq 0$ such that for each finite subset $\mathbf{d} \subseteq \mathbf{dom}$ with $|\mathbf{d}| \geq n$, $|\{\mathbf{I} \in Inst(\mathbf{S}) \mid adom(\mathbf{I}) \subseteq \mathbf{d}\}| \leq |\{\mathbf{I} \in Inst(\mathbf{S}') \mid adom(\mathbf{I}) \subseteq \mathbf{d}\}|$. Show that $\mathbf{S} \preceq_{calc} \mathbf{S}'$ implies $\mathbf{S} \preceq_{abs} \mathbf{S}'$. Then show that $\mathbf{R} \not\preceq_{abs} R$.

★ **Exercise 11.24** [HY84] Let A and B be relational attributes. Consider the complex value type $T = \langle A, \{B\}\rangle$, where each instance of T is a finite set of pairs having the form $\langle a, \hat{b}\rangle$, where $a \in \mathbf{dom}$ and $\hat{b}$ is a finite subset of $\mathbf{dom}$. Show that for each relational schema $\mathbf{R}$, $\mathbf{R} \preceq_{abs} T$ and $T \not\preceq_{abs} \mathbf{R}$. (See Exercise 11.23 for the definition of $\preceq_{abs}$.)

♠ **Exercise 11.25** [BV84b, CP84]

(a) Let (U, Σ) be a (full dependencies) schema and $\mathbf{R}$ an acyclic decomposition of U (in the sense of acyclic joins). Then $\pi_{\mathbf{R}}$ is one-to-one iff $\mathbf{R}$ has the lossless join property. *Hint:* First prove the result for the case where the decomposition has two elements (i.e., it is based on an mvd). Then generalize to acyclic decompositions, using an induction based on the GYO algorithm.

(b) [CKV90] Show that (a) can be generalized to include unary ind's in Σ.

Exercise 11.26 [Hon82] Let (U, Σ) be an fd schema and $\mathbf{R}$= $\{R_1, \ldots, R_n\}$ a decomposition of U. Consider the following notions of "satisfaction" by $\mathbf{I}$ over $\mathbf{R}$ of Σ:

$\mathbf{I} \models_1 \Sigma$**:** if $I_j \models \pi_{R_j}(\Sigma)$ for each $j \in [1, n]$.
$\mathbf{I} \models_2 \Sigma$**:** if $\bowtie \mathbf{I} \models \Sigma$.
$\mathbf{I} \models_3 \Sigma$**:** if $\mathbf{I} = \pi_{\mathbf{R}}(I)$ for some I over U such that $I \models \Sigma$.

(a) Show that $\models_1$ and $\models_2$ are incomparable.

(b) Show that if $\mathbf{R}$ preserves dependencies, then $\models_1$ implies $\models_2$.

(c) What is the relationship of $\models_1$ and $\models_2$ to $\models_3$?

(d) What is the relationship of all of these to the notion of satisfaction based on the weak URA?

♠ **Exercise 11.27** [Hon82] Prove Theorem 11.3.4.

Exercise 11.28 [MUV84, Hon82] Prove Proposition 11.3.5.