Mathematics of Information Processing
M. Anshel and W. Gewirtz, Editors
Proceedings of Symposia in Applied Mathematics
Volume 34, 1986, pp. 19-71
Amserican Mathematical Society
Providence, Rhode Island

THE THEORY OF DATA DEPENDENCIES - A SURVEY

Ronald Fagin and Moshe Y. Vardi

Abstract: The language of database dependencies can be seen as a language for specifying the semantics of databases. Dependency theory studies the properies of this language and its use in database management systems. We survey here three aspects of dependency theory: the *implication problem*, the *universal relation model*, and *acyclicity* of database schemes.

1. Introduction

In the *relational database model*, conceived by Codd in the late 60's [Co1], one views the database as a collection of relations, where each relation is a set of tuples over some domain of values. One notable feature of this model is its being almost devoid of semantics. A tuple in a relation represents a relationship between certain values, but from the mere syntactic definition of the relation one knows nothing about the nature of this relationship, not even if it is a one-to-one or one-to-many relationship.

One approach to remedy this deficiency is to devise means to specify the missing semantics. These semantic specifications are often called *semantic* or *integrity constraints*, since they specify which databases are meaningful for the application and which are meaningless. Of particular interest are the constraints called *data dependencies*, or dependencies for short.

1980 *Mathematical Subject Classification*. Primary 68B15, Secondary 03B25.

The study of dependencies began in 1972 with the introduction by Codd [Co2] of the *functional dependencies*. After the introduction, independently by Fagin and Zaniolo [Fa1,Za] in 1976, of *multivalued dependencies*, the field became chaotic for a few years in which researchers introduced many new classes of dependencies. The situation has stabilized since 1980 with the introduction, again independently by various researchers, of *embedded implicational dependencies* (EIDs). Essentially, EIDs are sentences in first-order logic stating that if some tuples, fulfilling certain equalities, exist in the database then either some other tuples must also exist in the database or some values in the given tuples must be equal. The class of EIDs seems to contain most previously studied classes of dependencies. (Recently, De Bra and Paredaens [DP] considered *afunctional dependencies*, which are not EIDs.) We give basic definitions and historical perspective in Section 2.

Most of the papers in dependency theory deal exclusively with various aspects of the *implication problem*, i.e., the problem of deciding for a given set of dependencies Σ and a dependency τ whether Σ *logically implies* τ. The reason for the prominence of this problem is that an algorithm for testing implication of dependencies enables us to test whether two given sets of dependencies are equivalent or whether a given set of dependencies is redundant. A solution for the last two problems seems a significant step towards automated database schema design, which some researchers see as the ultimate goal for research in dependency theory [BBG]. We deal with the implication problem in Section 3.

An emerging application for the theory of dependencies is the *universal relation model*. This model aims at achieving *data independence*, which was the original motivation for the relational model. In the universal relation model the user views the data as if it is stored in one big relation. The data, however, is not available in this form but rather in several smaller relations. It is the role of the database management system to provide the interface between the users' view and the actual data, and it is the role of the database designer to specify this interface. There have been different approaches to the question of what this interface should be like. We describe one approach, the *weak* universal relation approach, in Section 4.

One of the the major contributions of theoretical computer science is delineate the line between the computationally feasible and the infeasible.

The plethora of unsolvability and intractability results forces researchers to lower their aims, and to restrict themselves to "real-world" problems. That is, rather than trying to solve problems in their full generality, they try solve the cases that most often arise in practice. Research on *acyclicity* of database schemes falls in this vein. The idea is to view database schemes as *hypergraphs*. It turns out the acyclicity properties for hypergraphs translate into highly desirable database properties. Many problems that are intractable for general database schemes can be solved quite efficiently for acyclic schemes. Furthermore, there are arguments that many applications can be represented by acyclic schemes. We discuss acyclicity in Section 5.

A survey like ours of a rich theory necessarily has to be selective. The selection naturally reflects our tastes and biases. A more comprehensive, though less up to date, coverage can be found in the books [Ma,Ul].

2. Definitions and historical perspective

We begin with some fundamental definitions about relations. We are given a fixed finite set U of distinct symbols, called *attributes*, which are column names. From now on, whenever we speak of a set of attributes, we mean a subset of U. Let R be a set of attributes. An *R-tuple* (or simply a *tuple*, if R is understood) is a function with domain R. Thus, a tuple is a mapping that associates a value with each attribute in R. Note that under this definition, the "order of the columns" does not matter. If S is a subset of R, and if t is an R-tuple, then $t[S]$ denotes the S-tuple obtained by restricting the mapping to S. An *R-relation* (or a *relation over R*, or simply a *relation*, if R is understood), is a set of R-tuples. In database theory, we are most interested in *finite* relations, which are finite sets of tuples (although it is sometimes convenient to consider infinite relations). If I is an R-relation, and if S is a subset of R, then by $I[S]$, the *projection* of I onto S, we mean the set of all tuples $t[S]$, where t is in I. A *database* is a finite collection of relations.

Conventions: Upper-case letters A,B,C, ... from the start of the alphabet represent single attributes; upper-case letters $R,S,...,Z$ from the end of the alphabet represent sets of attributes; upper-case letters $I,J,...$ from the

middle of the alphabet represent relations; and lower-case letters $r,s,t,...$ from the end of the alphabet represent tuples.

Assume that relations $I_1,...,I_n$ are over attribute sets $R_1,...,R_n$ respectively. The *join* of the relations $I_1,...,I_n$, which is written either $\bowtie\{I_1,...,I_n\}$ or $I_1 \bowtie ... \bowtie I_n$, is the set of all tuples t over the attribute set $R_1 \cup ... R_n$, such that $t[R_i]$ is in I_i for each i. (Our notation exploits the fact that the join is associative and commutative.)

Certain sentences about relations are of special practical and/or theoretical interest for relational databases. For historical reasons, such sentences are usually called *dependencies*. The first dependency introduced and studied was the *functional dependency* (or FD), due to Codd [Co2]. As an example, consider the relation in Figure 2.1, with three columns: EMP (which represents employees), DEPT (which represents departments), and MGR (which represents managers). The relation in Figure 2.1 obeys the FD "DEPT→MGR", which is read "DEPT determines MGR". This means that whenever two tuples (that is, rows) agree in the DEPT column, then they necessarily agree also in the MGR column. The relation in Figure 2.2 does not obey this FD, since, for example, the first and fourth tuples agree in the DEPT column but not in the MGR column. We now give the formal definition. Let X and Y be subsets of the set U of attributes. The FD $X \rightarrow Y$ is said to hold for a relation I if every pair of tuples of I that agree on each of the attributes in X also agree in the attributes in Y.

The original motivation for introducing FDs (and some of the other dependencies we discuss) was to describe database *normalization*. Before giving an example of normalization, we need to define the notion of a relation scheme. A *relation scheme* is simply a set R of attributes. Usually, there is also an associated set Σ of sentences about relations over R. A relation is an *instance* of the relation scheme if it is over R and obeys the sentences in Σ. Thus, the sentences Σ can be thought of as "constraints", that every "valid instance" must obey. Although we do not do so, we note that it is common to define a relation scheme to be a pair $<R,\Sigma>$, where the constraints Σ are explicitly included.

We now consider an example of normalization. Assume that the attributes are {EMP,DEPT,MGR}, and that the only constraint is the FD DEPT→MGR. So, in every instance of this scheme, two employees in the

same department necessarily have the same manager. It might be better to store the data not in one relation, as in Figure 2.1, but rather in two relations, as in Figure 2.3: one relation that relates employees to departments, and one relation that relates departments to managers. We shall come back to normalization in Section 4.

It is easy to see that FDs can be represented as sentences in first-order logic [Nil]. Assume, for example, that we are dealing with a 4-ary relation, where the first, second, third, and fourth columns are called, respectively, A, B, C, and D. Then the FD $AB \rightarrow C$ is represented by the following sentence:

$$(\forall abc_1c_2d_1d_2)((Pabc_1d_1 \wedge Pabc_2d_2) \Rightarrow (c_1 = c_2)). \tag{2.1}$$

Here $(\forall abc_1c_2d_1d_2)$ is shorthand for $\forall a \forall b \forall c_1 \forall c_2 \forall d_1 \forall d_2$, that is, each variable is universally quantified. Unlike Nicolas, we have used individual variables rather than tuple variables. Incidentally, we think of P in (2.1) as a *relation symbol*, which should not be confused with an *instance* (that is, a *relation*) I, for which (2.1) can hold.

Let X and Y be sets of attributes (subsets of U), and let Z be U-XY (by XY, we mean $X \cup Y$). Thus, Z is the set of attributes not in X or Y. As we saw by example above (where X, Y, and Z are, respectively, the singleton sets {DEPT}, {EMP}, and {MGR}), the FD $X \rightarrow Y$ is a sufficient condition for a "lossless decomposition" of a relation with attributes U into two relations, with attributes XY and XZ respectively. This means that if I is a relation with attributes XYZ that obeys the FD $X \rightarrow Y$, then I can be obtained from its projections $I[XY]$ and $I[XZ]$, by joining them together. Thus, there is no loss of information in replacing relation I by the two relations I_1 and I_2. We note that this fact, which is known as Heath's Theorem [He], is historically one of the first theorems of database theory.

It may be instructive to give an example of a decomposition that does lose information. Let I be the relation in Figure 2.4, with attributes STORE, ITEM, and PRICE. Let I_1 and I_2 be two projections of I, onto {STORE, ITEM} and {ITEM, PRICE}, respectively, as in Figure 2.5. These projections contain less information than the original relation I. Thus, we see from relation I_1 that Macy's sells toasters; further, we see from relation I_2 that someone sells toasters for 20 dollars, and that some-

one sells toasters for 15 dollars. However, there is no way to tell from relations I_1 and I_2 how much Macy's sells toasters for.

The next dependency to be introduced was the *multivalued dependency,* or MVD, which was defined, independently by Fagin [Fa1] and Zaniolo [Za]. It was introduced because of the perception that the functional dependency provided too limited a notion of "depends on". As we shall see, multivalued dependencies provide a necessary and sufficient condition for lossless decomposition of a relation into two of its projections. Before we give the formal definition, we present a few examples. Consider the relation in Figure 2.6, with attributes EMP, SALARY, and CHILD. It obeys the functional dependency EMP→SALARY, that is, each employee has exactly one salary. The relation does *not* obey the FD EMP→CHILD, since an employee can have more than one child. However, it is clear that in some sense an employee "determines" his set of children. Thus, the employee's set of children is "determined by" the employee and by nothing else, just as his salary is. Indeed, as we shall see, the multivalued dependency EMP↠CHILD (read "employee multidetermines child") holds for this relation. As another example, consider the relation in Figure 2.7, with attributes EMP, CHILD, and SKILL. A tuple (e,c,s) appears in this relation if and only if e is an employee, c is one of e's children, and s is one of e's skills. This relation obeys no nontrivial (nontautologous) functional dependencies. However, it turns out to obey the multivalued dependencies EMP↠CHILD and EMP↠SKILL. Intuitively, the MVD EMP↠CHILD means that the set of names of the employee's children depends only on the employee, and is "orthogonal" to the information about his skills.

We are now ready to formally define multivalued dependencies. Let I be a relation over U. As before, let X and Y be subsets of U, and let Z be U-XY. The multivalued dependency $X \twoheadrightarrow Y$ holds for relation I if for each pair r, s of tuples of I for which $r[X] = s[X]$, there is a tuple t in I where (1) $t[X] = r[X] = s[X]$, (2) $t[Y] = r[Y]$, and (3) $t[Z] = s[Z]$. Of course, if this multivalued dependency holds for I, then it follows by symmetry that there is also a tuple u in I where (1) $u[X] = r[X] = s[X]$, (2) $u[Y] = s[Y]$, and (3) $u[Z] = r[Z]$.

Multivalued dependencies obey a number of useful properties. For example, if U is the disjoint union of X, Y, Z, and W, and if I is a relation over U that obeys the MVDs $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then it follows that I obeys

the MVD $X \twoheadrightarrow Z$ [Fa1]. So, MVDs obey a law of transitivity. We shall discuss more properties of MVDs in Section 3, where we give a complete axiomatization for MVDs.

Note that MVDs, like FDs, can be expressed in first-order logic. For example, assume that $U=\{A,B,C,D,E\}$. Then the MVD $AB \twoheadrightarrow CD$ holds for a relation over U if the following sentence holds, where P plays the role of the relation symbol:

$$(\forall abc_1c_2d_1d_2e_1e_2)\ ((Pabc_1d_1e_1 \wedge Pabc_2d_2e_2) \Rightarrow Pabc_2d_2e_1). \qquad (2.2)$$

Embedded dependencies were introduced (Fagin [Fa1]) as dependencies that hold in a projection of a relation (although, as we shall see, for certain classes of dependencies they are defined a little more generally). We shall simply give an example of an embedded MVD; the general case is obvious from the example. Assume that we are dealing with 4-ary relations, where we call the four columns $ABCD$. We say that such a 4-ary relation I obeys the *embedded MVD* (or *EMVD*) $A \twoheadrightarrow B \mid C$ if the projection of R onto ABC obeys the MVD $A \twoheadrightarrow B$. Thus, the EMVD $A \twoheadrightarrow B \mid C$ can be written as follows:

$$(\forall ab_1b_2c_1c_2d_1d_2)((Pab_1c_1d_1 \wedge Pab_2c_2d_2) \Rightarrow \exists d_3 Pab_1c_2d_3). \qquad (2.3)$$

As a concrete example, assume that the relation of Figure 2.7, with attributes EMP, CHILD, and SKILL, had an additional attribute BIRTHDATE, which tells the date of birth of the child. Then this 4-ary relation I would obey the embedded MVD EMP$\twoheadrightarrow$CHILD | SKILL. Note that I need not obey the MVD EMP$\twoheadrightarrow$CHILD (although it does obey the MVD EMP$\twoheadrightarrow${CHILD,BIRTHDATE}).

Several dependencies were defined within a few years after the multivalued dependency was introduced; we shall mention these other dependencies later in this section. Of these, the most important are the *join dependency*, or JD [ABU,Ri2]), and the *inclusion dependency*, or IND [Fa2]. Assume that $X=\{X_1, ..., X_k\}$ is a collection of subsets of U, where $X_1 \cup ... \cup X_k = U$. The relation I, over U, is said to obey the join dependency $\bowtie [X_1, ..., X_k]$, denoted also $\bowtie [X]$, if I is the join of its projections $I[X_1],...,I[X_k]$. It follows that this join dependency holds for the relation I if and only if I

contains each tuple t for which there are tuples $w_1, ..., w_n$ of I (not necessarily distinct) such that $w_i[X_i] = t[X_i]$ for each i $(1 \leq i \leq n)$. As an example, consider the relation I in Figure 2.8 below.

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

Figure 2.8

This relation violates the join dependency $\bowtie [AB, ACD, BC]$. For, let w_1, w_2, w_3 be, respectively, the tuples (0,1,0,0), (0,2,3,4), and (5,1,3,0) of I; let X_1, X_2, X_3 be, respectively, AB, ACD, and BC; and let t be the tuple (0,1,3,4); then $w_i[X_i] = t[X_i]$ for each i $(1 \leq i \leq n)$, although t is not a tuple in the relation I. However, it is straightforward to verify that the same relation I obeys, for example, the join dependency $\bowtie [ABC, BCD, ABD]$.

Let us say that the join dependency $\bowtie [X_1, ..., X_k]$ has k *components*. Join dependencies are generalizations of multivalued dependencies; thus, each multivalued dependency is equivalent to a join dependency with two components, and conversely. Assume now that $X_1 \cup ... \cup X_k \subseteq U$, and denote $X_1 \cup ... \cup X_k$ by X. A relation I with attributes U is said to obey the *embedded join dependency* $\bowtie [X_1, ..., X_k]$ if its projection $I[X]$ obeys the join dependency $\bowtie [X_1, ..., X_k]$. We shall see soon that join dependencies can be written in first-order logic. Embedded join dependencies, too, can be so written, but they require existential quantifiers, just as embedded multivalued dependencies do. Note that our notation, the set U of attributes does not appear, and so the same syntactical object $\bowtie [X_1, ..., X_k]$ is used to represent a join dependency over X and an embedded join dependency over U. However, the two would be written in distinct ways in first-order logic. This is actually a nice convenience, especially in the case of functional dependencies, where a similar comment applies.

The intuitive semantics of *multivalued* dependencies were fairly well understood at the time they were first defined. However, it was not until several years after *join* dependencies were defined that their semantics was

adequately explained (by Fagin et al. [FMU]). Let us consider an example (from [FMU]). Assume that the attributes are *C*(ourse), *T*(eacher), *R*(oom), *H*(our), *S*(tudent), and *G*(rade). The informal meaning of these attributes is that teacher *T* teaches course *C*, course *C* meets in room *R* at hour *H*, and that student *S* is getting grade *G* in course *C*. If we were to define a single "universal" relation over these attributes, it would be

$\{(c,t,r,h,s,g)$: t "teaches" c; c "meets in" r "at hour" h; and s " is getting" g "in" $c\}$.

This relation is of the form

$$\{(c,t,r,h,s,g)\colon P_1tc \wedge P_2crh \wedge P_3sgc\}, \tag{2.4}$$

for certain predicates P_1, P_2, and P_3. The fact that a relation with attributes c,t,r,h,s,g is of the form (2.4) for for some predicates P_1, P_2, and P_3 is a severe constraint. In fact [FMU], this constraint is precisely equivalent to the join dependency $\bowtie[TC,CRH,SGC]$. The obvious generalization of this observation to arbitrary join dependencies explains their semantics. Before we leave join dependencies, let us note, as promised, they, too, can be written as sentences in first-order logic. For example, if we are dealing with relations with attributes c,t,r,h,s,g, then the join dependency $\bowtie[TC,CRH,SGC]$ can be written as

$$(\forall c t_1 t_2 r r_1 r_2 h h_1 h_2 s s_1 s_2 g g_1 g_2)$$
$$((Pctr_1h_1s_1g_1 \wedge Pct_1rhs_2g_2 \wedge Pct_2r_2h_2sg) \Rightarrow Pctrhsg) \tag{2.5}$$

So far, each of the dependencies we have discussed has two properties: (1) each is *uni-relational*, that is, deals with a single relation at a time, rather than with inter-relationships among several relations, and (2) each is typed. By *typed*, we mean that no variable appears in two distinct columns. For example, the sentence $(\forall xy)((Pxy \wedge Pyz) \Rightarrow Pxz)$, which says that a relation is transitive, is *not* typed, since the variable y appear in both the first and second columns of P in the sentence. The next dependency that we shall discuss violates both (1) and (2) above, that is, is neither uni-relational nor typed. This dependency is the *inclusion dependency*, or IND [CFP]. As an example, an IND can say that every MANAGER entry of the P relation appears as an EMPLOYEE entry of the Q relation. In general, an IND is of the form

$$P[A_1 \ldots A_m] \subseteq Q[B_1 \ldots B_m], \tag{2.6}$$

where P and Q are relation names (possibly the same), and where the A_i's and B_i's are attributes. If I is the P relation and J is the Q relation, then the IND (2.6) holds if for each tuple s of I, there is a tuple t of J such that $s[A_1 \ldots A_m] = t[B_1 \ldots B_m]$. Hence, INDs are valuable for database design, since they permit us to selectively define what data must be duplicated in what relations. INDs are commonly known in Artificial Intelligence applications as *ISA* relationships (cf. Beeri and Korth [BK]). Not surprisingly, the inclusion dependency, too, can be written in first-order logic. For example, if the P relation has attributes ABC, and the Q relation has attributes CDE, then the IND $P[AB] \subseteq Q[CE]$ can be written

$$(\forall abc)(Pabc \Rightarrow \exists d Qadb). \tag{2.7}$$

After multivalued dependencies were defined, there was a period where a large number of other dependencies were defined. We have already discussed the classes of join dependencies, embedded join dependencies, and inclusion dependencies. Others (many of which were introduced before join dependencies) include Nicolas's *mutual dependencies* [Ni1], which say that a relation is the join of three of its projections, Mendelzon and Maier's *generalized mutual dependencies* [MM], Paredaens' *transitive dependencies* [Pa], which generalize both FDs and MVDs, Ginsburg and Zaiddan's *implied dependencies* [GZ], which generalize FDs, Sagiv and Walecka's *subset dependencies* [SW], which generalize embedded MVDs, Sadri and Ullman's and Beeri and Vardi's *template dependencies* ([SU], [BV4]) which generalize embedded join dependencies, and Parker and Parsaye-Ghomi's *extended transitive dependencies* [PP], which generalize both mutual dependencies and transitive dependencies. We remark that the last 3 kinds of dependencies mentioned were introduced to deal with the issue of a complete axiomatization (see Section 3): subset dependencies were introduced to show the difficulty of completely axiomatizing embedded multivalued dependencies; extended transitive dependencies were introduced to show the difficulty of completely axiomatizing transitive dependencies; while template dependencies were introduced to provide a class of dependencies that include join dependencies and that can be completely axiomatized. Inclusion dependencies, which had been used informally for databases by many practitioners, were not seriously studied until relatively late [CFP].

Various researchers finally realized that all of these different types of dependencies can be united into a single class, which we shall call simply *dependencies*. Before we can define them formally, we need a few preliminary concepts. We assume that we are given a set of *individual variables* (which represent entries in a relation of a database). The *atomic* formulas are those that are either of the form $Pz_1...z_d$ (where P is the name of a d-ary relation, and where the z_i's are individual variables), or else of the form $x=y$ (where x and y are individual variables). Atomic formulas $Pz_1...z_d$ we call *relational formulas*, and atomic formulas $x=y$ we call *equalities*. A *dependency* is a first-order sentence

$$(\forall x_1...x_m)((A_1 \wedge ... \wedge A_n) \Rightarrow \exists y_1...y_r(B_1 \wedge ... \wedge B_s)), \tag{2.8}$$

where each A_i is a relational formula and where each B_i is atomic (either a relational formula or an equality). We assume also that each of the x_j's appears in at least one of the A_i's, and that $n \geq 1$, that is, that there is at least one A_i. We assume that $r \geq 0$ (if $r=0$ then there are no existential quantifiers), and that $s \geq 1$ (that is, there must be at least one B_i.) Note that because of all these assumptions, each dependency is obeyed by an empty database with no tuples. Furthermore, our assumptions guarantee that we can tell if a dependency holds for a relation by simply considering the collection of tuples of the relation, and ignoring any underlying "domains of attributes". Intuitively, in considering whether a dependency holds for a relation, the quantifiers can be assumed to range over the elements that appear in the relation, and not over any larger domain. This property is called *domain independence*. See Fagin [Fa4] for a much more complete discussion of domain independence.

If each of the formulas B_i on the right-hand side of (2.8) is a relational formula, then we call the dependency a *tuple-generating dependency*; if all of these formulas are equalities, then we call the dependency an *equality-generating dependency*. Of the dependencies we have focused on above, the (embedded) multivalued dependency, the (embedded) join dependency, and the inclusion dependency are each tuple-generating dependencies; thus, each of the first-order sentences (2.2), (2.3), (2.5), and (2.7) above represent tuple-generating dependencies. Tuple-generating dependencies say that if a certain pattern of entries appears, then another pattern must appear. Functional dependencies, as we see by example in the sentence (2.1) above, are equality-generating dependencies. Equality-generating

dependencies say that if a certain pattern of entries appears, then a certain equality must hold. A *full* dependency is one in which r=0 and s=1 in (2.8), that is, one in which there are no existential quantifiers and in which there is only one atomic formula B_i on the right-hand side. Thus, a full dependency is of the form

$$(\forall x_1 \ldots x_m)((A_1 \wedge \ldots \wedge A_n) \Rightarrow B), \tag{2.9}$$

where each A_i is a relational formula, where B is atomic. Functional, multivalued, and join dependencies are all full dependencies. We may refer to a dependency (2.8) as an *embedded dependency*, to emphasize that we are allowing (but not requiring) existential quantifiers. Note that in the case of full dependencies, we would not gain anything by allowing the possibility of having several atomic formulas on the right-hand side, since such a sentence is equivalent to a finite set of full dependencies as we have defined them.

The class of dependencies was defined independently by a number of authors, who usually focussed on the uni-relational case. (Note that the only special case of a dependency that we have mentioned so far that is not uni-relational is the inclusion dependency.) Beeri and Vardi [BV7] refer to this class as the class of all *tuple-generating* and *equality-generating dependencies.* Fagin [Fa4] focused on the typed, uni-relational case, which he called *embedded implicational dependencies* (with the full dependencies being called *implicational dependencies*). Yannakakis and Papadimitriou [YP] defined *algebraic dependencies*, which are built out of expressions involving projection and join, and which, on the surface, look very different from our first-order definition. It is somewhat surprising that their class (which is typed) turns out [YP] to be identical to our typed, uni-relational dependencies. Paredaens and Janssens [PJ] defined *general dependencies*, which are full, typed, uni-relational dependencies. Also, Grant and Jacobs [GJ] defined *generalized dependency constraints,* which are full dependencies.

An often heard claim is that in the "real world" one rarely encounters dependencies in their most general form. According to this claim FDs, INDs, maybe MVDs are the only kinds of dependencies that earn the title "real world dependencies". We have two answers to this claim. First, we believe that there are real-world situations that do require the more general dependencies. Even when the database itself can be specified by FDs,

user views of this database may not be specifiable by FDs [Fa4, GZ]. Furthermore, even if only simple dependencies arise in practice, the more general dependencies are very useful theoretically. For example, statements about equivalence of queries can be expressed by dependencies [YP]. We refer the reader to [Hu, Va3, Va5] for more examples of the latter argument.

3. The Implication Problem

3.1. Implication and finite implication

Logical implication is a fundamental notion in logic. Let Σ be a set of sentences, and let τ be a single sentence. We say that Σ *implies* τ, denoted $\Sigma \models \tau$, if every model of Σ is also a model of τ. In our context, $\Sigma \models \tau$ if every database that satisfies all dependencies in Σ satisfies also τ. For example $\{A \rightarrow B, B \rightarrow C\} \models A \rightarrow C$.

The relevance of implication to database theory became apparent in Bernstein's work on synthesis of database schemes using FDs [Ber]. Let Σ_1 and Σ_2 be sets of dependencies. We say that Σ_1 is *equivalent to* Σ_2, denoted $\Sigma_1 \equiv \Sigma_2$, if every database that satisfies all dependencies in Σ_1 also satisfies all dependencies in Σ_2 and vice versa. We say that Σ_1 is *redundant* if $\Sigma_2 \subset \Sigma_1$ and $\Sigma_1 \equiv \Sigma_2$. (We use $\subseteq$ to denote containment and $\subset$ to denote proper containment.) Clearly, Σ is redundant if there is some $\tau \in \Sigma$ such that $\Sigma - \{\tau\} \models \tau$. Since Bernstein's synthesis algorithm requires eliminating redundant FDs , and since the problem of eliminating redundant dependencies can be reduced to the problem of testing implication of dependencies, the notion of implication became a central notion to dependency theory. The significance of implication was reconfirmed in later works, e.g., [BMSU,Ri2].

In database theory we often like to restrict our attention to finite databases, since in practice databases are finite. We say that Σ *finitely implies* τ, denoted $\Sigma \models_f \tau$, if every finite database that satisfies Σ satisfies also τ. Clearly, if $\Sigma \models \tau$ holds then $\Sigma \models_f \tau$ also holds. But it is possible that $\Sigma \models_f \tau$ holds while $\Sigma \models \tau$ does not. That is, it is possible that every finite database that satisfies Σ satisfies also τ, but there is an *infinite* database that satisfies Σ but not τ. Implication and finite implication lead to two

decision problems. The *implication problem* is to decide, for a given set Σ of dependencies and a single dependency τ, whether $\Sigma \models \tau$. The *finite implication problem* is to decide, for a given set Σ of dependencies and a single dependency τ, whether $\Sigma \models_f \tau$.

Let $\Sigma = \{\sigma_1,...,\sigma_n\}$. Then $\Sigma \models \tau$ ($\Sigma \models_f \tau$) if and only if $\sigma_1 \wedge ... \wedge \sigma_n \wedge \neg\tau$ is (finitely) unsatisfiable. (A sentence is (finitely) satisfiable if it has a (finite) model. It is (finitely) unsatisfiable if it has no (finite) model.) Since unsatisfiability is known to be recursively enumerable (Gödel's Completeness Theorem), and finite satisfiability is clearly recursively enumerable, it follows that the relationships $\models$ and $\not\models_f$ are recursively enumerable. Suppose now that for some class of dependencies $\models$ and $\models_f$ are the same. Then $\models$ and $\not\models_f$ complement each other and they are both recursively enumerable. It follows that they are both recursive [Ro]. Indeed, the standard technique for proving solvability of the implication problem is to show that implication and finite implication coincide.

Dependencies are $\forall^*\exists^*$ sentences, i.e., they are equivalent to sentences whose quantifier prefix consists of a string of universal quantifiers followed by a string of existential quantifiers. Thus, $\sigma_1 \wedge ... \wedge \sigma_n \wedge \neg\tau$ is a $\exists^*\forall^*\exists^*$ sentence. When Σ, however, consists of full dependencies, then $\sigma_1 \wedge ... \wedge \sigma_n \wedge \neg\tau$ is an $\exists^*\forall^*$ sentence. Thus, the (finite) implication problem for full dependencies is reducible to the (finite) satisfiability problem for $\exists^*\forall^*$ sentences. This class of sentences is known as the *initially extended Bernays-Schönfinkel class.* For this class, satisfiability and finite satisfiability coincide, and therefore both are recursive [DG]. Thus, for full dependencies, implication and finite implication coincide, and are recursive. Unfortunately, the satisfiability problem for the Bernays-Schönfinkel class require nondeterministic exponential time [Le], and hence is highly intractable. Since the class of full dependencies is a proper subset of the class of universal sentences, one may hope that the implication problem for full dependencies is not that hard. We study this problem in Section 3.2.

For simplicity we restrict ourselves in the sequel to uni-relational dependencies, i.e., dependencies that refer to a single relation.

3.2. The implication problem for full dependencies

Since for full dependencies implication and finite implication coincide, everything we say in this section about implication holds, of course, for finite implication as well.

Even though the significance of implication was not yet clear in 1974, it was studied by Armstrong [Arm], apparently just out of mathematical interest. Armstrong characterized implication of FDs using an *axiom system.* An axiom system consists of *axiom schemes* and *inference rules.* A *derivation* of a dependency τ from a set Σ of dependencies is a sequence $\tau_1,\tau_2,\ldots,\tau_n$, where τ_n is τ and each τ_i is either an instance of an axiom scheme or follows from preceding dependencies in the sequence by one of the inference rules. $\Sigma \vdash \tau$ denotes that there is a derivation of τ from Σ. An axiom system is *sound* if $\Sigma \vdash \tau$ entails $\Sigma \models \tau$, and it is *complete* if $\Sigma \models \tau$ entails $\Sigma \vdash \tau$. Armstrong's system, denoted $\mathcal{FD}$, consists of one axiom and three inference rules:

FD0 (reflexivity axiom): $\vdash X \rightarrow X$.
FD1 (transitivity): $X \rightarrow Y,\ Y \rightarrow Z \vdash X \rightarrow Z$.
FD2 (augmentation and projection): $X \rightarrow Y \vdash W \rightarrow Z$, if $X \subseteq W$ and $Y \supseteq Z$.
FD3 (union): $X \rightarrow Y,\ Z \rightarrow W \vdash XZ \rightarrow YW$.

Theorem 3.2.1. [Arm] The system $\mathcal{FD}$ is sound and complete for implication of FDs. □
(In fact, Armstrong proved a somewhat stronger result, which we shall not discuss here. See [Fa3].)

Armstrong did not consider the algorithmic aspects of his axiom system. This was done by Beeri and Bernstein [BB], who were motivated by the fact that one of the steps in Bernstein's synthesis algorithm [Ber] is a test for implication. They were the first to phrase the implication problem. (Beeri and Bernstein called it the *membership problem.* In some papers it is also called the *inference problem.)*

Let Σ be a set of dependencies, and let X be a set of attributes. The *closure* of X with respect to Σ is the set of all attributes functionally determined by X, that is, $\{A: \Sigma \models X \rightarrow A\}$. Clearly, once we know the closure of X with respect to Σ, we can find out easily whether $\Sigma \models X \rightarrow A$. Beeri and Bernstein showed that the system $\mathcal{FD}$ can be used to construct closures very fast.

Theorem 3.2.2. [BB] The implication problem for FDs can be solved in time $O(n)$, where n is the length of the input. □

A large part of dependency theory since 1976 was devoted to studying these two aspects of implication, i.e., axiomatization and complexity of the implication problem. For example, shortly after the introduction of MVDs in 1976, they were axiomatized by Beeri et al. [BFH], and Beeri proved that implication problem is solvable [Bee]. Both works tried to get results analogous to the results for FDs.

The axiom system $\mathcal{MVD}$ consists of one axiom and three inference rules:

MVD0 (reflexivity axiom): $\vdash X \twoheadrightarrow Y$, if $Y \subseteq X$.
MVD1 (transitivity): $X \twoheadrightarrow Y,\ Y \twoheadrightarrow Z \vdash X \twoheadrightarrow Z\text{-}Y$.
MVD2 (augmentation): $X \twoheadrightarrow Y \vdash XW \twoheadrightarrow YZ$ if $Z \subseteq W$.
MVD3 (complementation): $X \twoheadrightarrow Y \vdash X \twoheadrightarrow Z$, if $XYZ = U$ and $Y \cap Z \subseteq X$.

Theorem 3.2.3. [BFH] The system $\mathcal{MVD}$ is sound and complete for implication of MVDs. □

We note that Beeri et al. [BFH] also present a sound and complete axiomatization for FDs and MVDs taken together. This axiomatization contains all of the axiom schemes and inference rules for FDs and MVDs separately that we have already seen, along with two "mixed" rules, that account for the interaction of FDs and MVDs.

The analogue of closure of an attribute set X is now not an attribute set but rather a collection of attribute sets: $rhs_\Sigma(X) = \{Y\colon\ \Sigma \models X \twoheadrightarrow Y\}$. Now $rhs_\Sigma(X)$ can contain exponentially many sets, and hence is not very useful algorithmically. However, using the system $\mathcal{MVD}$ it is not hard to verify that $rhs_\Sigma(X)$ is a Boolean algebra. Furthermore, since it is a field of finite sets, it is an *atomic* Boolean algebra, and every every element is the union of the atoms it contains. The set of atoms of this Boolean algebra is called the *dependency basis* of X with respect to Σ, denoted $dep_\Sigma(X)$. Thus $dep_\Sigma(X) = \{Y\colon\ Y \neq \emptyset,\ \Sigma \models X \twoheadrightarrow Y$, and if $\Sigma \models X \twoheadrightarrow Z$, $Z \subseteq Y$, and $Z \neq \emptyset$, then $Z = Y\}$.

Lemma 3.2.4. [BFH] $dep_\Sigma(X)$ is a partition of U. Furthermore, $\Sigma \models X \twoheadrightarrow Y$ if and only if there are sets $W_1,...,W_m$ in $dep_\Sigma(X)$ such that $Y = W_1 \cup ... \cup W_m$.

Beeri [Bee] has shown how $dep_\Sigma(X)$ can be constructed efficiently using the system $\mathcal{MVD}$.

Theorem 3.2.5. [Bee] The implication problem for MVDs can be solved in time $O(n^4)$, where n is the length of the input. □

Beeri's algorithm was improved by Hagihara et al. [HITK], Sagiv [Sag1], and finally by Galil [Ga]. Galil's algorithm runs in time $O(n \log n)$. These papers and [Bee,BFH] discuss also the interaction of FDs and MVDs.

It is easy to see that Lemma 3.2.4 does not depend on Σ being a set of MVDs. Thus, testing whether an MVD $X \twoheadrightarrow Y$ is implied by a set Σ of dependencies can be done efficiently as long as $dep_\Sigma(X)$ can be constructed efficiently. This was shown in [MSY,Va4] to be the case when Σ is a set of JDs and FDs, and in [Va1] for the case when Σ is a set of typed full dependencies.

Theorem 3.2.6. [Va1] Testing whether an MVD or an FD is implied by a set of typed full dependencies can be done in time $O(n^2)$, where n is the length of the input. □

Let us refer now to implication of JDs. Aho et al. [ABU] described an algorithm, called later the *chase*, to test implication of JDs by FDs.

Theorem 3.2.7. [ABU] Testing whether a JD is implied by a set of FDs can be done in time $O(n^4)$, where n is the length of the input. □

More efficient implementations of the chase were described by Liu and Demers [LD] and by Downey et al. [DST]. The latter algorithm runs in time $O(n^2 \log^2 n)$.

The ideas in [ABU] were generalized by Maier et al. [MMS] to deal with arbitrary implication of FDs and JDs.

Theorem 3.2.8. [MMS] The implication problem for FDs and JDs is solvable in time $O(n^n)$, where n is the length of the input. □

The question then arose whether the exponential upper bound of the above theorem can be improved. Unfortunately, Theorems 3.2.6 and 3.2.7 probably describe the most general case for which an efficient decision procedure exists. Recall that a problem is NP-hard if it is as hard as any problem that can be solved in nondeterministic polynomial time. A problem is NP-complete if it is NP-hard and it can be solved in nondeterministic polynomial time. It is believed that NP-hard problems can not be solved efficiently, i.e., in polynomial time. ([GJ] is a good textbook on the theory of NP-completeness.) Thus, proving that a problem is NP-hard is a strong indication that the problem is computationally intractable.

Theorem 3.2.9.
1) [FT] Testing whether a set of MVDs implies a JD is NP-hard.
2) [BV3] Testing whether a JD and an FD imply a JD is NP-complete. □

Thus, we know how to test implication of FDs and JDs in exponential time, and we know that the problem is NP-hard. We do not know, however, how to pinpoint the complexity of this problem. We do not know for example whether testing implication of a JD by a set of MVDs can be done in nondeterministic polynomial time. One approach to the problem was to try to find a axiom system for FDs and JDs. Surprisingly, even for JDs alone finding a axiom system is extremely difficult (see [BV1,BV5,Sc3]).

NP-completeness strongly suggests, rather than proves, that a problem is intractable (i.e., it proves intractability under the assumption that there are problems that can be solved in nondeterministic polynomial time but not in deterministic polynomial time). In contrast, EXPTIME-completeness is a proof that a problem is intractable. A problem is EXPTIME-complete if it can be solved in exponential time and it is also as hard as any problem that can be solved in exponential time. Since it is known that there are problems that can be solved in exponential time and in fact do require exponential time, it follows that EXPTIME-complete problems require exponential time.

Theorem 3.2.10. [CLM2] The implication problem for typed full dependencies is EXPTIME-complete. □

Interestingly, Beeri and Vardi presented an elegant axiom system for typed full dependencies [BV4]. This demonstrates that there is no clear relation-

ship between having an axiom system for a class of dependencies and the complexity of the implication problem for that class.

In conclusion of this section, the reader should keep in mind that the above lower bounds describe a worst-case behavior of the problems. It is not clear at all that this worst-case behavior indeed arise in practice. We shall return to this point in Section 5.

3.3. The implication problem for embedded dependencies

While for full dependencies the implication problem is clearly solvable and the questions to answer involve upper and lower bounds, this is not so with embedded dependencies, since satisfiability and finite satisfiability do not coincide for the class of $\exists^*\forall^*\exists^*$ sentences, and the corresponding problems are both unsolvable [DG]. Thus, we have to deal here with both implication and finite implication and their corresponding decision problem. Since the class of dependencies is a proper subset of the class of $\forall^*\exists^*$ sentences, one may hope that the (finite) implication problem for embedded dependencies is solvable.

The first disappointing observation is that implication and finite implication do not coincide for embedded dependencies.

Theorem 3.3.1. [CFP,JK] There is a set Σ of FDs and INDs and a single IND τ such that $\Sigma \models_f \tau$, but $\Sigma \not\models \tau$.

Proof: (a) Let Σ be $\{A \rightarrow B, A \subseteq B\}$, and let τ be $B \subseteq A$. We first show that $\Sigma \models_f \tau$. Let I be a finite relation satisfying Σ. We now show that I satisfies τ, that is, $I[B] \subseteq I[A]$. Since I satisfies $A \rightarrow B$ it follows that $|I[B]| \leq |I[A]|$. Since $I[A] \subseteq I[B]$, it follows that $|I[A]| \leq |I[B]|$. Thus, $|I[A]| = |I[B]|$. But since $I[A] \subseteq I[B]$ and since both $I[A]$ and $I[B]$ are finite, we than have $I[B] = I[A]$, so $I[B] \subseteq I[A]$. This was to be shown.

To show that $\Sigma \not\models \tau$, we need only exhibit a relation (necessarily infinite) that satisfies Σ but not τ. Let I be the relation with tuples $\{(i+1,i): i \geq 0\}$. It is obvious that I satisfies Σ but not τ. □

One may think that this behavior is the result of the interaction between tuple-generating dependencies and equality-generating dependencies, but

an example in [BV7] shows that even for tuple-generating dependencies the two notions of implication and finite implication differ.

The simplest instance of embedded dependencies are the EMVDs. The (finite) implication problem for EMVDs has resisted efforts of many researchers, and is one of the most outstanding open problems in dependency theory. A significant part of the research in this area has been motivated by this problem. For example, underlying the search for bigger and bigger classes of dependencies was the hope that for the larger class a decision procedure would be apparent, while the specialization of the algorithm to EMVDs was too murky to be visible. Also, underlying the work on axiomatization was the hope that an axiom system may lead to a decision procedure just as the axiom systems for FDs and MVDs led to decision procedures for these classes of dependencies.

Maier et. al [MMS] suggested an extension of the chase to deal with EJDs, and this was further generalized by Beeri and Vardi [BV2] to arbitrary dependencies. Unfortunately, the chase may not terminate for embedded dependencies. It was shown, however, that the chase is a *proof procedure* for implication. That is, given Σ and τ, the chase will give a positive answer if $\Sigma \models \tau$, but will not terminate if $\Sigma \not\models \tau$. Furthermore, Beeri and Vardi [BV4] also presented a sound and complete axiom system for typed dependencies. Nevertheless, all these did not seem to lead to a decision procedure for implication. In 1980 researchers started suspecting that the (finite) implication problem for embedded dependencies was unsolvable, and the first result in this direction were announced in June 1980 by two independent teams.

Theorem 3.3.2. [BV6,CLM1] The implication and the finite implication problem for tuple-generating dependencies are unsolvable. □

This result is disappointing especially with regard to finite implication, which is the more interesting notion. As we recall, $\not\models_f$ is recursively enumerable. Thus, if $\models_f$ is not recursive, then it is not even recursively enumerable. That means that there is no sound and complete axiom system for finite implication.

Both proofs of Theorem 3.3.2 seem to use *untypedness* in a very strong way, and do not carry over to the typed case. Shortly later, however, both

teams succeeded in ingeniously encoding untyped dependencies by typed dependencies.

Theorem 3.3.3. [BV7,CLM2] The implication and the finite implication problem for typed tuple-generating dependencies are unsolvable. □

As dependencies, EMVDs have four important properties (see for example (2.3)):

(1) they are tuple-generating,
(2) they are typed,
(3) they have a single atomic formula on the right-hand side of the implication, and
(4) they have two atomic formulas on the left-hand side of the implication.

Dependencies that satisfy properties (1), (2), and (3) above are called *template dependencies,* or TDs [SU]. Thus, EMVDs and EJDs are in particular TDs. Since Theorem 3.3.2 covers properties (1) and (2), the next step was to extend unsolvability to TDs.

Theorem 3.3.4. [GL,Va2] The implication and finite implication problems for TDs are unsolvable. □

In fact, Vardi [Va2] proved a stronger result: the unsolvability for the class of *projected join dependencies.* A projected join dependency (PJD) is of the form $\bowtie[X_1,...,X_k]_X$, where $X \subseteq X_1 \cup ... \cup X_k \subseteq U$. It is obeyed by a relation I if $I[X] = \bowtie\{I[X_1],...,I[X_k]\}[X]$. For an application of PJDs see [MUV]. PJDs extend slightly JDs, since if $X = X_1 \cup ... \cup X_k$, then the PJD $\bowtie[X_1,...,X_k]_X$ is equivalent to the JD $\bowtie[X_1,...,X_k]$. Thus the class of PJDs lies strictly between the classes of EJDs and TDs. The implication and finite implication problems for EJDs are, however, still wide open.

Even though the existence of an axiom system for a certain class of dependencies does not guarantee solvability of the implication problem, finding such a system seems to be a valuable goal. In particular attention was given to *k-ary* systems. In a k-ary axiom systems, all inference rules are of the form $\tau_1,...,\tau_n \vdash \tau$, where $n \leq k$. It is easy to verify, for example, that the systems $\mathcal{FD}$ and $\mathcal{MVD}$ in Section 3.3 are 2-ary.

Theorem 3.3.5. [PP,SW] For all $k>0$, there is no sound and complete k-ary axiom system for implication and finite implication of EMVDs. □

We refer the reader to [BV4,Va2] for a discussion regarding the existence of a non-k-ary axiom system for EMVDs.

Let us refer now to what some people believe are the only "practical" dependencies, FDs and INDs.

Recall that FDs are full dependencies, so implication and finite implication coincide and both are solvable (and by Theorem 3.2.1, quite efficiently). INDs, on the other hand, are embedded dependencies, so a straightforward application of the chase does not yield a decision procedure. A more careful analysis, however, shows that the chase can be forced to terminate.

Theorem 3.3.6. [CFP] The implication and finite implication problem for INDs are equivalent and are PSPACE-complete. □

(PSPACE-complete problems are problems that can be solved using only polynomial space and are hard as any problem that can be solved using polynomial space. It is believed that this problems can not be solved in polynomial time [GJ].)

Let us consider now implication of arbitrary dependencies by INDs. Since containment of tableaux [ASU] can be expressed by dependencies [YP], a test for implication of dependencies by INDs is also a test for containment of conjunctive queries under INDs. We do not know whether implication and finite implication coincide in this case. We have, however, a positive result for implication.

Theorem 3.3.7. [JK] Testing implication of dependencies by INDs is PSPACE-complete. □

The finite implication problem for this case is still open.

Casanova et al. [CFP] investigated the interaction of FDs and INDs, and they discovered that things get more complicated when both kinds of dependencies are put together. First, they showed that implication and finite implication are different (Theorem 3.3.1). In addition they showed

that there is no sound and complete k-ary axiom system for implication and finite implication of FDs and INDs. (Mitchell [Mi1], however, has shown that in a more general sense there is a k-ary axiom system for implication of FDs and INDs.) In view of their results, it did not come as a surprise when Chandra and Vardi and, independently, Mitchell proved unsolvability.

Theorem 3.3.8. [CV,Mi2] The implication and the finite implication problems for FDs and INDs are unsolvable. □

Some people claim is that in practice we encounter only INDs that have a single attribute on each side of the containment, e.g., MANAGER⊆EMPLOYEE. Such INDs are called *unary* INDs (UINDs). Reviewing the proof of Theorem 3.3.1, we realize that even for FDs and UINDs implication and finite implication differ. Considering our experience with dependencies, this looks like a sure sign that the problems are unsolvable. The next result by Kannelakis et al. comes therefore as a refreshing surprise.

Theorem 3.3.9. [KCV] The implication and the finite implication problem for FDs and UINDs are both solvable in polynomial time. □

For other positive results for INDs see [KCV,JK,LMG].

In conclusion to this topic, we would like to mention an argument against the relevance of all the above unsolvability results. The assumption underlying these results is that the input is an arbitrary set Σ of dependencies and a dependency τ. The argument is that the given set Σ is supposed to describe some "real life" application, and in practice it is not going to be arbitrary. Thus, even if we concede that TDs arise in practice, still not every set of TDs arises in practice. The emphasis of this argument is on "real world sets of dependencies", rather than on "real world dependencies". For further study of this argument see [Sc1,Sc2]. While we agree with the essence of this argument, we believe that the results described above are useful in delineating the boundaries between the computationally feasible and infeasible. This is especially important, since we do not yet have robust definitions of real world sets of dependencies.

4. The Universal Relation Model

4.1. Motivation

A primary justification given by Codd for the introduction of the relational model was his view that earlier models were not adequate to the task of boosting the productivity of programmers [Co1,Co3]. One of his stated motivations was to free the application programmer and the end user from the need to specify access paths (the so-called "navigation problem"). A second motivation was to eliminate the need for program modification to accommodate changes in the database structure, i.e., to eliminate access path dependence in programs.

After a few years of experience with relational database management systems, it was realized [CK] that, though being a significant step forward, the relational model by itself fails to achieve complete freedom from user-supplied navigation and from access path dependence. The relational model was successful in removing the need for *physical navigation*; no access paths need to be specified within the storage structure of a single relation. Nevertheless, the relational model has not yet provided independence from *logical navigation,* since access paths among several relations must still be satisfied.

For example, consider a database that has relations *ED*(Employee, Department) and *DM*(Department, Manager). If we are interested in the relationship between employees and managers through departments, then we have to tell the system to take the join of the *ED* and *DM* relations and to project it on the attributes *EM*. This is of course an access path specification, and if the database were to be reorganized to have a single relation *EDM*, then any programs using this access path would have to be modified accordingly.

The *universal relation model* aims at achieving complete access path independence by letting us ask the system in an appropriate language "tell us about employees and their managers", expecting the system to figure out the intended access path for itself. Of course, we cannot expect the system to always select the intended relationship between employees and managers automatically, because the user might have something other than the simplest relationship, the one through departments, in mind, e.g., the manager of the manager of the employee. We shall, in a universal relation system, have to settle for eliminating the need for logical navigation along

certain paths, those selected by the designer, while allowing the user to navigate explicitly in more convoluted ways.

Unlike the relational model, the universal relation model was not introduced as a single clearly defined model, but rather evolved during the 1970's through the work of several researchers. As a result, there have been a significant confusion with regard to the assumptions underlying the model, the so-called "universal relation assumptions". We refer the reader to [MUV], where an attempt is made to clarify the situation.

In this and the next section we restrict ourselves to finite databases.

4.2. Decomposition

The simplest way to implement the universal relation model is to have the database consist a *universal relation,* i.e., a single relation over the set U of all attributes. There are two problems with this approach. First, it assumes that for each tuple in the database we always can supply values for all the attributes, e.g., it assumes that we have full biographic information on all employees. Secondly, storing all the information in one universal relation causes problems when this information needs to be updated. These problems, called *update anomalies,* were identified by Codd [Co2]. The solution to these problems is to have a conceptual database that consists of the universal relation, while the actual database consists of relations over smaller sets of attributes. That is, the database scheme consists of a collection $\mathbf{R} = \{R_1,...,R_k\}$ of attributes sets whose union is U, and the database consists of relations $I_1,...,I_k$, over $R_1,...,R_k$, respectively.

A principal activity in relational database design is the decomposition of the universal relation scheme into a database scheme that has certain nice properties, traditionally called *normal forms.* (We shall not go here into *normalization theory,* which is the study of these normal forms, and the interested reader is referred to [Ma,Ul].) More precisely, starting with the universal scheme U and a set of dependencies Σ, we wish to replace the universal scheme by a database scheme $\mathbf{R} = \{R_1,...,R_k\}$. The idea is to replace the universal relation by its projection on $R_1,...,R_k$. That is, instead of storing a relation I over U, we *decompose* it into $I_1 = I[R_1]$, ..., $I_k = I[R_k]$, and store the result of this decomposition. The map $\Delta_{\mathbf{R}}$ defined by $\Delta_{\mathbf{R}}(I) = \{I[R_1],...,I[R_k]\}$ is called the *decomposition map.*

Clearly, a decomposition cannot be useful unless no loss of information is incurred by decomposing the universal relation. (This is called in [BBG] the *representation principle.*) That is, we must be able to reconstruct I from $I_1,...,I_k$. More precisely, the decomposition map has to be *injective.* For our purposes it suffices that the decomposition map is injective for relations that satisfy the given set Σ of dependencies. In this case we say that it is injective with respect to Σ. When the decomposition map is injective it has a left inverse, called the *reconstruction map.* The basic problems of *decomposition theory* are to formulate necessary and sufficient conditions for injectiveness and to find out about the reconstruction map.

The natural candidate for the reconstruction map is the join, i.e., $I=I_1 \bowtie \ldots \bowtie I_k$. The naturalness of the join led many researchers to the belief that if the reconstruction map exists then it is necessarily the join. This belief was refuted by Vardi [Va3], who constructed an example where the decomposition map is injective, but the reconstruction map is not the join. It is also shown in [Va3] how to express injectiveness as a statement about implication of dependencies. Unfortunately, even when Σ consists of full dependencies, that statement involves also inclusion dependencies. It is not known whether there is an effective test for injectiveness.

If we insist that the join be the reconstruction map, then we can get a stronger result.

Theorem 4.2.1. [BR,MMSU] Let Σ be a set of dependencies, and let R be a database scheme. Δ_R is injective with respect to Σ with the join as the reconstruction map if and only if $\Sigma \models \bowtie[R]$. □

Thus, if Σ consists of full dependencies then we can effectively test whether the decomposition map is injective with respect to Σ.

Another desirable property of decompositions is *independence* [Ri1]. Intuitively, independence means that the relations of the database can be updated independently from each other. For further investigation of the relationship between injectiveness and independence see [BH, BR, MMSU, Va3 .

A point that should be brought up is that decomposition may have some disadvantages. Essentially, decomposition may make it easier to

update the database, but it clearly makes it harder to query it. Since the join operation can be quite expensive computationally, reconstructing the universal relation may not be easy even when the reconstruction map is the join. In fact, even testing whether the relations of the database can be joined without losing tuples is NP-complete, and hence, probably computationally intractable. Let the database consists of relations $I_1,...,I_k$ over attribute sets $R_1,...,R_k$. We say that the database is *join consistent* if there is a universal relation I such that $I_j = I[R_j]$, for $1 \leq j \leq k$. (Rissanen [Ri1] calls a join consistent set of relations *joinable*. A join consistent database is also called *globally consistent* [BFMY], *join compatible* [BR], *valid* [Ri3], *consistent* [Fa6], or *decomposed* [Va3].) It is easy to verify that the database is join consistent if $I_j = \bowtie \{I_1,...,I_k\}[R_j]$, for $1 \leq j \leq k$.

Theorem 4.2.2. [HLY] Testing whether a database is join consistent is NP-complete. □

Thus there is a trade-off between the ease of updating the database and the ease of querying it. The smaller the relation schemes, the easier it is to update the database and the harder it is to query it. Recognizing this trade-off, Schkolnick and Sorenson investigated what they called *denormalization* [SS]. The idea is to decompose the universal scheme with both the ease of updating and the ease of querying in mind. The result of the decomposition depends in this approach on the predicted use of the database.

4.3. The Universal Relation Interface

Suppose now that decomposition has been achieved. That is, assume that, starting with the universal scheme U and a set Σ of dependencies, we have designed a database scheme $\mathbf{R} = \{R_1,...,R_k\}$, and we now have a database $\mathbf{I} = \{I_1,..,I_k\}$ over R. Two questions have now to be resolved: how to determine whether the database is semantically meaningful, i.e., satisfies the given dependencies, and how to respond to the users' queries that refer to the universal relation. If the database is join consistent, then we can construct the universal relation I such that $\Delta_{\mathbf{R}}(I) = \mathbf{I}$. But if the database is not join consistent, then there is no corresponding universal relation.

We outline here one approach to the problem, called the *weak* universal relation approach. (This approach was suggested by Honeyman [Ho]

and further developed in [GM,GMV,MUV]. For other approaches and their relationship to the weak universal relation approach see [GM,GMV,MRW,MUV].) According to this approach, a universal relation exists at least in principle, even though it may not be known. The database is seen, from this viewpoint, as a partial specification of the universal relation. More precisely, the relations $I_1,...,I_k$ are partial descriptions of the projection of the universal relation I on the relation schemes $R_1,...,R_k$. Thus a universal relation I is considered to be a *weak universal relation* for I with respect to Σ if it satisfies Σ and $I_j \subseteq I[R_j]$, for $1 \leq i \leq k$. I is *consistent* with Σ i.e., semantically meaningful, if it has a weak universal relation with respect to Σ.

The above definition is existential in nature and does not lend itself to an effective test. The *consistency problem* is to decide, for a given set Σ of dependencies and a database I over a database scheme R, whether I is consistent with Σ.

Theorem 4.3.1.
1) [GMV] The consistency problem for embedded dependencies is unsolvable.
2) [GMV] The consistency problem for full dependencies is EXPTIME-complete.
3) [Ho] The consistency problem for FDs is solvable in polynomial time. □

Thus, for embedded dependencies there is no effective test for consistency, for full dependencies there is an effective though intractable test, and the good news is that for FDs there is a polynomial time test for consistency. We note that the presence of the independence property, mentioned is Section 4.2, may make it easier to test for consistency. We refer the reader to [CM,Gr2,GY,Sa] for the study of independence in the context of the weak universal relation approach.

We now refer to the issue of query answering. For simplicity we restrict ourselves to queries of the form "give me the relationship between employees and managers". More precisely, the query is a set X of attributes, and the desired answer is the so-called *basic relationship on X*. If we had a unique universal relation I, then answer would undoubtedly be $I[X]$. But in our case we have only weak universal relations, and we clearly have

infinitely many of those. Since we cannot know which of the possible universal relations actually represent the "real world" at a given moment, we assume that the only facts that can be deduced about the universal relation from the database are those that hold is all weak universal relations. This motivated researchers ([MUV,Ya] following [Sa]) to adopt the following definition. Let $weak(I,\Sigma)$ be the set of all weak universal relations of I with respect to Σ. We can see this set as the embodiment of the information represented by the database [Me]. The answer to the query X, denoted $I[X]$, is therefore taken to be $\cap \{J[X] : J \in weak(I,\Sigma)\}$. Note that the answer is with respect to Σ.

The above definition does not seem to lead to an effective procedure for computing $I[X]$.

Theorem 4.3.2.
1) [GMV] Computing answers with respect to embedded dependencies is unsolvable.
2) [GMV] Computing answers with respect to full dependencies is EXPTIME-complete.
3) [Ho] Computing answers with respect to FDs can be done in polynomial time. □

We refer the reader to [Gr3,MRW,MUV,Sag2,Sag3,Ya] for further study of query answering.

We conclude this section by considering again the questions raised in the previous section. There we started with a universal relation I and applied the decomposition map Δ_R, to get the database $\Delta_R(I) = \{I[R_1],...,I[R_k]\}$. Suppose now that we pose the query U to this database. In this case we would expect our query answering mechanism to be the desired reconstruction map, i.e., we would expect $I = \Delta_R(I)[U]$.

Theorem 4.3.3. [MUV] The following two conditions are equivalent:
1) $\Sigma \models \bowtie [R]$.
2) $I = \Delta_R(I)[U]$, for every universal relation I that satisfies Σ. □

In other words, if our query answering mechanism happens to be the reconstruction map, then for join consistent databases it is actually the join.

5. Acyclic database schemes

In the last few years, acyclic database schemes have been introduced and studied [BFMMUY]. The idea is to view database schemes as hypergraphs. A *hypergraph* is a pair $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a finite set of *nodes*, and $\mathcal{E}$ is a set of *edges* (or *hyperedges*), which are arbitrary nonempty subsets of $\mathcal{N}$. An ordinary undirected graph (without self-loops) is, of course, a hypergraph where every edge has exactly two nodes. We shall identify a hypergraph by only mentioning its edges, and tacitly assume that the nodes are precisely those that appear in some edge. We can then view a database scheme as a hypergraph where the relation schemes are the edges. The hypergraph of Figure 5.2 corresponds to the database scheme of Figure 5.1. The correspondence should be clear: for example, there is a {SUPPLIER,PART,COST} edge in the hypergraph of Figure 5.2 because of the {SUPPLIER,PART,COST} relation scheme in Figure 5.1, and so on.

Unlike the situation for ordinary undirected graphs, there are a number of inequivalent, natural definitions of acyclicity for hypergraphs. The type of acyclicity (due to [BFMMUY]) which we shall focus on in this section can be defined by a generalization of one of the usual definitions of acyclicity for ordinary undirected graphs. In particular, an ordinary undirected graph is acyclic in the usual sense if and only if, when considered as a hypergraph, it is acyclic under our definition.

A number of basic, desirable properties of relational database schemes turn out to be equivalent to acyclicity. These properties were defined and studied by a number of researchers, in quite different contexts. It is somewhat remarkable that all of these properties are equivalent. We describe here only a few of these properties. For a more complete survey, see [BFMY, Fa5, Fa6]. Furthermore, there are arguments that the class of acyclic database schemes are natural from a semantic point of view [Li, Sc1].

There is a simple, efficient algorithm for determining acyclicity. For this paper, we shall simply take this algorithm as defining acyclicity. The algorithm is called *Graham's algorithm*, in honor of Marc Graham, who showed [Gr1] that if a hypergraph was accepted by his algorithm, then this was sufficient to imply a certain nice database property. Graham's algorithm was also defined, independently, by Yu and Ozsoyoglu [YO].

The algorithm proceeds by applying the following operations repeatedly, in any order, until none can be applied:

(a) if a node is isolated (that is, if it belongs to precisely one edge), then delete that node;
(b) if an edge is a subset of another edge, then delete the first edge.

The algorithm clearly terminates. If the end result is the empty set, then the original hypergraph is acyclic; otherwise, it is cyclic.

As an example, let us apply this algorithm to the hypergraph of Figure 5.3. Somewhat surprisingly, it turns out that this hypergraph is acyclic, even though it seems to contain a "cycle"; we shall come back to this point later.

We begin the algorithm by writing the edges, one underneath the other:

A	B	C			
		C	D	E	
A				E	F
A		C		E	

(For convenience, we have put common vertices in the same column.)

We begin by deleting the isolated nodes B, D, and F. We are left with:

A	C	
	C	E
A		E
A	C	E

Since the first (AC) row is contained in the last (ACE) row, we now delete the first row:

	C	E
A		E
A	C	E

We now delete the new first and second rows, since each is contained in the new third row. We are left with a single row:

$$A \qquad C \qquad E$$

We now delete the isolated nodes A, C, and E. We are then left with the empty set. Since the algorithm terminates with the empty set, the hypergraph of Figure 5.3 is acyclic.

It is instructive to see an example in which the hypergraph is cyclic. This time, we apply the algorithm to the hypergraph of Figure 5.4. This hypergraph contains three of the four edges of the hypergraph of Figure 5.3. The algorithm begins with

$$\begin{array}{cccccc} A & B & C & & & \\ & & C & D & E & \\ A & & & & E & F \end{array}$$

After deleting the isolated nodes B, D, and F, we are left with:

$$\begin{array}{ccc} A & C & \\ & C & E \\ A & & E \end{array}$$

The algorithm now halts, since no node is isolated and no row is a subset of another row. Since what is left is not the empty set, the hypergraph is cyclic.

Note that the acyclic hypergraph of Figure 5.3 has a cyclic subhypergraph, namely, the hypergraph of Figure 5.4. (A *subhypergraph* of a hypergraph is simply the hypergraph consisting of a subset of the edges.) This counterintuitive phenomenon does not happen with ordinary graphs: that is, it is not possible for a subgraph of an ordinary acyclic graph to be cyclic. Later, we shall mention another type of acyclicity for hypergraphs, where this counterintuitive phenomenon does not occur.

A simple analysis of a natural implementation of Graham's algorithm shows that it can made to run in cubic time. We remark that Tarjan and Yannakakis [TY] have recently obtained a a linear time algorithm for determining acyclicity.

We now discuss a particular desirable property of database schemes that is equivalent to acyclicity. Recall that a database is *join consistent* if there is a single universal relation such that each relation in the database is the appropriate projection of U.

We say that a pair of relations is join consistent if the database consisting only of these two relations is join consistent. Let us say that the two relations are I_1, with attributes R_1, and I_2, with attributes R_2. Let $X=R_1 \cap R_2$. Thus, X is the set of attributes that I_1 and I_2 have in common. It is easy to see that I_1 and I_2 are join consistent precisely if $I_1[X] = I_2[X]$, that is if they agree on their common part. Let us say that a database is *pairwise consistent* if each pair of relations is join consistent. It is clear every join consistent database is pairwise consistent. It would be very nice if the converse were true, since then there would be a simple test for join consistency, namely, pairwise consistency. Unfortunately, however, the converse does not hold. For, it is easy to verify that the database of Figure 5.5 is pairwise consistent; however, it is not hard to see that it is not join consistent. In fact, we already knew that there could be no simple test for join consistency, since, as we noted earlier, determining join consistency is an NP-complete problem [HLY].

However, in the acyclic case our desired converse holds, that is, pairwise consistency and join consistency are equivalent.

Theorem 5.1. [BFMY] If the scheme is acyclic, then a database is join consistent if and only if it is pairwise consistent. □

Can there be any *cyclic* schemes for which join consistency and pairwise consistency are equivalent? The answer is no.

Theorem 5.2. [BFMY] If the scheme is cyclic, then there is a database that is pairwise consistent but not join consistent. □

Putting Theorems 5.1 and 5.2 together, we see that a scheme is acyclic if and only if every pairwise consistent database is join consistent. Therefore (using also the fact that join consistency implies pairwise consistency), it follows that a scheme is acyclic if and only if checking pairwise consistency is an algorithm for testing join consistency. This is an example of a desirable database property that is equivalent to acyclicity.

There is one viewpoint on what we have just discussed that should be emphasized. In the *unrestricted* case (where we do not assume acyclicity), testing join consistency is an NP-complete problem. However, in the *acyclic* case, there is a polynomial-time algorithm for testing join consistency (namely, testing pairwise consistency). This gives us an example of an NP-hard problem that has a polynomial-time algorithm if the scheme is acyclic. We shall mention another such example soon.

We now consider another condition which is equivalent to acyclicity of a hypergraph. The join dependency $\bowtie [X_1, ..., X_k]$ is said to be acyclic precisely if the hypergraph with edges $X_1, ..., X_n$ is acyclic.

Theorem 5.3. [FMU] A join dependency is acyclic if and only if it is equivalent to a set of multivalued dependencies. □

It is known [BFMY] that an acyclic join dependency $\bowtie [R_1, ..., R_n]$ is in fact equivalent to a set of at most n-1 multivalued dependencies (where n is the number of R_i's). Further, the constructions in [BFMY] show that there is a polynomial-time algorithm for finding such a set of n-1 multivalued dependencies. As an example, the acyclic join dependency $\bowtie [ABC, CDE, EFA, ACE]$, which corresponds to the acyclic hypergraph of Figure 5.3, is equivalent to the set $\{AC \twoheadrightarrow DEF, CE \twoheadrightarrow ABF, AE \twoheadrightarrow BCD\}$ of multivalued dependencies.

Beeri et al. [BFMY] and Goodman and Tay [GT] consider the converse question, of when a given set of MVDs is equivalent to some JD. In particular, Goodman and Tay give a polynomial-time algorithm for answering this question.

We can now give another example of an NP-hard problem which acyclicity rends tractable. The problem of deciding whether a set of typed full dependencies implies a JD is NP-hard; in fact, as noted in Theorem 3.2.9 above, this is even true if all of the typed full dependencies are MVDs. However, if the join dependency is acyclic, then there is a polynomial-time algorithm.

Theorem 5.4. Testing whether a set of typed full dependencies implies an acyclic JD can be done in polynomial time.

Proof: As noted above, there is a polynomial-time procedure for finding a set of n-1 MVDs which are equivalent to the given JD $\bowtie[R_1, ..., R_n]$. By Theorem 3.2.8, there is a polynomial-time algorithm for deciding whether the set Σ of typed full dependencies implies each of the multivalued dependencies. Clearly, Σ implies the acyclic JD if and only if it implies all of these MVDs. □

We note that Yannakakis [Ya] has found other problems involving JDs that are NP-hard in general, but which have polynomial-time algorithms if the JDs are acyclic.

Our final example for the desirability of acyclicity involves the property of injectiveness discussed in Section 4.2. We noted there that injectiveness of a decomposition map does not guarantee that the reconstruction map is the join. The situation is different if the decomposition is acyclic.

Theorem 5.5. [BV8] Let Σ be a set of full dependencies, and let R be an acyclic database scheme. If Δ_R is injective with respect to Σ, then the reconstruction map is the join. □

Note that combining Theorems 4.2.1, 5.4, and 5.5, we get a polynomial time test for the injectiveness of an acyclic decomposition with respect to typed full dependencies.

The idea of acyclicity has turned out to be a powerful unifying concept. We refer the reader to [BC, BG, GS] for applications to query processing, which we do not discuss here. We note that Sacca [Sa] and Laver et al. [LMG] deal with the effect of FDs on acyclicity. Further, Biskup and Brüggemann [BiBr] study the effect of FDs on the design process of acyclic database schemes.

Fagin [Fa6] has introduced even more restrictive types of acyclicity, which correspond to database schemes that enjoy even nicer properties. We now focus on one such type of acyclicity, called γ-*acyclicity*. For convenience, let us refer (as Fagin does in [Fa6]) to the type of acyclicity we have been discussing as α-*acyclicity*. As we noted earlier, the definition of α-acyclicity has the counterintuitive property that a subhypergraph of an α-acyclic hypergraph may be α-cyclic. As an example, the α-acyclic

hypergraph of Figure 5.3 has the α-cyclic subhypergraph of Figure 5.4. It turns out that for γ-acyclicity, this counterintuitive phenomenon does not occur; thus, every subhypergraph of a γ-acyclic hypergraph is γ-acyclic.

There are various graph-theoretic definitions of γ-acyclicity [Fa6]. For the purposes of this paper, we simply give an algorithm for determining γ-acyclicity. This algorithm is due to D'Atri and Moscarini [DM], and was proven correct by Fagin [Fa6]. It is very similar in spirit to Graham's algorithm for α-acyclicity, which we presented earlier. We then apply the following operations repeatedly, in any order, until none can be applied:

(a) if a node is isolated (that is, if it belongs to precisely one edge), then delete that node;

(b) if an edge is a singleton (that is, if it contains exactly one node), then delete that edge (but do not delete the node from other edges that might contain it);

(c) if an edge is empty, then delete it;

(d) if two edges contain precisely the same nodes, then delete one of these edges;

(e) if two nodes are edge-equivalent, then delete one of them from every edge that contains it. (We say that two nodes are *edge-equivalent* if they are in precisely the same edges.)

The algorithm clearly terminates. If the end result is the empty set of edges, then the original hypergraph is γ-acyclic; otherwise, it is γ-cyclic.

As an example, let us apply this algorithm to the hypergraph of Figure 5.6. As in the previous example, we begin by by writing the edges, one underneath the other. The edges are:

$$\begin{array}{cccccc} & B & C & D & E & F \\ A & B & C & D & & \\ & & C & & & \\ & & C & D & & \\ & & & & E & F \end{array}$$

Node A is isolated, and edge $\{C\}$ is a singleton, so both are deleted, by rules (a) and (b). This leaves us with

$$\begin{array}{ccccc} B & C & D & E & F \\ B & C & D & & \\ & C & D & & \\ & & & E & F \end{array}$$

Nodes E and F are edge-equivalent, and so, by rule (e), we delete F from both edges that contain it. Similarly, nodes C and D are edge-equivalent, and so we delete D from all three edges that contain it. We are left with

$$\begin{array}{ccc} B & C & E \\ B & C & \\ & C & \\ & & E \end{array}$$

The third and fourth edges above are singletons, and so they are eliminated. This leaves

$$\begin{array}{ccc} B & C & E \\ B & C & \end{array}$$

Node E is isolated; after it is deleted, we are left with

$$\begin{array}{cc} B & C \\ B & C \end{array}$$

These edges are identical, so we delete one by rule (d). We are left with

$$B \quad C$$

Both nodes are now isolated, and so they are deleted. We are left with a single empty edge, which is deleted by rule (c). The end result is the empty set of edges, and so the original hypergraph is γ-acyclic.

Fagin [Fa6] presents a number of desirable database properties, each which is equivalent to the scheme being γ-acyclic. We mention only one property, which involves dependencies. We first need to define the concept of connectedness for hypergraphs; it is the obvious generalization of the definition of connectedness for graphs. A *path* from node s to node t is a sequence of $k \geq 1$ edges $E_1,\ldots,E_k$ such that

(i) s is in E_1,
(ii) t is in E_k, and
(iii) $E_i \cap E_{i+1}$ is nonempty if $1 \leq i < k$.

A hypergraph is *connected* if for each pair of nodes, there is a path from one to the other.

Theorem 5.5. [Fa6] Let R be a database scheme. Then the following are equivalent:

(1) R is γ-acyclic.
(2) For every connected subset $S \subseteq R$, we have $\bowtie[R] \models \bowtie[S]$. □

Note that $\bowtie[S]$ can be an embedded JD. Thus statement (2) of Theorem 5.5 above says that for every connected subset S of R, and for every join consistent database I over R, if $J \subseteq I$ is the subdatabase over S, then $\bowtie J$ is a projection of $\bowtie I$.

As an example, consider the database scheme of Figure 5.1, which is γ-cyclic. The join of the {SUPPLIER,PROJECT,DATE} relation with the {PROJECT,PART,COUNT} relation might introduce a SUPPLIER, PART, PROJECT triple that does not appear in the SUPPLIER, PART, PROJECT relation (the "connection trap" [Co1].)

We note also that the various notions of acyclicity turn out to be very useful for the design of universal relation interfaces. We refer the reader to [Fa6, MU, Ya].

BIBLIOGRAPHY

[ABU] A. V. Aho, C. Beeri, and J. D. Ullman, The theory of joins in relational data bases. ACM Trans. on Database Systems 4(1979), 297-314.

[ASU] A. V. Aho, Y. Sagiv, and J. D. Ullman, Equivalences among relational expressions. SIAM J. Computing 8(1979), 218-246.

[Ar] W. W. Armstrong, Dependency structures of database relationships. Proc. IFIP 74, North Holland, 1974, 580-583.

[Bee] C. Beeri, On the membership problem for functional and multivalued dependencies in relational databases. ACM Trans. on Database Systems 5(1980), 241-259.

[BB] C. Beeri and P. A. Bernstein, Computational problems related to the design of normal form relational schemas. ACM Trans. on Database Systems 4(1979), 30-59.

[BBG] C. Beeri, P. A. Bernstein, and N. Goodman, A sophisticate's introduction to database normalization theory. Proc. Int. Conf. on Very Large Data Bases, 1978, Berlin, 113-124.

[BFH] C. Beeri, R. Fagin, and J.H. Howard, A complete axiomatization for functional and multivalued dependencies in database relations. Proc. ACM SIGMOD Conf. on Management of Data, 1977, Toronto, 47-61.

[BFMMUY] C. Beeri, R. Fagin, D. Maier, A. O. Mendelzon, J. D. Ullman, and M. Yannakakis, Properties of acyclic database schemes. Proc. 13th ACM SIGACT Symp. on Theory of Computing, 1981, Milwaukee, 355-362.

[BFMY] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, On the desirability of acyclic database schemes. J. ACM 30(1983), 479-513.

[BH] C. Beeri and P. Honeyman, Preserving functional dependencies. SIAM J. Computing 10(1981), 647-656.

[BK] C. Beeri and H. F. Korth, Compatible attributes in a universal relation. Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1982, Los Angeles, 51-62.

[BMSU] C. Beeri, A. O. Mendelzon, Y. Sagiv, and J. D. Ullman, Equivalence of relational database schemes, SIAM J. Computing 10(1981), 352-370.

[BR] C. Beeri and J. Rissanen, Faithful representation of relational database schemes. IBM Research Report, San Jose, California, 1980.

[BV1] C. Beeri and M. Y. Vardi, On the properties of join dependencies. In Advances in Database Theory (H. Gallaire, J. Minker, and J. M. Nicolas, Eds.), Plenum Press, 1981, 25-72.

[BV2] C. Beeri and M. Y. Vardi, A proof procedure for data dependencies. J. ACM 31(1984), 718-741.

[BV3] C. Beeri and M. Y. Vardi, On the complexity of testing implications of data dependencies. Hebrew University of Jerusalem Technical Report, Dec. 1980.

[BV4] C. Beeri and M. Y. Vardi, Formal systems for tuple and equality-generating dependencies. SIAM J. Computing 13(1984), 76-98.

[BV5] C. Beeri and M. Y. Vardi, Formal systems for join dependencies. Hebrew Univ. of Jerusalem Technical Report, 1981. To appear in Theoretical Computer Science.

[BV6] C. Beeri and M. Y. Vardi, The implication problem for data dependencies. Proc. XP1 Workshop on Relational Database Theoery, Stony Brook, NY, June 1980.

[BV7] C. Beeri and M. Y. Vardi, The implication problem for data dependencies. Proc. 8th Int. Colloq. on Languages Automata and Programming, 1981, Acre Israel. Appeared in: Lecture Notes in Computer Science - Vol. 115, Springer-Verlag, 1981, 73-85.

[BV8] C. Beeri and M. Y. Vardi, On acyclic database decompositions. Information and Control 6(1984), 75-84.

[Ber] P. A. Bernstein, Synthesizing third normal form relations from functional dependencies. ACM Trans. on Database Systems 1(1976), 277-298.

[BC] P. A. Bernstein and D. W. Chiu, Using semi-joins to solve relational queries, J. ACM 28(1981), 25-40.

[BG] P. A. Bernstein and N. Goodman, The power of natural semijoins, SIAM J. Computing 10(1981), 751-771.

[BiBr] J. Biskup and H. H. Brüggemann, Towards designing acyclic database schemas, Proc. ONERA-CERT Workshop on Logical Bases for Data Bases, Toulouse, 1982.

[CFP] M. A. Casanova, R. Fagin, and C. Papadimitriou, Inclusion dependencies and their interaction with functional dependencies. J. Computer and System Sciences 28(1984), 29-59.

[CM] E. P. F. Chan and A. O. Mendelzon, Independent and separable database schemes. Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Atlanta, 288-296.

[CLM1] A. K. Chandra, H. R. Lewis, and J. A. Makowsky, Embedded implicational dependencies and their inference problem. Proc. XP1 Workshop on Relational Database Theoery, Stony Brook, NY, June 1980.

[CLM2] A. K. Chandra, H. R. Lewis, and J. A. Makowsky, Embedded implicational dependencies and their inference problem. Proc. 13th ACM Symp. on Theory of Computing, 1981, Milwaukee, 342-354.

[CV] A. K. Chandra and M. Y. Vardi, The implication problem for functional and inclusion dependencies is undecidable. IBM Research Report RC 9980, May 1983. To appear in SIAM J. Computing.

[Co1] E. F. Codd, A relational model of data for large shared data banks. Comm. ACM 13(1970), 377-387.

[Co2] E. F. Codd, Further normalizatioı of the data base relational model. Courant Computer Science Symposia 6: Data Base Systems, 1972, Prentice Hall, 33-64.

[Co3] E. F. Codd, Relational databases: a practical foundation for productivity. Comm. ACM 25(1982), 109-117.

[DM] A. D'Atri and M. Moscarini, Acyclic hypergraphs: their recognition and top-down vs bottom-up generation. Consiglio Nazionale Delle Ricerche, Istituto di Analisi dei Sistemi ed Informatica, R.29, 1982.

[DP] P. De Bra and J. Paredaens, Conditional dependencies for horizontal decompositions. Proc. 10th Int. Colloq. on Languages Automata and Programming, 1981, Barcelona. Appeared in: Lecture Notes in Computer Science - Vol. 154, Springer-Verlag, 1983, 67-82.

[DST] P. J. Downey, R. Sethi, and R. E. Tarjan, Variations on the common subexpression problem. J. ACM 27(1980), 758-771.

[DG] B. S. Dreben and W. D. Goldfarb, The Decision Problem: Solvable Classes of Quantificational Formulas. Addison Wesley, 1979.

[Fa1] R. Fagin, Multivalued dependencies and a new normal form for relational databases. ACM Trans. on Database Systems 2(1977), 262-278.

[Fa2] R. Fagin, A normal form for relational databases that is based on domains and keys. ACM Trans. on Database Systems 6(1981), 387-415.

[Fa3] R. Fagin, Armstrong databases. Proc. 7th IBM Symp. on Mathematical Foundations of Computer Science, Kanagawa, Japan, May 1982. Also appeared as IBM Research Report RJ3440 (April 1982), San Jose, California.

[Fa4] R. Fagin, Horn clauses and database dependencies. J. ACM 29(1982), 952-985.

[Fa5] R. Fagin, Acyclic database schemes of various degrees: a painless introduction. Proc. CAAP83 8th Colloquium on Trees in Algebra and Programming., Appeared in: Springer-Verlag Lecture Notes in Computer Science - vol. 159, 1983, ed. G. Ausiello and M. Protasi, 65-89.

[Fa6] R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes. J. ACM 30(1983), 514-550.

[FMU] R. Fagin, A. O. Mendelzon, and J. D. Ullman, A simplified universal relation assumption and its properties. ACM Trans. on Database Systems 7(1982), 343-360.

[TF] P. C. Fischer and D.-M. Tsou, Whether a set of multivalued dependencies implies a join dependency is NP-hard. SIAM J. Computing 12(1983), 259-266.

[Ga] Z. Galil, An almost linear-time algorithm for computing a dependency basis in a relational database. J. ACM 29(1982), 96-102.

[GJ] M. R. Garey and D. S. Johnson, Computers and Intractibility: A Guide to the Theory of NP-Completeness. Freeman, 1979.

[GZ] S. Ginsburg and S. M. Zaiddan, Properties of functional dependency families. J. ACM 29(1982), 678-698.

[GS] N. Goodman and O. Shmueli, Tree queries: a simple class of queries. ACM Trans. on Database Systems 7(1982), 653-677.

[GT] N. Goodman and Y. C. Tay, A characterization of multivalued dependencies equivalent to a join dependency. Infornation Processing Letters 18(1984), 261-266.

[Gr1] M. H. Graham, On the universal relation. Technical Report, Univ. of Toronto, Sept. 1979.

[Gr2] M. H. Graham, Path expressions in databases. Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Atlanta, 366-378.

[Gr3] M. H. Graham, Functions in databases. ACM Trans. on Database Systems 8(1983), 81-109.

[GM] M. H. Graham and A. O. Mendelzon, Notions of dependency satisfaction. Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Los Angeles, 177-188.

[GMV] M. H. Graham, A. O. Mendelzon, and M. Y. Vardi, Notions of dependency satisfaction. Stanford University Technical Report STAN-CS-83-979, Aug. 1983. To appear in J. ACM.

[GY] M. H. Graham and M. Yannakakis, Independent database schemes. J. Computer and System Sciences 28(1984), 121-141.

[GJ] J. Grant and B. E. Jacobs, On the family of generalized dependency constraints. J. ACM 29(1982).

[GL] Y. Gurevich and H. R. Lewis, The inference problem for template dependencies. Proc. First ACM SIGACT-SIGMOD Principles of Database Systems (1982), Los Angeles, 221-229.

[HITK] K. Hagihara, M. Ito, K. Taniguchi, and T. Kasami, Decision problems for multivalued dependencies in relational databases. SIAM J. Computing 8(1979), 247-264.

[He] I. J. Heath, Unacceptable file operations in a relational data base. Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, 1971, San Diego.

[Ho] P. Honeyman, Testing satisfaction of functional dependencies. J. ACM 29(1982), 668-677.

[HLY] P. Honeyman, R. E. Ladner, and M. Yannakakis, Testing the universal instance assumption. Information Processing Letters 10(1980), 14-19.

[Hu] R. Hull, Finitely specifiable implicational dependency families. J. ACM 31(1984), 210-226.

[JK] D. S. Johnson and A. Klug, Testing containment of conjunctive queries under functional and inclusion dependencies. J. Computer and System Sciences 28(1984), 167-189.

[KCV] P. C. Kannelakis, S. S. Cosmadakis, and M. Y. Vardi, Unary inclusion dependencies have polynomial-time inference problems. Proc. 15th ACM SIGACT Symp. on Theory of Computing, 1983, Boston, 264-277.

[LMG] K. Laver, A. O. Mendelzon, and M. H. Graham, Functional dependencies on cyclic database schemes. Proc. ACM SIGMOD Symp. on Management of Data, 1983, San Jose, 79-91.

[Le] H. Lewis, Complexity results for classes of quantificational formulas. J. Computer and Systems Sciences 21(1980), 317-353.

[Li] Y. E. Lien, On the equivalence of database models. J. ACM 29(1982), 333-363.

[LD] L. Liu and A. Demers, An algorithm for testing lossless join property in relational databases. Information Processing Letters 11(1980), 73-76.

[Ma] D. Maier, The Theory of Relational Databases. Computer Science Press, Rockville, Maryland, 1983.

[MMSU] D. Maier, A. O. Mendelzon, F. Sadri, and J. D. Ullman, Adequacy of decompositions of relational databases. J. Computer and Systems Sciences 21(1980), 368-379.

[MMS] D. Maier, A. Mendelzon, and Y. Sagiv, Testing implications of data dependencies. ACM Trans. on Database Systems 4(1979), 455-469.

[MRW] D. Maier, D. Rozenshtein, and D. S. Warren, Windows on the world. Proc. ACM SIGMOD Symp. on Management of Data, 1983, San Jose, 68-78.

[MSY] D. Maier, Y. Sagiv, and M. Yannakakis, On the complexity of testing implications of functional and join dependencies. J. ACM 28(1981), 680-695.

[MU] D. Maier and J. D. Ullman, Connections in acyclic hypergraphs. Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (1982), Los Angeles, 34-39.

[MUV] D. Maier, J. D. Ullman, and M. Y. Vardi, On the foundations of the universal relation model. ACM Trans. on Database Systems 9(1984), 283-308.

[Me] A. Mendelzon, Database states and their tablueax. ACM Trans. on Database Systems 9(1984), 264-282.

[MM] Mendelzon, A. O. and D. Maier, Generalized mutual dependencies and the decomposition of database relations. Proc. Int. Conf. on Very Large Data Bases, (A. L. Furtado and H. L. Morgan, eds.), 1979, 75-82.

[Mi1] J. C. Mitchell, Inference rules for functional and inclusion dependencies. Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Atlanta, 58-69.

[Mi2] J. C. Mitchell, The implication problem for functional and inclusion dependencies. Information and Control 56(1983), 154-173.

[Ni] J-M. Nicolas, First order logic formalization for functional, multivalued, and mutual dependencies. Proc. ACM SIGMOD Symp. on Management of Data, 1978, 40-46.

[Pa] J. Paredaens, Transitive dependencies in a database scheme. MBLE Research Report R387, 1979.

[PJ] J. Paredaens and D. Janssens, Decompositions of relations: a comprehensive approach. In Advances in Data Base Theory - Vol. 1 (H. Gallaire, J. Minker, and J-M. Nicolas, eds.), Plenum Press, 1981, 73-100.

[PP] D. S. Parker and K. Parsaye-Ghomi, Inference involving embedded multivalued dependencies and transitive dependencies. Proc. ACM SIGMOD Symp. on Management of Data, 1980, 52-57.

[Ri1] J. Rissanen, Independent components of relations, ACM Trans. on Database Systems 2(1977), 317-325.

[Ri2] J. Rissanen, Theory of relations for databases - a tutorial survey. Proc. 7th Symp. on Math. Found. of Comp. Science, 1978, Lecture Notes in Computer Science - Vol. 64, Springer-Verlag, 537-551.

[Ri3] J. Rissanen, On equivalence of database schemes. Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1982, Los Angeles, 23-26.

[Ro] H. Rogers, Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.

[Sac] D. Sacca, On the recognition of coverings of acyclic database hypergraphs. Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Atlanta, 297-304.

[SU] F. Sadri and J. D. Ullman, Template dependencies: A large class of dependencies in relational databases and their complete axiomatization. J. ACM 29(1981), 363-372.

[Sag1] Y. Sagiv, An algorithm for inferring multivalued dependencies with an application to propositional logic. J. ACM 27(1980), 250-262.

[Sag2] Y. Sagiv, Can we use the universal instance assumption without using nulls? Proc. ACM SIGMOD Symp. on Management of Data, 1981, 108-120.

[Sag3] Y. Sagiv, A characterization of globally consistent databases and their correct access paths. ACM Trans. on Database Systems 8(1983), 266-286.

[SW] Y. Sagiv and S. Walecka, Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies. J. ACM 29(1982), 103-117.

[SS] M. Schkolnick and P. Sorenson, The effects of denormalization on database performance. The Australian Computer Journal 14(1982), 12-18.

[Sc1] E. Sciore, Real-world MVDs. Proc. ACM SIGMOD Symp. on Management of Data, 1981, 121-132.

[Sc2] E. Sciore, Inclusion dependencies and the universal instance. Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, Atlanta, 48-57.

[Sc3] E. Sciore, A complete axiomatization of full join dependencies. J. ACM 29(1982), 373-393.

[TY] R. E. Tarjan and Yannakakis, M., Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Computing 13(1984), 566-579.

[Ul] J. D. Ullman, Principles of Database Systems. Computer Science Press, Rockville, Maryland (1982)

[Va1] M. Y. Vardi, The implication problem for data dependencies in relational databases. Ph.D. Dissertation (in Hebrew), The Hebrew University in Jerusalem, Sept. 1981.

[Va2] M. Y. Vardi, The implication and finite implication problems for typed template dependencies. J. Computer and System Sciences 28(1984), 3-28.

[Va3] M. Y. Vardi, On decomposition of relational databases. Proc. 23rd IEEE Symp. on Foundation of Computer Science, Chicago, 1982, 176-185.

[Va4] M. Y. Vardi, Inferring multivalued dependencies from functional and join dependencies. Acta Informatica, 19(1983), 305-324.

[Va5] M. Y. Vardi, A note on lossless database decompositions. Information Processing Letters 18(1984), 257-260.

[Ya] M. Yannakakis, Algorithms for acyclic database schemes. Proc. Int. Conf. on Very Large Data Bases, 1981, 82-94.

[YP] M. Yannakakis and C. Papadimitriou, Algebraic dependencies. J. Computer and System Sciences 25(1982), 3-41.

[YO] C. T. Yu and M. Z. Ozsoyoglu, An algorithm for tree-query membership of a distributed query. Proc. IEEE COMPSAC, 1979, 306-312.

[Za] C. Zaniolo, Analysis and design of relational schemata for database systems, Ph.D. Dissertation, Tech. Rep. UCLA-ENG-7669, UCLA, July 1976.

IBM Research Laboratory, San Jose, California 95193

EMP	DEPT	MGR
Hilbert	Math	Gauss
Pythagoras	Math	Gauss
Turing	Computer Science	von Neumann

Figure 2.1

EMP	DEPT	MGR
Hilbert	Math	Gauss
Pythagoras	Math	Gauss
Turing	Computer Science	von Neumann
Cauchy	Math	Euler

Figure 2.2

EMP	DEPT
Hilbert	Math
Pythagoras	Math
Turing	Computer Science

DEPT	MGR
Math	Gauss
Computer Science	von Neumann

Figure 2.3

STORE	ITEM	PRICE
Macy's	Toaster	$20.00
Sears	Toaster	$15.00
Macy's	Pencil	$ 0.10

Figure 2.4

STORE	ITEM
Macy's	Toaster
Sears	Toaster
Macy's	Pencil

ITEM	PRICE
Toaster	$20.00
Toaster	$15.00
Pencil	$ 0.10

Figure 2.5

EMP	SALARY	CHILD
Hilbert	$80K	Hilda
Pythagoras	$30K	Peter
Pythagoras	$30K	Paul
Turing	$70K	Tom

Figure 2.6

EMP	CHILD	SKILL
Hilbert	Hilda	Math
Hilbert	Hilda	Physics
Pythagoras	Peter	Math
Pythagoras	Paul	Math
Pythagoras	Peter	Philosophy
Pythagoras	Paul	Philosophy
Turing	Tom	Computer Science

Figure 2.7

SUPPLIER	PROJECT	DATE

SUPPLIER	PART	COST

SUPPLIER	PART	PROJECT

Figure 5.1

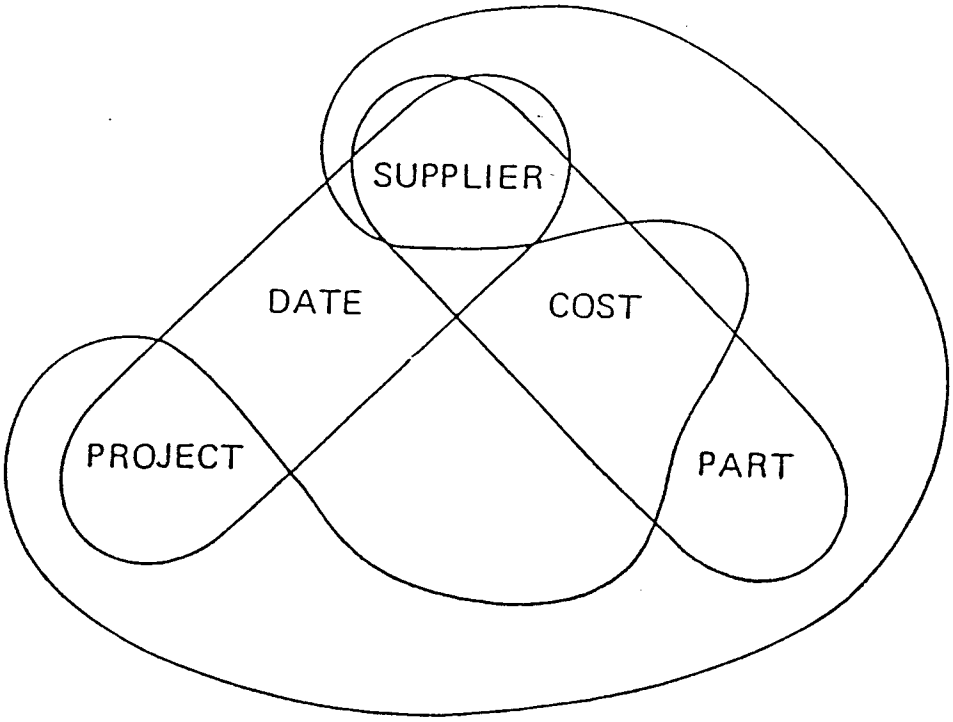

Figure 5.2

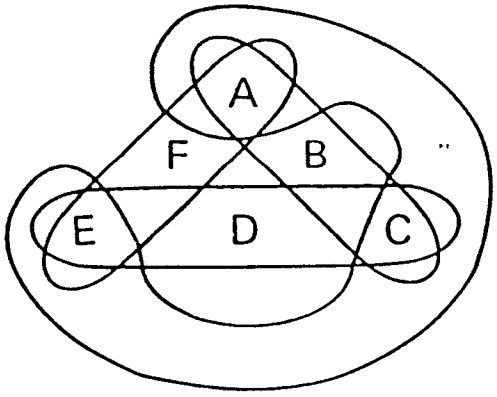

Figure 5.3

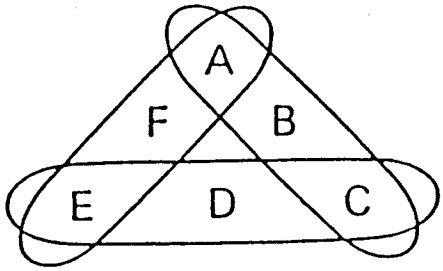

Figure 5.4

A	B
0	0
1	1

B	C
0	0
1	1

A	C
0	1
1	0

Figure 5.5

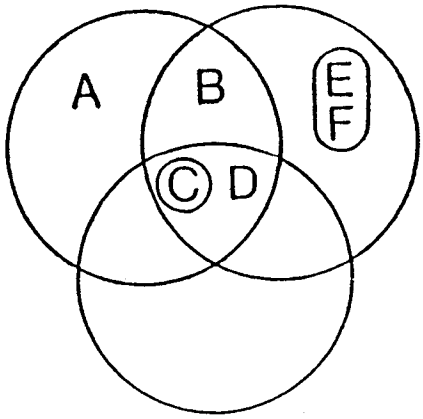

Figure 5.6