Elements of Relational Database Theory

Paris C. Kanellakis

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-39
October 1989

Elements of Relational Database Theory

Paris C. Kanellakis
Department of Computer Science
Brown University, P.O. Box 1910
Providence, RI 02912, USA.

Abstract: The goal of this paper is to provide a systematic and unifying introduction to relational database theory, including some of the recent developments in database logic programming. The first part of the presentation covers the two basic components of the relational data model: its specification component, that is the database scheme with dependencies, and its operational component, that is the relational algebra query language. The choice of basic constructs, for specifying the semantically meaningful databases and for querying them, is justified through an in-depth investigation of their properties. Some important research themes are reviewed in this context: the analysis of the hypergraph syntax of a database scheme and the extensions of the query language using deduction or universal relation assumptions. The subsequent parts of the presentation are structured around the two fundamental concepts illustrated in the first part, namely dependencies and queries. The main themes of dependency theory are implication problems and applications to database scheme design. Queries are classified in a variety of ways, with emphasis on the connections between the expressibility of query languages, finite model theory and logic programming. The theory of queries is very much related to research on database logic programs, which are an elegant formalism for the study of the principles of knowledge base systems. The optimization of such programs involves both techniques developed for the relational data model and new methods for analysing recursive definitions. The exposition closes with a discussion of how relational database theory deals with the problems of complex objects, incomplete information and database updates.

1 Introduction

1.1 Some Motivation and Historical Remarks

The practical need for efficient manipulation of large amounts of structured information and the basic insight that "data should be treated as an integrated resource, independent of application programs" led to the development of database management as an important area of computer science research. Database research has had a major impact on software systems and computer science in general; it has provided one of the few paradigms of man-machine interaction which is both of a very high level, akin to programming in logic, and computationally efficient. *Database theory* grew as the theory corresponding to and directly influencing a number of *database management system* (DBMS) implementations.

Database technology presents the theoretical community with challenging research questions, which can be classified into three broad categories: problems of *relational database theory*, problems of *transaction processing* and problems of *access methods*. The unity of research in these dissimilar areas has been provided by the database management systems themselves, which are large integrated software systems addressing issues in all three areas. Of the three facets of database theory, relational theory is the one more closely identified with the field. It is also the subject of our presentation and it may be viewed as an important application of mathematical logic to computer science problems. The emphasis of the other two facets of database theory is on algorithms and data structures; and these are outside the scope of our exposition.

For a view of the subject in its entirety we refer the reader to [Ull1], an advanced textbook on databases, and to [KorS], an introductory level one. Relational database theory is the topic of [Mai1], a monograph which can be used to fill in many of the proofs omitted here. Transaction processing, e.g., *database concurrency control* and *recovery*, has many connections to the theory of operating systems and to distributed algorithms. We recommend two recent monographs in this area: [Pap] for the theoretical development and [BerHG] for a more applied treatment. The last branch of this threefold classification, the study of access methods, is the area closest to the theory of data structures and is best reviewed in that context. Its distinguishing characteristic is an emphasis on large volumes of structured data residing in secondary storage; see [Ull1].

The pioneering work of E.F. Codd, in the early 1970's, offered both the theoretical and practical communities an elegant, well-motivated and appropriately restricted computational paradigm for database management: *the relational data model*. Much of the systems effort in the 1970's was devoted to efficient implementations of this approach; a largely successful endeavor which has made relational DBMS's a database technology standard. For historical perspectives on the relational data model and on the interplay of theory and practice we recommend [Cod5], [Sel], and [Ull3].

The relational data model is the cornerstone of relational database theory, which was to a large extent developed by analysing and enriching this original kernel. In this framework, a *database* is a set of (finite) relations interpreting a set of nonlogical relation symbols, the *database scheme*. Relations are manipulated using the "procedural" *relational algebra* query language or its equivalent "declarative" *relational calculus*. The duality of algebra and calculus is based on an algebraization of first-order definitions. The addition of *recursion* (via fixpoints or equivalently via deduction) to first-order definitions leads to more expressive query languages, which are related to logic programming. Database query programs have been studied extensively. Their optimization based on "compile-time" transformations and efficient "run-time" evaluation is crucial, if one is to map a high level programming paradigm into a computationally feasible one.

The querying capability of the relational data model (i.e., the definition of functions from databases to databases) is complemented by a capability for specification of "the legal" databases (i.e., the definition of sets of databases). This allows constraining the databases under consideration and, consequently, expressing more knowledge about the objects represented. Codd defined *functional dependencies* as a useful device for this purpose. *Dependencies*, in general, are semantically meaningful and syntactically restricted sentences of the predicate calculus that must be satisfied by any "legal" database. Their presence remedies some of the semantic poverty of relations, e.g., with pure relations one has trouble representing the fact that some relationships are one-to-one or one-to-many. Studies of the decision problem and other computational properties of dependencies have been motivated by questions of good database scheme design. Interestingly, they have also contributed to basic research in mathematical logic.

Theoretical research on the relational data model itself attained a certain degree of maturity in the late 1970's and early 1980's. The efficiency and robustness provided by the relational DBMS's of the 1970's opened the way to more ambitious goals. Current research on logic and databases aims at developing logic programming into a database tool of comparable efficiency. Here the term *knowledge base* is appropriate given the potential new applications. This has brought the logic programming and database communities into closer contact and has stimulated new research directions for theoretical computer science. For a historical perspective on the relationship between the two fields we recommend [GalM], [GalMN], [MaiW], [Min1], and [Min2].

In order to highlight the unity of the subject, we structure our presentation around the two basic concepts of dependencies and queries. This allows us to introduce the current work on logic and databases as the natural evolution of the first investigations into the relational data model.

For more details on dependencies and queries, we recommend the surveys [FagV] and [Cha2] respectively. We stress the contributions of database theory to mathematical logic and, in particular, to finite model theory; see [Gur1, Gur2]. Space limitations do not allow a detailed examination of many interesting questions. Fortunately, we can refer the reader to a number of additional recent surveys: [Var8] for dependency theory, [JarK] for relational algebra optimization, [BanR] for database logic program evaluation, [Kan2] for queries and parallel computation, and [Imm4] for queries and the expressibility of logics.

Despite its elegance, relational theory does not give satisfactory answers to many database problems. This is, in part, the price one has to pay for the simplicity of the relational data model. For example, it is often easier to define hierarchical structures directly, than indirectly through dependencies. In order to represent and manipulate *complex objects* one can extend the data-type relation; this is usually done by introducing *set* and *tuple constructors*. We discuss some of the work on complex objects and recommend [Hul2] for a recent survey of this area, as well as [HulK] for further readings on *semantic data models*. We also discuss two other issues of practical significance: querying *incomplete information* databases and *updating* a database, i.e., coping with uncertainty and with dynamic change. In both cases relational database theory offers interesting solutions.

Relational formalisms for DBMS's have the advantage of being "value-oriented" [Ull3]. That is, a clear separation is achieved between the logical and the physical description of the data and access is specified at the logical level by value. This is also in the spirit of logic programming. At present, there is a growing interest in alternative "object-oriented" approaches, e.g., see [Ban2].

"Object-oriented" approaches emphasize the organization of the data into *inheritance* hierarchies of meaningful *abstract data-types*: *objects that encapsulate both the data and the operations for accessing it.* There is a fair amount of ongoing experimentation in object-oriented DBMS's. Some of this work realizes earlier semantic data model proposals. Some of it may be viewed as design of programming languages with types that *persist* beyond the scope of programs. What is the right formal model for the object-oriented framework is an interesting open question. It's solution will (most probably) combine elements of relational database theory and of the theory of abstract data-types.

1.2 A Roadmap of the Contents

A brief description of the material is now in order.

A detailed overview of the relational data model is contained in Section 2. After some basic definitions, in Section 2.1, we attempt to clarify the two choices made: of query language and dependencies. Section 2.2 is a three part justification of relational algebra based on: (I) its equivalence to the relational calculus, (II) its low computational complexity, and (III) its potential for query optimization via the homomorphism technique. The three part argument for functional dependencies, in Section 2.3, has analogous goals and is based on: (I) the elegant axiomatization of these special sentences, (II) their polynomial-time computational properties, and (III) the preservation of many of these properties when other semantically meaningful statements are added. The attribute notation is a particular feature of relational database theory; the database scheme is a hypergraph on a set of attributes. In Section 2.4, we illustrate some of the advantages of this notation. Hypergraph acyclicity is a simple to test syntactic condition on the database scheme, which implies a host of interesting semantic properties for the database. We take a critical look at the limitations of the relational data model in Section 2.5. We describe two extensions to the basic framework: deductive and universal relation data models. We argue why these extensions are related and comment on the wealth of analysis about them.

Dependency theory is the subject of Section 3. The classification of dependencies in Section 3.1 focuses on some important classes of statements and on the main computational question: testing for dependency implication. This problem is closely related to the decision problem in logic. In Section 3.2, we examine database scheme design: from (I) defining schemes with desirable semantic properties such as independence, to (II) listing normal form schemes that have these properties and are also constructible.

In deductive and universal relation data models, dependencies assume (indirectly) some of the functionality of query languages. In Section 4 we come back to the study of extensions of the relational data model, but through a direct investigation of the expressive power of query language primitives. The classification of queries in Section 4.1 proceeds: (I) along the lines of computational complexity, and (II) based on the expressibility of logics over finite structures, fixpoint logics in particular. An important and robust class of queries are the fixpoint queries, which correspond to the functions expressible in many database logic programming formalisms. Their analysis is at the heart of the topical interest in knowledge bases. In Section 4.2, we identify some of the emerging themes in this area: (I) logic programs without function symbols but which may have negation (we call them recursive rules), (II) optimization of recursive rules via the stage function technique, and (III) basic "top-down" versus "bottom-up" tradeoffs in recursive rule evaluation. We close our discussion of queries, in Section 4.3, with references to the work on complex object data models.

Section 5 is a discussion of how relational database theory deals with incomplete information and updates. These are topics that have attracted a fair amount of attention. Unfortunately, they do not seem to fit in as tidy a framework as queries and dependencies. These two topics, together with the open questions of query and dependency theory, account for much of the ongoing research in relational database theory.

References to the literature are given in the text and there are additional bibliographic comments at the ends of sections. As with any presentation of this type, the choice of material has to be selective and the list of references is by no means comprehensive.

2 The Relational Data Model

2.1 Relational Algebra and Functional Dependencies

We start with some definitions and notation that are basic in database theory. We then introduce the syntax and the semantics of relational algebra and of functional dependencies.

Let $\mathcal{U}$ be a countably infinite set of *attributes*. We denote attributes using capital letters from the beginning of the alphabet, $A, B, C, A_1, \ldots$, and capital letters from the end of the alphabet, $X, Y, Z, X_1, \ldots$, to denote sets of attributes. We usually do not distinguish between attribute A and attribute set $\{A\}$. Notation XY is a convenient shorthand for the union of attribute sets X and Y. Thus, ABC denotes the attribute set $\{A, B, C\}$.

Every attribute A has an associated set of *values* $\Delta[A]$, called A's *domain*. *The domain* is the set of values Δ, the union of the domains of all the attributes. Δ is countably infinite and disjoint from $\mathcal{U}$. We denote values using small letters, $a, b, c, a_1, \ldots$, from the beginning of the alphabet. Throughout our exposition we fix $\mathcal{U}$ and Δ, moreover, we assume that $\Delta = \Delta[A]$ for each A in $\mathcal{U}$. This is for notational simplicity only; under some weak technical conditions, the theory can be developed for any $\Delta[A]$'s (see Remark 2.1.4).

The following definitions highlight the clean distinction made in the relational data model between *relation schemes* and *relations* (the latter are sometimes called *relation instances* or *states* in the literature). This distinction parallels the one in mathematical logic between the syntactic concept of *vocabulary of relation symbols* and the semantic concept of *structure over this vocabulary.*

Definition 2.1.1: The *universe* U is a finite subset of $\mathcal{U}$. A *relation scheme* R is a subset of U. A *database scheme* D over U is a set of relation schemes with union U.

Definition 2.1.2: Let D be a database scheme over U, R a relation scheme and X a subset of U. An X-*tuple* t is a mapping from X into Δ, such that each A in X is mapped to an element of $\Delta[A]$. A *relation* r over R is a finite set of R-tuples. A *database* d over D is a set of relations; one relation over each relation scheme of D.

We omit the qualifications (over ...) in the above definitions when they are clear from the context; also we use tuple instead of X-tuple if X is understood. We use $D, D_1, \ldots$, for database schemes and $R, R_1, \ldots$, for relation schemes. We denote tuples by $t, t_1, \ldots$, relations by $r, r_1, \ldots$, and databases by $d, d_1, \ldots$; finally, we use the notation $\alpha(r)$ for the relation scheme of relation r and $\alpha(d)$ for the database scheme of database d. A relation r over R is represented in figures by a "table" with "columns" named with the attributes of R and with the tuples as "rows".

Note that in database theory we define a relation (a database) to be a finite set. There are situations where it is useful to adopt a more liberal view and allow *unrestricted relations* (*unrestricted databases*) i.e., both finite and infinite sets of tuples. In these cases, we will explicitly use the term unrestricted.

Remark 2.1.3: Attribute notation is the norm in database theory literature and is quite useful. In Section 2.4, we will see some of its advantages. A relation scheme R, such that $|R| = n$ for some integer $n \geq 0$, and a relation symbol in logic, with *arity* n, are very similar concepts. The only minor difference is whether to use attributes or numbers to name "columns" of relations. The choice is largely a matter of convenience and there is the obvious translation based on some arbitrary ordering of the attributes. Hence, notation R is the only symbol used here in an overloaded fashion, which is always disambiguated by the context. We use R either as a relation scheme, or as a relation symbol, or as a relation variable.

We first define the operational component of the relational data model. This is the *relational algebra query language* proposed by Codd in [Cod1]. The programs of this language are algebraic expressions, which denote mappings between databases called *queries*. The language is based on a small number of primitive operations on relations, which we now describe as follows. In clause (1) we have the *scheme restrictions* required of the *argument* relation schemes and the scheme of the *result* relation; in clause (2) we have the semantics of each operation. It is easy to see that these operations are well defined on relations, i.e., the result is a relation provided the scheme restrictions are satisfied.

Let t be an X-tuple and Z a subset of X, then the *projection* of t on Z, denoted $t[Z]$, is the restriction of the mapping t on Z.

Projection $\pi_X(r)$ is the projection of r on X.

1. $X \subseteq \alpha(r)$ and $\alpha(\pi_X(r)) = X$.
2. $\pi_X(r) = \{t[X] : t \in r\}$.

Natural Join $r_1 \bowtie r_2$ is the (natural) join of r_1 and r_2.

1. no scheme restriction and $\alpha(r_1 \bowtie r_2) = \alpha(r_1) \cup \alpha(r_2)$.
2. $r_1 \bowtie r_2 = \{t : t \text{ is } \alpha(r_1) \cup \alpha(r_2)\text{-tuple, such that } t[\alpha(r_1)] \in r_1 \text{ and } t[\alpha(r_2)] \in r_2\}$.

Union $r_1 \cup r_2$ is the union of r_1 and r_2.

1. $\alpha(r_1) = \alpha(r_2)$ and $\alpha(r_1 \cup r_2) = \alpha(r_1)$.
2. $r_1 \cup r_2 = \{t : t \in r_1 \text{ or } t \in r_2\}$.

Difference $r_1 - r_2$ is the difference of r_1 minus r_2.

1. $\alpha(r_1) = \alpha(r_2)$ and $\alpha(r_1 - r_2) = \alpha(r_1)$.
2. $r_1 - r_2 = \{t : t \in r_1 \text{ and } t \notin r_2\}$.

Selection $\varsigma_{A=B}(r)$ is the selection on r by $A = B$.

1. $A, B \in \alpha(r)$ and $\alpha(\varsigma_{A=B}(r)) = \alpha(r)$.
2. $\varsigma_{A=B}(r) = \{t : t \in r \text{ and } t[A] = t[B]\}$.

Renaming $\varrho_{B|A}(r)$ is the renaming in r of A into B.

1. $A \in \alpha(r)$, $B \notin \alpha(r)$ and $\alpha(\varrho_{B|A}(r)) = (\alpha(r) - \{A\}) \cup \{B\}$.
2. $\varrho_{B|A}(r) = \{t : \text{for some } t' \in r,\ t[B] = t'[A] \text{ and } t[C] = t'[C] \text{ when } C \neq B\ \}$.

A feature that we have omitted from our relational algebra is *constants*, that is, symbols representing the elements of the domain. The presence of constants does not affect the essentials of database theory and they can always be added with a certain degree of notational difficulty, see [ChaH1, ChaH2]. We use query languages without constants throughout our exposition; except in Section 4.2.III where we study algebraic properties of the selections on constants, $\varsigma_{A=a}$.

(The symbol ς is used for selection, since the more familiar symbol σ will be used to denote a dependency).

Remark 2.1.4: Note that renaming allows the introduction of attributes not in $\alpha(r)$; this is important for the algebraization theorem of the next section. We could have defined the operations without scheme restrictions, by making the result empty if the restrictions are violated (see [Cha1]). The conventions we have chosen instead are quite common in the literature and do not affect any of the theorems. In the scheme restriction for renaming, by our assumption about the domain, $\Delta=\Delta[A]=\Delta[B]$. In general, a technical condition is used for the set $\mathcal{U}$ of attributes (see [ImiL1]): "For each attribute A, there is an infinite sequence of attributes $A_1, A_2, \ldots$, with the same attribute domain as A. These are the attributes A can be renamed into."

Definition 2.1.5: Let $D = \{R_1, \ldots, R_m\}$ be a database scheme, then the ***relational algebra expressions over*** D are the expressions E generated by the following grammar, subject to the scheme restrictions.

$$E := R_1|\ldots|R_m|\pi_X(E)|(E \bowtie E)|(E \cup E)|(E - E)|\varsigma_{A=B}(E)|\varrho_{B|A}(E)$$

It is clear that one may represent a relational algebra expression over D as a tree, i.e., the parse tree of its generation. Each internal node of this tree is labeled by a relational operation and each leaf by a relation scheme of D over universe U. For each expression there is an associated relation scheme $\alpha(E)$, easily determined from the scheme restrictions. (Note that, because of renaming, there might be attributes in $\alpha(E)$ that are not in U). The symbols of a relational expression are syntactic symbols, e.g., R_i is a relation variable. These symbols are interpreted in the following intuitive way.

A relational algebra expression E over D denotes a function from databases over D to databases over $\{\alpha(E)\}$. This (total) function is defined via a "bottom-up" evaluation of the *parse tree* of E. On *input* database d over D associate: (1) with each leaf of the tree with label R, the relation r over R of d, and (2) with each internal node of the tree, the result of its label operation with argument relations the relations associated with its successor nodes. The *output* $E(d)$ is the database of one relation associated with the tree's root.

If d consists of one relation r, we use the abbreviation $r' = E(r)$ for $\{r'\} = E(d) = E(\{r\})$. Let us illustrate Definitions 2.1.1, 2.1.2 and 2.1.5 with an example.

Example 2.1.6: Consider database scheme $D = \{R\}$, where the universe U is the same as the relation scheme $R = AB$. One can think of relation r over R (the only relation in our database d over D) as a directed graph. This graph has no isolated nodes, its arcs are the tuples of r and its nodes are the unary relation (i.e., set) $\pi_A(r) \cup \pi_B(r)$. Let E, E_1, E_2, and E_3 be the relational algebra expressions:

$$E = R, \quad E_1 = \varrho_{B|A}(\varrho_{C|B}(R)), \quad E_2 = (R \bowtie \varrho_{B|A}(\varrho_{C|B}(R)))$$

$$E_3 = (R \cup \varrho_{B|C}(\pi_{AC}(R \bowtie \varrho_{B|A}(\varrho_{C|B}(R)))))$$

Note that R is used as a relational variable and all these expressions are ***subexpressions*** of E_3, i.e., their parse trees are subtrees of its parse tree.

r

A	B
a_1	a_2
a_2	a_3
a_3	a_4
a_1	a_3

r_1

B	C
a_1	a_2
a_2	a_3
a_3	a_4
a_1	a_3

r_2

A	B	C
a_1	a_2	a_3
a_2	a_3	a_4
a_1	a_3	a_4

r_3

A	B
a_1	a_2
a_2	a_3
a_3	a_4
a_1	a_3
a_2	a_4
a_1	a_4

Figure 1: The relations of Example 2.1.6

Let r be as in Figure 1, then we have $r = E(r)$. Also, $r_i = E_i(r)$, for $i = 1,2,3$, the other relations in Figure 1. Note how renaming just manipulates the relation scheme, how join works and how duplicates are eliminated when projection or union are performed. To see the intuitive meaning of these expressions let us examine $r_3 = E_3(r)$. For any relation r, the relation r_3 consists of those pairs of nodes $< a, b >$ of the graph represented by r, such that there is a directed path of one or two arcs from a to b. □

Definition 2.1.7: Let E_1, E_2 be relational algebra expressions over D such that $\alpha(E_1) = \alpha(E_2)$. We say that E_1 is *contained* in E_2, denoted $E_1 \subseteq_e E_2$, if for all databases d over D we have that every tuple in $E_1(d)$ is also in $E_2(d)$. We say that E_1 and E_2 are *equivalent*, denoted $E_1 \equiv_e E_2$, if $E_1 \subseteq_e E_2$ and $E_2 \subseteq_e E_1$ hold.

It is possible to define other relational operations, besides *Projection, Join, Union, Difference, Selection* and *Renaming*. For instance, one can define *Intersection* as the obvious set theoretic intersection on relations with the same scheme. However, *Join* on relations with the same scheme does exactly that. For a large repertoire of relational operations, e.g., *Cartesian-Product, Equi-Join, Division*, we can just use an equivalent expression built from the basic set of operations we chose. Another such example is *Select* with conditions that are propositional, (i.e., $\vee, \wedge, \neg$,) combinations of atoms of the form $A = B$. For detailed expositions of relational algebra we refer to any of the standard database textbooks, e.g., [Mai1, Ull1].

Relational operations and relational algebra expressions can be defined as above (verbatim) for unrestricted relations. This allows us to define, as in Definition 2.1.7, *unrestricted containment* and *unrestricted equivalence*, which we will carefully distinguish from *containment* and *equivalence*. Testing for expression containment and equivalence are central computational problems for expression optimization. The unrestricted notions imply the finite ones (since for them d ranges over finite and infinite databases), but they are not necessarily the same. Let us see why.

Testing for containment is *co-recursively enumerable* (co-r.e.), by going through all possible inputs (i.e., all finite d's) and simultaneously checking for noncontainment via the bottom-up evaluation. But testing unrestricted containment is *recursively enumerable* (r.e.), by reducing it to the validity of a sentence of the *first-order* (f.o.) *predicate calculus with equality* [End] and using the Gödel completeness theorem. For this reduction one can use the correspondence between relational algebra expressions and f.o. formulas from Section 2.2.I. Thus, equality of the finite and unrestricted notions would imply decidability; the problems would be r.e. and co-r.e. and therefore *recursive*.

This equality is not true in general because: unrestricted containment and equivalence of relational expressions are *undecidable*. This can be shown to follow from the undecidability of validity of f.o. sentences. For the undecidability of unrestricted containment of special subclasses of relational expressions see [ImiL1], [HenMT]. Containment and equivalence of relational expressions are also undecidable; this can be shown using the undecidability of validity over finite structures of [Tra].

For many cases of interest, we have equality of the finite and unrestricted notions and thus *decidability*. For instance, one such case is for the expressions built using only *Projection* and *Join*, the class of *project-join* expressions. Database theory has developed around properties of project-join expressions. Here is an example, which highlights some important project-join containment properties.

Example 2.1.8: The operation of join is *associative* and *commutative*. Therefore, if $r_1, \dots, r_m$ are relations respectively over $R_1, \dots, R_m$ then the join $r_1 \bowtie \dots \bowtie r_m$ is unambiguously defined. One property of this join is that $\pi_{R_j}(\bowtie \{r_i : 1 \leq i \leq m\}) \subseteq r_j$, for $1 \leq j \leq m$.

Let $D = \{R_1, \dots, R_m\}$ be a database scheme, whose relation schemes have union R. If r is a relation over R we call the set of relations $\{\pi_{R_1}(r), \dots, \pi_{R_m}(r)\}$ a (full) *decomposition* of r, and we denote it by $\pi_D(r)$. We use $\bowtie \pi_D(r)$ for $\bowtie \{\pi_{R_i}(r) : 1 \leq i \leq m\}$. An important property that holds for decompositions is that: for all r, $r \subseteq \bowtie \pi_D(r)$.

If $r = \bowtie \pi_D(r)$ holds then r has a *lossless decomposition* over D, else it has a *lossy decomposition*. The idea here is that by decomposing relation r into its projections we lose no information when the decomposition is lossless, because then we can get the original relation back by joining the projections. The importance of lossless decompositions was identified in [AhoBU, Ris].

In relational algebra expression notation, we have the following containments, i.e., these are *identities*: (1) $R \subseteq_e \bowtie \pi_D(R)$, see above, and (2) $\bowtie \pi_D(\bowtie \pi_D(R)) \equiv_e \bowtie \pi_D(R)$, idempotence. □

We now turn to the specification component of the relational data model, the functional dependency (fd for short). This type of constraint can be thought of as a generalization of "keys-of-records". It was originally identified by Codd in [Cod2]. A tuple of a relation represents a relationship among certain values, but by itself it does not provide any information about the nature of this relationship. For example, it would be useful to know that only relations representing functional relationships are acceptable. This is exactly what the addition of fd's accomplishes.

Definition 2.1.9: Let R be a relation scheme and X, Y be subsets of R. An expression of the form $X \rightarrow Y$ is called a *functional dependency over* R (fd over R). The fd $\sigma = X \rightarrow Y$ is satisfied by relation r over R, when: for all tuples t_1, t_2 in r, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. Relation r satisfies a set Σ of fd's if r satisfies all the fd's in Σ.

A set Σ of fd's over R can be viewed as a semantic specification of a set of databases over database scheme $\{R\}$. That is, Σ specifies the set of databases satisfying Σ. Relations that do not satisfy some σ in Σ cannot be "possible worlds" of the application being modeled. We denote this specification as the pair $< \{R\}, \Sigma >$. This basic specification mechanism is developed in the theory of *dependencies*, through extensions and variations of fd's. As we shall see, fd's can be represented as sentences of the f.o. predicate calculus with equality. Moreover, they have many interesting and qualitatively different representations.

Example 2.1.10: Let R be ABC. Consider the relation r over $\alpha(r) = R$ in Figure 2. It is easy to verify that r satisfies the set of fd's $\Sigma = \{A \rightarrow B, B \rightarrow C, C \rightarrow B\}$ and that it does not satisfy the fd's $B \rightarrow A, BC \rightarrow A$. To see an intuitive application of this example: think of A as EMPLOYEE, B as DEPARTMENT and C as MANAGER in a COMPANY database. Every employee

r	A	B	C
	a	b	c
	a'	b	c
	a''	b'	c'

Figure 2: The database of Example 2.1.10

works in a unique department, every department has a unique manager and every manager directs a unique department. □

Two general concepts have emerged from the definitions and examples. A mapping between databases can be described by an expression of a query language. For these expressions testing containment is a fundamental problem. A set of databases can be specified by a database scheme coupled with dependencies. In the last example of this section let us indicate how *containment equations* can be used instead of dependencies. This theme is developed in [YanP], which is an in-depth study of project-join expression containments.

Example 2.1.11: The pair $< \{ABC\}, \{A \rightarrow B\} >$ specifies the set of relations as follows: $FD = \{ \ r : r$ over ABC and satisfying $A \rightarrow B \ \}$. Using containment we can also specify a set of relations $CN = \{ \ r : r$ over ABC and satisfying $E(r) \subseteq E'(r) \ \}$; where E, E' are the following relational algebra expressions over $R = ABC$ with $\alpha(E) = \alpha(E') = BB'$,

$$E = \pi_{BB'}(\pi_{AB}(R) \bowtie \pi_{AB'}(\varrho_{B'|B}(R))), \quad E' = \varsigma_{B=B'}(E)$$

A simple argument suffices to show that the two sets of relations FD and CN are identical. This technique can be generalized to any set of fd's. □

Additional Bibliographic Comments 2.1.12: The definitions of the relational data model and the insight that they can be used as a foundation of database management are due to Codd [Cod1, Cod2, Cod3, Cod4, Cod5]. The best known early pioneer of the idea of relations as databases is Childs [Chi]; see [Ull3] where some of the prehistory of the field is investigated.

Relational algebra, as indicated in [ImiL1], can be studied as a *cylindric algebra*. There has been a great deal of work on the subject of cylindric algebras, much of it by Tarski and his school, see [HenMT] for a standard exposition. Even in this well studied mathematical setting, the problems and the solutions of database theory are often novel. This is because of the practical motivation and the emphasis on computational complexity and on finite structures. Database theory has also contributed directly to the theory of cylindrical algebras, e.g., the *cylindric lattices* of [Cos3]. □

2.2 Why Relational Algebra?

Our goal in this section is to demonstrate that, the choice of relation as the only data-type and of *manipulating this data-type only via the relational algebra operations* is by no means ad-hoc.

2.2.I Algebra = Calculus

The relational algebra query language is a typical "procedural" programming formalism. The basic observation of Codd [Cod1] is that this "procedural" language has a very natural "declarative" counterpart, the *relational calculus query language.*

For the purposes of this presentation it suffices to describe the relational calculus in an informal but intuitive fashion. As we mentioned in Remark 2.1.3 relation scheme R and relation symbol R are similar concepts, modulo an ordering of the attributes. Under this simple translation the database scheme $\{R_1, \ldots, R_m\}$ becomes a vocabulary of relation symbols. We further assume that $\mathcal{V}$ is a countably infinite set of *variables*, disjoint from the attribute set $\mathcal{U}$ and the value set Δ (we denote variables using $x, y, z, x_1, \ldots$). For a detailed treatment of the relational calculus we refer to [Mai1], which uses relation schemes and to [Ull1], which uses relation symbols.

The *formulas* of the f.o. predicate calculus with equality over vocabulary $\{R_1, \ldots, R_m\}$ are built out of: *atomic formulas* of the form $x = y$ and $R_i(x_1 \ldots x_{n_i}), 1 \leq i \leq m$ (where R_i has arity $n_i \geq 0$) using the *propositional connectives* $\vee, \wedge, \neg$ and *quantifiers* $\forall x, \exists x$. *Free* and *bound* variables are defined in the standard fashion [End]; *sentences* are formulas without free variables. Let us denote by $\varphi(x_1, \ldots, x_n)$ a f.o. formula with exactly n *distinct* free variables $x_1, \ldots, x_n$.

There is a connection at the definition level between database theory and finite model theory. If d is a database then the set of values that occur in d is finite, let us call it δ. (To be technically consistent with mathematical logic if d is empty $\delta = \{a\}$ for some fixed value a. In fact we could develop the theory using as the domain δ some superset of the values occurring in d, see Section 4.1.) Thus, given database d over $\{R_1, \ldots, R_m\}$ we have a finite f.o. relational structure $< \delta, d >$.

Let $\varphi(x_1, \ldots, x_n)$ be a formula as above, let d be a database whose relations are *interpreting* the relation symbols $\{R_1, \ldots, R_m\}$, let δ be as above and let $a_1, \ldots, a_n$ be values from δ *interpreting* the n variables $x_1, \ldots, x_n$, respectively. We can define *satisfaction* of $\varphi(a_1, \ldots, a_n)$ by $< \delta, d >$ in the standard f.o. fashion [End]; it is denoted by $< \delta, d > \models \varphi(a_1, \ldots, a_n)$.

Definition 2.2.1: A *relational calculus expression* F over $\{R_1, \ldots, R_m\}$, with $\alpha(F) = R$, is an expression of the form $\{R, x_1, \ldots, x_n : \varphi(x_1, \ldots, x_n)\}$, where R is a relation symbol of arity $n \geq 0$ and φ is a f.o. formula with exactly n distinct free variables. F defines a function from databases over $\{R_1, \ldots, R_m\}$ to databases over $\{R\}$ as follows: on input database d, with set of values δ, the output $F(d)$ is the database consisting of the relation $\{a_1 \ldots a_n :< \delta, d > \models \varphi(a_1, \ldots, a_n)\}$.

Query language expressions, in general, are used to denote functions of some *type*: from databases over $\{R_1, \ldots, R_m\}$ to databases over $\{R\}$. Using Definition 2.1.7 we can define containment and equivalence for expressions of the same type, e.g., $E \equiv_e F$ for a relational algebra expression E and a relational calculus expression F. Hence, when we mention expression equivalence or containment we will imply that the types of the expressions are the same. Similar comments apply to unrestricted equivalence and containment.

Theorem 2.2.2: *Every relational algebra expression can be translated, in time polynomial in its size, into an equivalent relational calculus expression. Every relational calculus expression can be translated, in time polynomial in its size, into an equivalent relational algebra expression.*

The proof of this theorem, known as Codd's theorem [Cod1, Cod3], is by structural induction on algebra and calculus expressions. The arguments are simple, but the theorem has great practical significance. The efficient translation of calculus into algebra reveals a "procedural" evaluation for the function defined "declaratively" by a calculus expression. The expressive power of other query languages can be assessed, based on whether they can express or not all the calculus or algebra queries. We will use the term PTIME *language equivalent* for the two conditions of Theorem 2.2.2. So, *relational algebra and relational calculus are* PTIME *language equivalent.*

For an example, the relational expression E_3 from Example 2.1.6 is equivalent and unrestricted equivalent to the relational calculus expression: $\{R', x, y : R(xy) \vee \exists z(R(xz) \wedge R(zy))\}$.

Let d be an unrestricted database. The satisfaction $<\Delta, d> \models \varphi(a_1, \ldots, a_n)$, is defined as in the finite case, only Δ is used instead of δ and $a_1, \ldots, a_n$ are from Δ. If we keep the syntax of relational calculus, but use Δ instead of δ for the semantics, we have: *the relational calculus with unrestricted domain.* This is a query language for unrestricted databases, because the outputs could be infinite even for finite inputs. To see this, consider $\{R', x : \neg R(x)\}$, where if the input is finite the output is infinite because negation is with respect to Δ.

Remark 2.2.3: For the relational calculus with unrestricted domain there is a theorem analogous to 2.2.2: *The relational calculus with unrestricted domain and the relational algebra with "complement" are* PTIME *language unrestricted equivalent.* In this case we have to extend relational algebra on unrestricted databases with the operation of *complement*: $\neg(r) = \{t : t \notin r\}$, where $\alpha(\neg(r)) = \alpha(r)$. Both these query languages are over unrestricted databases, so we use unrestricted equivalence.

There is one flaw with the relational calculus. For some expression F it might be possible to find a database input d, such that the output is different in the relational calculus from the relational calculus with unrestricted domain. In these cases F is called *domain dependent* and using Δ instead of δ matters, e.g., in $F = \{R', x : \neg R(x)\}$. If no such databases d exist, then F is called *domain independent.* Domain independent formulas have been proposed as a class of "well formed" relational calculus expressions. The rationale here is that: when one evaluates a domain dependent formula one usually computes all of δ; this can be wasteful even if feasible and should be avoided if possible.

Unfortunately, the class of domain independent formulas is not recursive [Dip, Var2]. Fortunately, they have recursive subclasses for which Theorem 2.2.2 can be refined. One example, are the *safe formulas* of [Ull1]. For these one can show: *The relational calculus, the safe relational calculus and the relational algebra are all* PTIME *language equivalent* [Mai1, Ull1]. Note that domain independent formulas are defined semantically, whereas safe formulas are defined syntactically. Another interesting fact is that: *given a relational calculus expression F and a database d, it is decidable if the output is the same in the relational calculus and in the relational calculus with unrestricted domain*, see [AylGSS].

Codd's theorem can be specialized to subsets of the relational calculus and algebra. The *positive existential calculus* of [ChaH2] consists of the relational calculus expressions built with $\exists, \vee, \wedge, =$ but without $\forall, \neg$. The relational algebra without difference consists of the relational algebra expressions built using all the defining operations of the algebra, except for *Difference.*

Theorem 2.2.4: *The positive existential relational calculus and the relational algebra without difference are* PTIME *language equivalent.*

Additional Bibliographic Comments 2.2.5: Kuhns in [Kuh] was among the first to propose using the predicate calculus as a database query language. Codd's theorem originally appeared in [Cod1, Cod3] for an algebra and a calculus with constants, comparators ($\leq$), and with variables in the calculus ranging over tuples. We presented a pure version of these languages (without constants or $\leq$) and a domain as opposed to a tuple relational calculus.

Let us comment on the relationship between Codd's theorem and cylindric algebras. A version of this theorem was known early on in the logic community, e.g., [ChiT, TarT]. It is used to show Tarski's Algebraization Theorem, see [HenMT]. Tarski's Algebraization Theorem provides a more general "algebraization" of the f.o. predicate calculus with equality: because the data-types manipulated are not restricted to be relations and the properties of the operations are specified equationally. However, it was only after Codd's seminal papers that the practical implications of this work became clear.

Theorem 2.2.2 should be properly viewed as one instance of a theorem scheme. For example, an analogous theorem holds for a relational algebra with aggregate operations [Klu1], e.g., with operations *max, min*, etc. Similar theorems can be shown for other data models, see [AbiBe, DahM, KupV1]. In the area of complex object data models algebraization has been a central theme, see [Hul2] for a survey and [HulS, KupV2, ParV, Vgu] for recent developments in the area. Algebra = calculus theorems can be shown for subsets of the predicate calculus as in Theorem 2.2.4. For instance, an algebra for the f.o. predicate calculus without equality can be found in [Cha1].

The notion of domain independent formulas is due to Fagin in [Fag6] (the definition used here is a variation of the original one). We refer the reader to [Fag6] for a review of a number of syntactically defined sets of formulas, that can be used instead of the safe formulas of [Ull1]. A recent analysis of domain independence can be found in [VgeT]. □

2.2.II LOGSPACE Data Complexity

We assume some familiarity with complexity classes for sequential computation such as the classes LOGSPACE, NLOGSPACE, PTIME, NP (NPTIME), PHIER, PSPACE, EXPTIME, with the class NC for parallel computation, and with the notion of logspace-completeness in a complexity class; see the surveys [GarJ, Coo]. All these classes are properly contained in EXPTIME. The following containments ($\subseteq$) are conjectured to be proper ($\subset$).

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{NC} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PHIER} \subseteq \text{PSPACE} \subset \text{EXPTIME}$$

Fagin's work in [Fag1] was crucial in establishing the first link between computational complexity and finite model theory (and thus database theory) through a logical characterization of NP. We will return to this subject in Section 4, where we examine a whole spectrum of possible query languages; see also the survey [Imm4]. In this subsection, we introduce the basic definition of data complexity from [ChaH2] and we examine the data complexity of relational algebra queries.

Query language expressions denote queries, i.e., functions from databases to databases. What is the computational complexity of these functions? Let us assume that a database d, represented in some standardized binary encoding, has size $|d|$. This size typically dominates, by many orders of magnitude, the size of a query expression and is therefore the asymptotic parameter of interest. In other words, we may assume that the query expression E has size bounded by some constant, since we can exhaustively analyze it before we apply it to the database.

Definition 2.2.6: The *data complexity* of the query denoted by expression E is the computational complexity of testing membership in the set $\{ < t, d > : \text{tuple } t \text{ is in } E(d) \}$. We say that a query is in a computational complexity class if its membership problem is in this class; and that it is logspace-complete in this class if its membership problem is.

The following observation complements the effective translation of Theorem 2.2.2, by emphasizing that the evaluation can be performed efficiently with respect to the database. Given the potential large size of the database, PTIME algorithms are the only "reasonable" ones. Among them, NC algorithms are "preferable" because they facilitate the use of parallel processing.

Theorem 2.2.7: *Relational algebra or calculus queries are in* LOGSPACE.

Note that LOGSPACE $\subseteq$ NC, so this theorem states that there is a lot of potential parallelism in relational algebra queries. Much of the work on access methods realizes this promise, e.g., very efficient join algorithms for external storage. To prove Theorem 2.2.7 one uses the relational operation semantics and the fact that the width of the tables processed is fixed. The data complexity for many query languages, both declarative and procedural, is determined in [Var5].

There is a price to pay for the low data complexity. This is the limited expressive power of relational algebra or calculus. Let r be a relation representing an undirected graph and $Trans(r)$ its transitive closure. The *transitive closure* query maps r to $Trans(r)$, for all r.

Theorem 2.2.8: *The transitive closure query is not expressible in relational algebra or calculus.*

For unrestricted relations this theorem is still true and has an easy proof using the Compactness Theorem of first-order logic [End]. Unfortunately, compactness and other useful properties fail when attention is restricted to finite structures only [Gur1, Gur2]. Theorem 2.2.8 was first shown in [Fag2], in a stronger form using Ehrenfeucht-Fraisse game techniques. The stronger form is: *it is not possible to characterize connected graphs using a monadic existential second-order sentence over finite structures* (i.e., the second order quantifiers are over monadic relations, they form a prefix of the sentence and they are all existential). Other proofs appear in [AhoU, ChaH2, Gai, GaiV, Imm1].

Additional Bibliographic Comments 2.2.9: Theorem 2.2.8 has been generalized to certain classes of queries, [BeeKBR, Cos4]. Ajtai and Gurevich have recently announced that it holds for all "unbounded Datalog programs" (see Section 4.2.II). The difficulty here is the finiteness assumption, that requires the development of tools to replace the Compactness Theorem of first-order logic.

Data complexity is not the only measure of interest. If we assume that the query expression is part of the input then we have the notion of *expression complexity*. Vardi has shown in [Var5] that expression complexity is typically one exponential higher than data complexity. The expression complexity of project-join expressions is further investigated in [Cos1, HonLY, MaiSY].

From these investigations one can infer that: relational operations are less structured than common integer operations, exponentiation included. This is because, for relational algebra expressions, intermediate results of a computation can be much larger than both inputs and outputs. Circuits of relational operations are compared to Boolean circuits in [Yan2]. $\square$

2.2.III Query Optimization and Homomorphisms

The procedural nature of a query language facilitates the "compile-time" optimization of programs. Query optimization has been used to implement the relational data model in a reasonably efficient fashion, e.g., [AstEtal, SelEtal, WonY]. In these implementations, a relational algebra expression is algorithmically transformed into an equivalent expression, that is better by some measure of complexity. The subject of query optimization flourished in the 1970's and is still an active area of research; see [JarK] for a survey.

Two optimization ideas occur most often in the literature. We summarize them in the two bullets below. The rationale for these is as follows. The join has been identified as the most expensive relational operation and its cost grows with the size of the argument relations. Thus, it is preferable to apply it on relations after these have been reduced in size through selection. Also, it pays to perform as few joins as possible.

- This is the algebraic rewriting of expressions, with emphasis on the *propagation of selections before the joins* [AhoU]. An early use of this idea appears in [Pal]. A companion problem, usually solved by dynamic programming, is to find the *best order of join evaluations* in an expression $\bowtie \{r_i : 1 \leq i \leq m\}$.
- This is the *minimization of the total number of joins.*

We will defer the first idea to the context of more powerful query languages in Section 4.2.III. Here, we focus on the second one, which reveals the importance of *homomorphism* techniques for expression containment and database theory in general. Let us proceed with the formalization of the second question.

The set of variables $\mathcal{V}$ can be partitioned into disjoint subsets, one for each attribute in $\mathcal{U}$. Each attribute A corresponds to $\mathcal{V}[A] = \{x, x_1, x_2, \ldots\}$; x the lexicographically first one of these variables is called *distinguished* and the others *nondistinguished.* We also extend the notion of a *tuple* from a mapping into Δ to a mapping into $\Delta \cup \mathcal{V}$.

Definition 2.2.10: A *tableau* T over relation scheme R is a set of R-tuples, such that if tuple t is in T then $t[A]$ is in $\mathcal{V}[A]$. The *target relation scheme* $\alpha(T)$ of T is the set $\{A$: for some t in T, $t[A]$ is distinguished $\}$. The *summary* t_T of T is the $\alpha(T)$-tuple of its distinguished variables.

For examples of tableaux see Figure 3, Example 2.3.12. These examples appear in Section 2.3, in order to emphasize the intimate connections between tableaux and dependencies.

There are two ways of thinking about tableaux. (1) The first way is to consider them as relations over R with domain $\mathcal{V}$. In this case we can think of $T_1 \subseteq T_2$ as a set inclusion between two relations over the same scheme R. (2) The second way of thinking about tableaux is as expressions of the *tableau query language.* A tableau T denotes a (total) function from relations over R to relations over $\alpha(T)$ as follows. A *valuation* h is a mapping from $\mathcal{V}$ into Δ; valuation h can be extended to tuples componentwise. On input relation r over R the output $T(r)$ is a relation over $\alpha(T)$, where:

$T(r) = \{h(t_T) : h$ is a valuation such that $h(t)$ is in r for each tuple t of T $\}$.

As expressions tableaux define mappings from one relation databases to one relation databases. Expression equivalence and containment in this case are denoted by $T_2 \subseteq_e T_1$ and $T_2 \equiv_e T_1$, where we must have $\alpha(T_1) = \alpha(T_2)$.

A subtle point is that $\subseteq$ and $\subseteq_e$ are different concepts. In fact, if $T_1 \subseteq T_2$ and $\alpha(T_1) = \alpha(T_2)$ then $T_2 \subseteq_e T_1$; this is no coincidence.

A *homomorphism* h from T_1 to T_2 is a mapping from $\mathcal{V}$ into $\mathcal{V}$ such that: (1) $h(x) = x$ for x distinguished, and (2) $h(T_1) \subseteq T_2$ for $h(T_1) = \{h(t) : t$ tuple in T_1, h extending componentwise $\}$.

For example, if $T_1 \subseteq T_2$ then there is a homomorphism defined by the identity mapping from T_1 to itself as part of T_2. Note the similarity of definition between a valuation and a homomorphism, it is easy to see that a homomorphism composed with a valuation is a valuation. Using composition and both interpretations of tableaux one can show a simple, but central fact in database theory:

Theorem 2.2.11: *For two tableaux T_1, T_2 with the same summary we have that, $T_2 \subseteq_e T_1$ iff there is a homomorphism from T_1 to T_2.*

The homomorphism technique was first developed for *conjunctive queries* in [ChaM]. Conjunctive queries are a superset of tableau queries; see Remark 2.2.15 below. The basic results for tableau optimization (Theorems 2.2.11-2.2.13) are from [AhoSU1, AhoSU2]. The following two theorems illustrate the possible optimizations of relational expressions, using the tableau as a tool. To illustrate the expressive power of the tableau language we consider project-join relational expressions. There are tableau queries that cannot be expressed as project-join ones; see [YanP] for the precise expressive power of tableaux.

Theorem 2.2.12: *For every project-join expression over $\{R\}$ there is an equivalent tableau T over R constructed recursively as follows:* (1) *if $E = R$ then T is the tableau with one R-tuple of all distinguished variables,* (2) *if $E = \pi_X(E_1)$ and T_1 is the tableau of E_1 then T is obtained from T_1 by changing each distinguished symbol x in $\mathcal{V}[A]$ such that $A \notin X$ into a new nondistinguished symbol, and* (3) *if $E = (E_1 \bowtie E_2)$ and T_1, T_2 are the tableaux of E_1, E_2 then T is the union of the sets of tuples of T_1, T_2.*

Theorem 2.2.13: *Each tableau T has a minimal subset of its tuples T_{min} equivalent to T. This T_{min} is unique up to renaming of the nondistinguished symbols. If T is the tableau of the project-join expression E then the T_{min} above is the tableau of a project-join expression equivalent to E with a minimum number of joins.*

Tableau minimization is NP-complete. This follows from the results of [ChaM]. Tighter bounds, e.g., for project-join expressions, are derived in [AhoSU1, AhoSU2] as well as syntactic constraints on tableaux under which the problem can be solved in PTIME. Thus, in general, to minimize the number of joins one might have to examine all homomorphims from T to T; this is not as bad as it sounds because T is an expression and not a database.

Theorem 2.2.11 can be extended from a single tableau to sets of tableaux, all with the same summary. The output for such a set of tableaux is the union of the outputs of the single tableau queries. The following is from [SagY].

Theorem 2.2.14: $\{T_{21}, \ldots, T_{2m}\} \subseteq_e \{T_{11}, \ldots, T_{1n}\}$ *iff for each $T_{2j}, 1 \leq j \leq m$, there exists a $T_{1i}, 1 \leq i \leq n$, such that $T_{2j} \subseteq_e T_{1i}$.*

Remark 2.2.15: As expressions, tableaux have typed variables and a single relation input. We say that they are *typed and untagged*, where tags would correspond to multiple relation inputs. The above development can be carried-out for untyped variables and many relation inputs; such an expression is called an *untyped and tagged tableau* and is essentially the same as a conjunctive query of [ChaM]. The typed untagged case is particularly important in the theory of dependencies. On the other hand, unions of untyped tagged tableaux express the positive existential queries (see Theorem 2.2.4). This easily follows from putting positive existential formulas in disjunctive normal form. A consequence is that there is an algorithm for testing positive existential query containment.

Removing the typing does not affect any of Theorems 2.2.11–2.2.14. Untyped tableau queries T' are monotone, that is for all relations r_1, r_2 we have that $r_1 \subseteq r_2$ entails $T'(r_1) \subseteq T'(r_2)$. In addition, for a tableau T and for all relations r we have that $r \subseteq T(r)$; because there is always a homomorphism from T to the identity tableau (one tuple of all distinguished variables). Typing does, however, account for some additional special structure. A good source for the many properties that result from typing is [FagMUY].

Additional Bibliographic Comments 2.2.16: The homomorphism technique is related to *subsumption* techniques, that have been used since the early 1970's to determine the equivalence of logic programs. The algorithmic analysis was developed as part of database theory. We recommend [Mah] and [Sag6] for discussions of this technique for logic programming applications.

Tableaux can be extended to include constants and other features such as inequalities [Klu2]. They can be defined over databases that satisfy dependencies. This rich topic was initiated in [AhoSU1, AhSU2] for fd's. Containment in the presence of another common type of dependencies, *inclusion* dependencies (see Section 3), has been examined in [JohK, KanCV]. □

2.3 Why Functional Dependencies?

The study of dependency theory began with the introduction of fd's in [Cod2] and grew into a rich topic, interesting in its own right. In this section we present the fundamentals of dependency theory using fd's and some of their extensions, and we defer generalizations and applications to Section 3. We would like to emphasize that this choice is not only for reasons of exposition. Fd's are the most relevant dependencies from a practical standpoint. Also, most of the fundamental concepts were introduced and are best illustrated in this restricted context.

Throughout this section we assume one relation scheme with fd's defined on it.

2.3.I Dependency Implication and its Axiomatization

The following definition of implication applies to dependencies in general.

Definition 2.3.1: Let Σ be a set of dependencies and σ a single dependency. We say that Σ *implies* σ, $\Sigma \models \sigma$, if every unrestricted database that satisfies Σ also satisfies σ; Σ *finitely implies* σ, $\Sigma \models_f \sigma$, if every database that satisfies Σ also satisfies σ.

The importance of finite implication in database theory first became apparent in Bernstein's work [Ber]. As we shall see below, finite implication and implication coincide for fd's, as they often do in dependency theory. Although finite implication is the relevant notion from a practical standpoint, implication is also important because it is closely related to unsatisfiability of logical sentences.

Definition 2.3.2: Two sets of dependencies Σ, Σ' are *equivalent* if they are satisfied by the same set of databases. In this case we say that Σ is a *cover* for Σ' and vice-versa. If Σ is a cover of Σ' and is also a subset of Σ', then we say it is a *contained cover.*

Clearly, Σ' is redundant if it has a contained cover Σ that is a proper subset of it, because Σ is a more economical specification equally expressive as Σ'.

To come back to finite implication, it is easy to see that: Σ and Σ' are equivalent iff $\Sigma \models_f \sigma'$ for all σ' in Σ' and $\Sigma' \models_f \sigma$ for all σ in Σ. For fd's we can use $\models$ instead of $\models_f$, since these are the same. Testing if Σ' is redundant can be similarly reduced to finite implication.

Example 2.3.3: As was observed by Nicolas [Nic], fd's can be represented as sentences of the f.o. predicate calculus with equality. Let us demonstrate how this is done for universe ABC and the fd $\sigma = A \rightarrow B$. The vocabulary in the predicate calculus will be $\{R\}$, where the arity of R is 3 and A corresponds to the first argument of relation symbol R etc. The fd σ is expressed by the sentence:

$$\varphi_\sigma = \forall x \forall y \forall z \forall y_1 \forall z_1 (R(xyz) \wedge R(xy_1z_1)) \Rightarrow (y = y_1)$$

It follows from the definition of fd's that the set of finite relational structures satisfying φ_σ and the set of relations satisfying σ are the same. This is also true for unrestricted relations. Similar arguments show that any set of fd's can be expressed by a set of sentences of the f.o. predicate calculus with equality. Note, however, that the database notation for fd's is often preferable, because it is less cumbersome to use and it intuitively captures the meaning of fd's.$\square$

The identification of a dependency σ with a sentence φ_σ is true for fd's and for many other dependencies. One of its consequences is the reduction of dependency (finite) implication to the (finite) unsatisfiability problem of f.o. logic. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, then $\Sigma \models_{(f)} \sigma$ iff we have that the sentence $\varphi_{\sigma_1} \wedge \ldots \wedge \varphi_{\sigma_k} \wedge \neg\varphi_\sigma$ is (finitely) unsatisfiable.

Recall that a sentence is (finitely) unsatisfiable if it has no (finite) models. Also, unsatisfiability for the f.o. predicate calculus with equality is r.e., by the Gödel Completeness Theorem, and finite unsatisfiability is co-r.e., by enumerating and testing all finite structures. From unsatisfiability we can infer finite unsatisfiability, from finite satisfiability we can infer satisfiability and from implication we can infer finite implication (but the converses do not always hold). From this discussion it follows that if a set of dependencies is identified with a set of sentences, for which satisfiability and finite satisfiability coincide, then dependency implication and finite implication coincide and are decidable. This happens to be the case for fd's.

Theorem 2.3.4: *For fd's finite implication and implication are the same and decidable.*

The following argument for this theorem is somewhat of an overkill, since the theorem can be shown without an excursion into satisfiability. However, it is quite instructive since it may be used for many nontrivial extensions of fd's. Let us look at the structure of the sentence $\varphi_{\sigma_1} \wedge \ldots \wedge \varphi_{\sigma_k} \wedge \neg\varphi_\sigma$ in the fd case. This can be written as a $\exists^*\forall^*$-sentence, that is a sentence in prenex normal form whose quantifier prefix consists of a string of $\exists$ followed by a string of $\forall$. This is known as a sentence of the *initially extended Bernays-Shönfinkel class*, for which satisfiability and finite satisfiability coincide [DreG].

Implication is a semantic notion, which is commonly studied using the syntactic device of a *formal system* or *axiomatization.* An axiomatization (for a set of dependencies in general) consists of *axiom schemes* and *inference rules.* A *derivation* of a dependency σ from a set of dependencies Σ is a sequence of dependencies $\sigma_1, \ldots, \sigma_n$ such that $\sigma_n = \sigma$ and each σ_i in the sequence is either a member of Σ, or an instance of an axiom scheme, or follows from preceding σ_j's in the sequence via an instance of an inference rule. For an example see formal system FD below.

We denote the existence of a derivation of σ from Σ by $\Sigma \vdash \sigma$. We say that a formal system $\vdash$ ($\vdash_f$) is *sound* for (finite) implication if $\Sigma \vdash \sigma$ ($\Sigma \vdash_f \sigma$) entails $\Sigma \models \sigma$ ($\Sigma \models_f \sigma$); and it is *complete* if $\Sigma \models \sigma$ ($\Sigma \models_f \sigma$) entails $\Sigma \vdash \sigma$ ($\Sigma \vdash_f \sigma$).

Formal systems for fd implication and finite implication were first studied by Armstrong [Arm]. His original system was slightly different from the sound and complete system FD, which we use here. FD consists of one axiom scheme and two inference rules. If Σ, σ are dependencies over R then any subset of R can be substituted for X, Y and Z. It is also an example of a *k-ary* system for $k = 2$, because each inference rule has at most two antecedents.

FDr reflexivity axiom scheme: $\vdash X \to \emptyset$

FDt transitivity inference rule: $X \to Y$ *and* $Y \to Z \vdash X \to Z$

FDa augmentation inference rule: $X \to Y \vdash XZ \to YZ$

Theorem 2.3.5: *The system* FD *is sound and complete for (finite) implication of fd's.*

Let us comment on the proof. Soundness of a formal system for implication entails soundness for finite implication (the converse is not true for all dependencies) and it is usually straightforward to show. Completeness for finite implication entails completeness for implication (the converse is not true for all dependencies) and it is typically the harder property to show. For this direction one usually examines the combinatorics of derivations and, if σ is not derivable from Σ, then one constructs a counterexample database satisfying Σ and falsifying σ. See [Mai1, Ull1, Var8] for standard expositions of the counterexample construction.

A closer examination of the FD system reveals that, to derive σ from Σ, one need only restrict attention to derivations with attributes in Σ and σ. This observation and Theorem 2.3.5 are an alternative proof of Theorem 2.3.4.

Let us close this subsection with some additional properties of fd's. Fd's are domain independent sentences, as defined in Secton 2.2.I. This is important, because testing for their satisfaction does not depend on whether Δ or δ is used.

Let Σ be a set of dependencies from some class of dependencies Σ^*. The *closure* of Σ over Σ^* are all dependencies in this class implied by Σ; it is denoted by Σ^+. We say that an unrestricted relation is an *Armstrong* relation if it satisfies all dependencies in Σ^+ and *simultaneously* falsifies all those in $\Sigma^* - \Sigma^+$. An interesting property for the set of fd's over a fixed universe is that there exists a finite Armstrong relation for fd's.

Additional Bibliographic Comments 2.3.6: The first complete and sound axiomatization for fd's is from [Arm], but rules for fd's and some of their properties appeared early on in [DelC]. The concept of Armstrong relation is due to Fagin from [Fag6]; research on these relations is surveyed in [Fag5]. Their structure for fd's is examined in [BeeDFS] and their existence for more general statements is investigated in [Fag6]. They have been used as a tool for "database design by example" in [ManR].

The expressive power of fd's as a specification mechanism is examined in [Hull, GinZ]. For other "global" dependency questions different from implication, such as the existence of Armstrong relations, see [Var3]. □

2.3.II PTIME Computational and Algebraic Properties

The computational complexity of fd implication was considered by Beeri and Bernstein in [BeeB], who demonstrated that implication can be performed optimally in linear-time. This required a more detailed analysis than the decidability property, which follows from the equality of implication and finite implication.

Let Σ be a set of fd's over the universe, A, B attributes and X, Y sets of attributes from the universe. It is not hard to see that every fd $X \rightarrow Y$ is equivalent to the set of fd's $\{X \rightarrow B : B \in Y\}$. Let us define $closure(X,\Sigma) = \{A : \Sigma \models X \rightarrow A\}$. Given Σ, X, Y, if we can compute $closure(X,\Sigma)$ then, clearly, we can decide the fd implication $\Sigma \models X \rightarrow Y$ in the same amount of time. The following theorem and algorithm are due to Beeri and Bernstein [BeeB]. The algorithm can be made to run in linear-time with the appropriate data structures.

Theorem 2.3.7: *(Finite) implication of fd's is in linear-time.*

```
procedure closure($X$,$\Sigma$)
        ATRLIST := $X$ ; FDLIST := $\Sigma$ ;
        repeat
                erase all occurrences of attributes in ATRLIST
                from the left-hand sides of fd's in FDLIST ;
                for each $\emptyset \rightarrow Y$ on FDLIST do
                        ATRLIST := ATRLIST $\cup$ $Y$
        until no new attributes are added to ATRLIST in repeat loop
return closure := ATRLIST
end
```

Extensive use of this algorithm has been made in database scheme design. From Definition 2.3.2 recall the notions of *cover* and *contained cover* for fd's. Such covers are *minimum* if they contain the minimum number of fd's possible. As shown in [Mai2], it is possible to compute minimum covers of fd's in quadratic-time. An algorithm for minimum contained covers is presented in [BeeB], where it is also shown that this computational task is NP-complete.

Theorem 2.3.8: *Minimum covers of fd's can be computed in* PTIME *and minimum contained covers in* NP*; but deciding if there is a contained cover with less than k fd's, for a given k, is* NP*-complete.*

In the previous subsection we established that every fd σ (every set of fd's Σ) can be associated with a sentence φ_σ (a set of sentences φ_Σ). From the relationship of dependency (finite) implication and (finite) unsatisfiability it follows that: $\Sigma \models_{(f)} \sigma$ iff $\varphi_\Sigma \models_{(f)} \varphi_\sigma$, where the second $\models_{(f)}$ is (finite) implication for sentences of the f.o. predicate calculus with equality. This is not the only relationship with mathematical logic; fd's have a number of elegant algebraic properties.

The Problem of Dependency Implication for Two Tuple Relations: The counterexample relation in the proof of Theorem 2.3.5, that is used for showing completeness, may be chosen to have only two tuples. Thus, for fd's one can show that: $\Sigma \models_{(f)} \sigma$ iff $\Sigma \models_2 \sigma$, where $\models_2$ is dependency implication over two-tuple relations. $\Box$

The Problem of Implication for Propositional Horn Clauses: One need not go to the f.o. predicate calculus with equality to represent fd's, it suffices to consider propositional Horn clauses. Without loss of generality, we have fd's with single attribute right-hand sides.

Now translate attribute A into a propositional constant A, translate fd $\sigma = A_1 \dots A_n \rightarrow A$ into a propositional Horn clause $prop_\sigma = A_1 \wedge \dots \wedge A_n \Rightarrow A$, and a set of fd's Σ into the obvious set of propositional Horn clauses $prop_\Sigma$. One can show that: $\Sigma \models_{(f)} \sigma$ iff $prop_\Sigma \models prop_\sigma$, where the second $\models$ is propositional sentence implication. The converse of this reduction also holds, i.e., propositional Horn clause implication is immediately reducible to fd implication. Thus, Horn clause implication can be decided in linear-time. $\Box$

The Generator Problem for Finitely Presented Algebras: Let Γ be a finite set of *generators* (function symbols of arity 0) and O a finite set of *operators* (function symbols of arity > 0). A (ground) *term* τ is built out of $\Gamma \cup O$ in the standard f.o. fashion and a (ground) *equation eq* is a statement the form $\tau = \tau'$. Equations are thus sentences of the f.o. predicate calculus with equality over vocabulary $\Gamma \cup O$ and they are satisfied by structures over this vocabulary, called *algebras*, according to the semantics of the calculus. If eq_Σ is a finite set of equations and eq_σ an equation then the implication $eq_\Sigma \models eq_\sigma$ is defined as in the caclulus.

The uniform word problem for fin·'tely presented algebras is the problem of deciding $eq_\Sigma \models eq_\sigma$, given a finite set of equations eq_Σ and an equation eq_σ. *The generator problem for finitely presented algebras* is defined as follows: on input (1) a finite set of equations eq_Σ over $\Gamma \cup O$, (2) an element γ of Γ, and (3) a subset Γ' of Γ, decide if there exists a term τ over $\Gamma' \cup O$ such that $eq_\Sigma \models \gamma = \tau$. If the answer is yes we denote this by $Generator(eq_\Sigma, \gamma, \Gamma')$. Both the uniform word problem and the generator problem are shown to be in PTIME in [Koz]. For the generator problem the algorithm of [Koz] is a generalization of the [BeeB] algorithm for fd closure and was independently derived.

To see the relationship of fd closure with the generator problem, translate each attribute A of the universe into a generator A. Translate each fd $\sigma = A_1 \dots A_n \rightarrow A$ into eq_σ that is the equation $A = f_\sigma(A_1 \dots A_n)$, and a set of fd's into the obvious finite set of equations eq_Σ. The universe has become the set of generators and the function symbols for the fd's in Σ (one per fd) form the set of operators. One can show that: $\Sigma \models_{(f)} X \rightarrow A$ iff $Generator(eq_\Sigma, A, X)$. □

The Uniform Word Problem for Γ-Semilattices: A Γ-semilattice is an algebra $< \Lambda, \odot, \Gamma >$, where Λ is a nonempty set, the *carrier* of the algebra, $\odot$ is a binary associative commutative idempotent operation on the carrier, and Γ is a finite set of named elements of the carrier. It is a finite Γ-semilattice if the carrier is finite. A (ground) term τ built out of operator $\odot$ and generators Γ is interpreted over a Γ-semilattice in the natural way; the meaning of τ is the element of the carrier computed by interpreting $\odot$ as the binary operation and Γ as the corresponding named elements of the carrier. A (ground) equation leq has the form $\tau = \tau'$ and is satisfied by a Γ-semilattice if τ and τ' have the same meaning in this semilattice. We say that a set of equations leq_Σ (finitely) implies equation leq_σ, denoted $leq_\Sigma \models^l_{(f)} leq_\sigma$, if every (finite) Γ-semilattice that satisfies every equation in leq_Σ also satisfies equation leq_σ. Note that this is a uniform word problem. But the algebra is no longer finitely presented, because the presentation consists of leq_Σ together with associativity commutativity and idempotence axioms for $\odot$. These properties are expressible using equations with variables, i.e., *nonground* equations.

To see the relationship with fd implication, translate each attribute A of the universe into a generator A, translate each fd $\sigma = A_1 \dots A_n \rightarrow A$ into leq_σ which is the equation $A_1 \odot \dots \odot A_n \odot A = A_1 \odot \dots \odot A_n$, and a set of fd's Σ into the obvious set of equations leq_Σ. One can show that: $\Sigma \models_{(f)} \sigma$ iff $leq_\Sigma \models^l_{(f)} leq_\sigma$. It is also true that finite implication and implication are the same over Γ-semilattices. □

We have seen six qualitatively different formulations of fd implication, which we summarize below. In all these statements finite implication and implication coincide.

Theorem 2.3.9: *Let Σ be a set of fd's and σ be fd $X \rightarrow A$ then: $\Sigma \models_{(f)} \sigma$ iff $\Sigma \models_2 \sigma$ iff $\varphi_\Sigma \models_{(f)} \varphi_\sigma$ iff $prop_\Sigma \models prop_\sigma$ iff $Generator(eq_\Sigma, A, X)$ iff $leq_\Sigma \models^l_{(f)} leq_\sigma$.*

Additional Bibliographic Comments 2.3.10: The connection of fd's, two-tuple implication and propositional logic was first identified by Fagin. It has been extended to the class of functional and *multivalued* (see Section 2.3.III) dependencies in [SagDPF].

The connection of fd's and generator problems was established in [CosK1], where it was also extended to the class of functional and inclusion dependencies (see Section 3), in two ways. One of these extensions holds for implication and the other for functional and *unary inclusion* (see Section 2.3.III) dependency finite implication.

The connection of fd's and Γ-semilattices is part of the "folklore" of database theory. It has been extended to the class of *partition* dependencies and Γ-lattices in [CosKS]. A Γ-lattice $< \Lambda, \odot, \oplus, \Gamma >$ has two operations $\odot$ and $\oplus$, such that $< \Lambda, \odot, \oplus >$ is a *lattice* and Γ are named elements of Λ. A partition dependency is defined as an equation between ground terms over $\odot, \oplus$ and elements of Γ. A partition dependency can be interpreted as a statement about relations over the universe and it generalizes the functional dependency. As shown in [CosKS], because of the lattice semantics, there are certain partition dependencies which are not expressible using f.o. sentences; despite this, partition dependency implication is in PTIME and is the same as finite implication. $\square$

2.3.III Functionality, Decomposition and Inclusion

Decomposing a relation r over R into a set of relations over $D = \{R_1, \ldots, R_m\}$ with $R = U = \cup_{i=1}^{m} R_i$ (i.e., computing $\pi_D(r) = \{\pi_{R_1}(r), \ldots, \pi_{R_m}(r)\}$ as in Example 2.1.8) is a technique commonly used in storing, querying and updating data. One can think of "the world" as being relation r and of $\pi_D(r)$ as its convenient representation.

The set of "possible worlds" is specified by $< \{U\}, \Sigma >$, where Σ is a set of dependencies over U. This simplifying framework is known as the *pure universal relation assumption*, where U is the universe, r is a universal relation satisfying Σ, and $< \{U\}, \Sigma >$ denotes the set of universal relations satisfying Σ.

A *(full) decomposition* π_D is thus a function from $< \{U\}, \Sigma >$ into the set of databases over D. The least one could require of this function is that it be *injective*, i.e., one-to-one. This would guarantee that it has a left inverse, called the *reconstruction* function, from the range of π_D onto $< \{U\}, \Sigma >$. The existence of the reconstruction function allows a decomposition without loss of information, since: "the parts can be put together again to form the original whole". This is called the *representation principle* in [BeeBG], an early and good survey of database scheme design. A desirable candidate for the reconstruction function is the join of the relations over D. Note that *join reconstruction* is a desirable feature, but it does not always follow from injectiveness [Var4], additional conditions might be required.

Decomposition π_D is *lossless* if for all r in $< \{U\}, \Sigma >$ we have that $r = \bowtie \pi_D(r)$. If a decomposition is lossless then it is easy to see that it is injective and that join reconstruction is possible. One of the first results in database theory involved conditions for lossless decompositions into two parts, in the presence of one fd [DelC, Hea].

By taking $\cup_{i=1}^{m} R_i = R \subseteq U$ ($\subseteq$ instead of $=$) it is possible to express *embedded decomposition* properties of the universal relation (instead of only full ones). To describe embedded lossless decompositions we add assertions to our dependency language of the form: for all r, $\pi_R(r) = \bowtie \pi_D(r)$, (note that $\pi_R(r) \subseteq \bowtie \pi_D(r)$ is a tautology). These are typed statements, in the sense that they only involve comparisons of values of the same attribute; this is a similarity with fd's.

Statements about embedded lossless decompositions are natural integrity constraints. The importance for database scheme design of two-part full lossless decompositions, called multivalued dependencies, and of two-part embedded lossless decompositions, called embedded multivalued dependencies, was identified early on in [Del, Fag3] and [Zan1]. Multi-part lossless decompositions, called join and embedded join dependencies, were first defined and studied in [AhoBU] and [Ris].

Inclusion dependencies, first identified in [CasFP], are another useful class of statements about relational databases. Here we concentrate on unary inclusion dependencies, the simplest (and most common) inclusion dependencies. For example, they can be used to express referential integrity and ISA constraints. To describe them it suffices to add assertions to our dependency language of the form: for all r, $\pi_A(r) \subseteq \pi_B(r)$. They are unlike fd's, because they are untyped. Also, unlike fd's, they can be used as constraints between two different relations in a more general multi-relational setting.

In the following definition we summarize the additions, motivated by decomposition and inclusion properties, to our repertoire of dependencies.

Definition 2.3.11: Let D be a database scheme whose $m \geq 2$ relation schemes have union $R \subseteq U$, where U is the universe. *Embedded join dependency* (ejd) $\bowtie [D]$ is satisfied by relation r over U if $\pi_R(r) = \bowtie \pi_D(r)$. It is called: a *join dependency* (jd) if $R = U$; an *embedded multivalued dependency* (emvd) if $m = 2$; a *multivalued dependency* (mvd) if $R = U$ and $m = 2$. The mvd $\bowtie [\{XY, XZ\}]$ with $Z = U - XY$ is denoted by $X \twoheadrightarrow Y$. *Unary inclusion dependency* (uind) $A \subseteq B$ is satisfied by relation r over U if $\pi_A(r) \subseteq \pi_B(r)$.

Tableau notation (recall Section 2.2.III) is often used in the context of fd's and ejd's. Every ejd $\sigma = \bowtie [D]$ is associated with a tableau T_σ, which is the one equivalent to the expression $\bowtie \pi_D(U)$ (see Theorem 2.2.12). So its summary is an R-tuple of distinguished variables, where $R \subseteq U$. Every fd $\sigma = X \rightarrow Y$ is associated with a tableau T_σ, which is the one associated with the mvd $X \twoheadrightarrow Y$. Note that for both fd's and jd's the summary of the associated tableau is a U-tuple of distinguished variables.

Clearly, every mvd, jd and emvd is an ejd. Also, every mvd is an emvd and a jd. The following example illustrates most of these definitions.

Example 2.3.12: Let $U = ABCA'$ and let fd $AA' \rightarrow C$, ejd $\bowtie [\{AB, AC\}]$ and uind $C \subseteq B$ be the dependencies specifying the "legal" universal relations.

The relation r in Figure 3 satisfies the three dependencies. The two tableaux of Figure 3 correspond to the fd $AA' \rightarrow C$ and to the emvd $\bowtie [\{AB, AC\}]$ respectively. The first tableau also corresponds to to the mvd $AA' \twoheadrightarrow C$ or $\bowtie [\{AA'C, AA'B\}]$.

One may think of this situation as an INTERESTS universal relation, where A stands for NAME, B for JOB, C for SPORT and A' for SEASON. The fd asserts that SEASON and NAME functionally determine SPORT (each person does one sport each season). The ejd is an emvd, which can be interpreted as follows: every value of NAME is associated with a set of values of JOB and a set of values of SPORT and these two sets are independent of each other. That is, if tuples abc, ab_1c_1 are in the projection of the universal relation on NAME JOB SPORT then tuple abc_1 is also in this projection. Intuitively, NAME embedded multivalued determines JOB and SPORT, which are independent of each other. The uind asserts that every SPORT is someone's JOB (there are professional athletes in each sport).

It is possible to express these dependencies using sentences of the f.o. predicate calculus, with equality for the fd and without equality for the emvd and uind. We list these sentences. One should note the presence of the $\exists$ quantifiers for the emvd and uind, this is an important difference from the fd.

$$fd : \forall x \forall y_1 \forall z \forall x' \forall y \forall z_1 (R(xy_1zx') \wedge R(xyz_1x')) \Rightarrow (z = z_1)$$

$$emvd : \forall x \forall y \forall z_1 \forall x'_1 \forall y_1 \forall z \forall x'_2 (R(xyz_1x'_1) \wedge R(xy_1zx'_2)) \Rightarrow \exists x'_3 R(xyzx'_3)$$

$$uind : \forall x_1 \forall x_2 \forall x_3 \forall x_4 R(x_1x_2x_3x_4) \Rightarrow \exists y_1 \exists y_2 \exists y_3 R(y_1x_3y_2y_3) \square$$

r	A	B	C	A'
	a	b	c	a'
	a	b'	c	a'
	a	b	c'	a''
	a	b'	c'	a''
	a_1	c	c'	a''
	a_2	c'	c'	a''

summary	x	y	z	x'
T_{fd}	x	y_1	z	x'
	x	y	z_1	x'

summary	x	y	z	
T_{emvd}	x	y	z_1	x'_1
	x	y_1	z	x'_2

Figure 3: The relation and tableaux of Example 2.3.12

Remark 2.3.13: Of course, where one draws the line for dependencies is an educated, but somewhat arbitrary choice. As we shall see in Section 3, fd's and ejd's are typical cases of the larger class of *embedded implicational dependencies* (eid's), which were identified by a number of researchers independently as the natural closure of fd's and ejd's [BeeV5, BeeV6, Fag6, YanP]. Eid's are unirelational and typed, but can be generalized to the multirelational and untyped case; this closure is refered to generically as *embedded dependencies* in [FagV], and this is where we will draw the line in Section 3. Inclusion dependencies (ind's) are among the most studied embedded dependencies that are not eid's; uind's are special ind's that are also not eid's. As sentences of the f.o. predicate calculus with equality, embedded dependencies have quantifier prefix $\forall^*\exists^*$. They are called *full dependencies* if the quantifier prefix is only $\forall^*$. The full eid's are called *full implicational dependencies* (fid's); fd's and jd's are fid's.

Here we focus on ejd's, fd's and uind's in order to stress the existence of PTIME computational properties. For these statements implication can be expressed as an unsatisfiability problem of a f.o. sentence, whose quantifier prefix is $\exists^*\forall^*\exists^*$. (The argument is similar as in the fd case, only the last existential quantifiers must be added). For fd's and jd's this sentence has quantifier prefix $\exists^*\forall^*$. Thus finite implication and implication coincide and are decidable for fd's and jd's; this need not be the situation in the presence of either ejd's or uind's.

We use N for the input size to a dependency implication problem. Let us first examine testing a decomposition for losslessness, given a set of fd's. It is easy to show that: full decomposition π_D is lossless iff the jd $\bowtie [D]$ is implied by the given set of fd's. Also, here finite implication and implication coincide. The first algorithm for this problem used $O(N^4)$ time and was described in [AhoBU]. It was improved in [LiuD] and [DowST]; the best current bound from [DowST] is based on the operation of congruence closure and is $O(N^2 \log^2 N/\log k)$ time and $O(N^2 k)$ space for k a parameter with $1 \leq k \leq N$. The proof also applies to an embedded decomposition, i.e., an ejd as the implied statement. The results were extended in [KanCV] to finite implication (in $O(N^3)$ time) and implication (same bounds as [DowST]) of a ejd by a set of fd's and uind's. As shown in [CasFP] finite implication and implication differ in the presence of fd's and uind's.

Theorem 2.3.14: *Finite implication and implication of an ejd by a set of fd's and uind's do not coincide, but are both in* PTIME.

In this theorem, it is important that the domains of attributes be unbounded, since two-tuple relations no longer characterize the implication problems. If some domain has only two values then implication of a jd by a set of fd's is NP-complete, [Kan1].

An important technique grew out of the algorithm of [AhoBU] for testing lossless joins. This algorithm was extended in [MaiMS] to antecedent statements being fd's and jd's and named the *chase*. The chase can be further extended into a semi-decision procedure for embedded dependency implication and an exponential decision procedure for full dependency implication, see [BeeV5, BeeV6] for the general setting. In its most general form it is similar to resolution with paramodulation and it uses a correspondence between tableaux and embedded dependencies. For the ind chase we refer to [JohK]. But we should note that equational forms of reasoning seem more natural for ind problems [CosK1, CosK2, Cos2, Mit]. Let us describe the chase for jd's and fd's from [MaiMS].

In [MaiMS] it is also shown that the *chase* procedure below is Church-Rosser, i.e., its result is unaffected by nondeterministic choices in its steps. Note that, procedure *chase*(Σ, σ) is a decision procedure, since it always terminates. From its description one can see that no new symbols are generated during the computation and this fact accounts for the termination. It runs in exponential time.

```
procedure chase($\Sigma$,$\sigma$)
input $\Sigma$ a set of fd's and jd's, $\sigma$ a fd or jd ;
    $T := T_\sigma$, where $T_\sigma$ is the tableau associated with $\sigma$ ; success := false ;
    repeat
        if $\sigma'$ is a fd in $\Sigma$ such that relation $T \not\models \sigma'$, because $t[A] \neq t'[A]$
          then replace all occurrences of one of $t[A], t'[A]$ by the other
          keep the distinguished symbol if one is such or else
          keep the lowest subscript nondistinguished symbol
        if $\sigma'$ is a jd in $\Sigma$ such that relation $T \not\models \sigma'$ because $T \neq \bowtie \pi_D(T)$
          then replace relation $T$ by relation $\bowtie \pi_D(T)$
    until no further modifications of $T$ are possible;
    if $\sigma$ is fd $X \rightarrow Y$
        then success := true if all nondist. vars for $Y$ in $T_\sigma$ became dist. in $T$;
    if $\sigma$ is jd $\bowtie [D]$
        then success := true if the summary of $T_\sigma$ is a tuple of $T$;
output if success = true then $\Sigma \models_{(f)} \sigma$ else $\Sigma \not\models_{(f)} \sigma$ ;
end
```

Using properties of the chase it is possible to show a number of positive results. These we summarize in Theorem 2.3.15, which complements Theorem 2.3.14. The simultaneous presence of fd's and uind's makes finite implication and implication different, but they are still both decidable and sometimes efficiently so. We state the theorem using the more general embedded implicational dependencies (instead of ejd's and fd's) and full implicational dependencies (instead of jd's and fd's); see Remark 2.3.13. The results for ejd's, jd's and fd's only are from [MaiMS, MaiSY, Var6], for eid's and fid's only are from [Var1], and the addition of uind's is from [KanCV].

Theorem 2.3.15: (1) *Implication and finite implication of an eid or uind from a set of fid's and uind's do not coincide, but are both in* EXPTIME. (2) *Implication and finite implication of a mvd, fd or uind from a set of fid's and uind's do not coincide, but are both in* PTIME.

Unfortunately, the precise complexity of the ejd implication problems is still open. The only known lower bounds are NP-hardness ones, [BeeV1, BeeV2, BeeV4, MaiSY]. For a discussion of the open questions see [FagV]. For example, it was shown in [FisT] that: *testing whether a set of mvd's implies a jd is* NP*-hard.*

One way of understanding implication is through the development of complete and sound formal systems for it, e.g., the system FD for fd's. Such formal systems are often nontrivial to construct, particularly in the presence of jd's, see [BeeV2, BeeV4, Sci2]. Also, unlike for fd's, no complete and sound k-ary system (for some fixed k) might exist. For example, it was shown in [CasFP] that this is the case for many fd and ind implication problems, including fd and uind finite implication. This is also a difficulty for emvd implication, see [ParP, SagW].

In the restricted world of mvd's, fd's and uind's (i.e., the only ejd's allowed are mvd's) the situation is almost ideal. All implication problems are efficiently solvable and have sound and complete formal systems that are simple.

The first sound and complete formal system for fd's and mvd's appeared in [BeeFH]; here we give a slightly modified version MFD from [Var8]. Mvd and fd implication was solved in $O(N^4)$ in [Bee] and these bounds were improved in [Gal, HagITK, Sag1]; the best current bound is $O(N \log N)$ from [Gal]. This computational behavior is due in large part to the algebraic properties of what is called in [BeeFH] a *dependency basis.* The results were extended in [KanCV] to also include uind's, with an overhead of $O(N^3)$ for finite implication and of $O(N \log N)$ for implication.

We summarize the known results in Figure 4. Namely, the complexities of the implication problems together with sound and complete formal systems for these problems. We have already seen system FD = { FDr, FDa, FDt }. To these rules we add a number of rules and use the following convention in Figure 4: MD = { MDc, MDa, MDd }, MFD = FD ∪ MD ∪ { MFDt, MFDi }, etc. We also note when finite and unrestricted notions are equal. All the rules that follow are 2-ary ones. The only exception is the last set of "cycle rules", where we have one cycle rule for each odd positive integer k.

MDc complement axiom scheme: $\vdash X \twoheadrightarrow U - X$

MDa augmentation inference rule: $X \twoheadrightarrow Y \vdash XZ \twoheadrightarrow YZ$

MDd difference inference rule, where $Y \cap Z = \emptyset$: $X \twoheadrightarrow Y$ *and* $Z \twoheadrightarrow Y_1 \vdash X \twoheadrightarrow Y - Y_1$

MFDt translation inference rule: $X \rightarrow Y \vdash X \twoheadrightarrow Y$

MFDi intersection inference rule, where $Y \cap Z = \emptyset$: $X \twoheadrightarrow Y$ *and* $Z \rightarrow Y_1 \vdash X \rightarrow Y \cap Y_1$

UDr reflexivity axiom scheme: $\vdash A \subseteq A$

UDt transitivity inference rule: $A \subseteq B$ *and* $B \subseteq C \vdash A \subseteq C$

UFDe $\emptyset$ interaction inference rule: $\emptyset \rightarrow A$ *and* $B \subseteq A \vdash A \subseteq B$ *and* $\emptyset \rightarrow B$

CD one cycle inference rule each for odd positive integer k:

$$A_0 \rightarrow A_1 \text{ and } A_2 \subseteq A_1, \ldots, A_{k-1} \rightarrow A_k \text{ and } A_0 \subseteq A_k \;\; \vdash$$
$$A_1 \rightarrow A_0 \text{ and } A_1 \subseteq A_2, \ldots, A_k \rightarrow A_{k-1} \text{ and } A_k \subseteq A_0$$

The efficient solution for mvd, fd and uind problems provides a convenient pause in our excursion into dependency theory. The next logical step is the investigation of emvd's, which motivated many of the generalizations examined in Section 3. The implication problems for emvd's are the major open questions in dependency theory. No upper or lower bounds are known for these problems beyond the trivial ones (even the NP-hardness bounds for ejd's do not apply to emvd's).

Open Problem 2.3.16: Are emvd implication and finite implication decidable?

Σ, σ	$\models$	$\models_f$	$\vdash$	$\vdash_f$
fd	$O(N)$	$=\models$	FD	$=\vdash$
mvd	$O(N \log N)$	$=\models$	MD	$=\vdash$
uind	$O(N)$	$=\models$	UD	$=\vdash$
mvd,fd	$O(N \log N)$	$=\models$	MFD	$=\vdash$
mvd,uind	$O(N \log N)$	$=\models$	MUD	$=\vdash$
fd,uind	$O(N)$	$O(N^3)$	UFD	CUFD
mvd,fd,uind	$O(N \log N)$	$O(N^3)$	UMFD	CUMFD

Figure 4: Mvd, fd, uind implication problems

2.4 On Hypergraphs and the Syntax of Database Schemes

With a database scheme D or a jd $\bowtie [D]$ or a decomposition π_D we can associate a *hypergraph*. Hypergraphs, like undirected graphs, consist of a finite set of *nodes* (in this case the attributes in D) and a finite set of sets of nodes or *edges* (in this case the relation schemes of D); where we assume that each node belongs to some edge. The many applications of hypergraph theory to database theory illustrate the use of attribute notation.

Most graph notions, such as paths or connectivity, immediately carry over to hypergraphs. Unlike graphs, however, hypergraphs have a number of inequivalent notions of "acyclicity". The *acyclicity* we examine here is the first one proposed in the context of database theory [BeeEtal]. It is also known as α-acyclicity, to distinguish it from other variants examined in [Fag7]. There is a large number of combinatorial characterizations of acyclicity [BeeFMY, FagMU], many of which have applications in database theory. Here we use the operational definition based on the GYO reduction, independently identified in [Gra1] and [YuO].

Definition 2.4.1: Database scheme D, jd $\bowtie [D]$, decomposition π_D and hypergraph D are *acyclic* if hypergraph D can be reduced to the empty set by some sequence of applications of the following two rules: (1) if an edge is a subset of another edge then delete it, and (2) if a node belongs to precisely one edge then delete it.

Testing Definition 2.4.1 is clearly in PTIME. As shown in [TarY], this can be done optimally in linear-time. An example of an acyclic hypergraph is given pictorially in Figure 5. Note that a subset of its edges form a cyclic hypergraph; this behavior is somewhat counterintuitive and does not occur for γ-acyclicity [Fag7]. Despite such anomalies, acyclic database schemes are well motivated from a semantic point of view, e.g., see [Lie, KifB].

An attractive feature of acyclicity, from a computational point of view, is that a number of NP-hard questions involving jd's or decompositions can be resolved in PTIME, provided the input is acyclic. Following the exposition of [FagV], we present three such questions. The first is about jd's, the second about decompositions, and the final one about database schemes and the pure universal relation assumption.

A jd is acyclic iff it is equivalent to (i.e., it implies and is implied by) some set of mvd's. This is one of the characterizations of acyclicity from [FagMU]. From [BeeFMY] it follows that, given an acyclic jd then an equivalent set of mvd's can be found in PTIME. The converse, given a set of mvd's finding an equivalent jd or reporting that none exists, was shown in [GooT] also to be in PTIME. There is an interesting consequence. Recall that testing implication of a jd by a set of mvd's is NP-hard [FisT], but testing implication of an acyclic jd by a set of mvd's is in PTIME. To

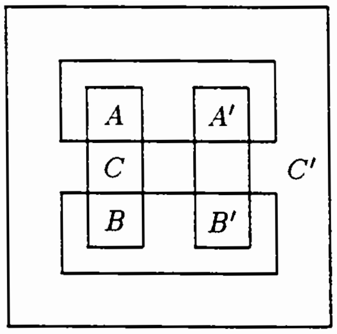

Figure 5: An acyclic hypergraph

see this use the [FagMU] characterization, the [BeeFMY] PTIME construction and mvd implication. In summary we have the following.

Theorem 2.4.2: (1) *A jd is acyclic iff there exists a set of mvd's equivalent to it.*
(2) *Given an acyclic jd, an equivalent set of mvd's can be constructed in* PTIME.
(3) *Given a set of mvd's, finding an equivalent jd or that none exists is in* PTIME.
(4) *Implication of an acyclic jd from a set of mvd's is in* PTIME.

As shown in [Var4] about decompositions, injectiveness (See Section 2.3.III) does not always entail that the reconstruction function is the join. This is the case when the dependencies are jd's and fd's and the decomposition is in two relation schemes [BeeV7, CosP]. In fact, the results of [BeeV7] are for the more general case of fid's (see Remark 2.3.13) and acyclic decompositions. In [KanCV] it is shown that uind's can be added without changing the theorem.

Theorem 2.4.3: *Let the possible universal relations $< \{U\}, \Sigma >$ be determined by a set Σ of fid's and uind's over U, and let the decomposition π_D be acyclic, then:* (1) *if π_D is injective the reconstruction function is the join, and* (2) *testing if π_D is injective is equivalent to testing $\Sigma \models_f \bowtie [D]$ and is in* PTIME.

We say that a database $d = \{r_1, \ldots, r_m\}$ over D is *join consistent* or is the *projection of a universal relation* if there is a relation r, such that $\pi_D(r) = d = \{r_1, \ldots, r_m\}$. In other words, the pure universal relation assumption states that all databases over D are join consistent. This is only a simplifying assumption. Not all databases are join consistent, as one can see from simple examples. Another problem is that, given a d over D testing whether it is join consistent is NP-complete, see [HonLY].

A less restrictive assumption about a database $d = \{r_1, \ldots, r_m\}$ over D is that it is *pairwise consistent.* Namely, every two relations r_i, r_j over R_i, R_j respectively agree on their common attributes $R_i \cap R_j$, i.e., $\pi_{R_i \cap R_j}(r_i) = \pi_{R_i \cap R_j}(r_j)$, for $1 \leq i, j \leq m$,. An equivalent way of defining pairwise consistency is requiring every database $\{r_i, r_j\}, 1 \leq i, j \leq m$, to be join consistent. Every join consistent database is pairwise consistent, but not conversely. Pairwise consistency can be expressed using inclusion dependencies. It is easy to see that pairwise consistency can be tested in PTIME. From [BeeFMY] we have the following characteriztion of acyclicity.

Theorem 2.4.4: (1) *If database scheme D is acyclic, then any database over D that is pairwise consistent is also join consistent.* (2) *If database scheme D is not acyclic, then there is a pairwise consistent database over D which is not join consistent.* (3) *Given database d over acyclic D, testing whether d is join consistent is in* PTIME.

Additional Bibliographic Comments 2.4.5: [Yan1] contains a large number of problems solvable in PTIME in the presence of acyclicity. Acyclicity is a very useful tool for query optimization. This has led to an investigation of query evaluation based on the *semi-join* operation, e.g., [BerC, BerG, GooS1, GooS2, GooS3, SagS2]. Acyclicity in the presence of fd's is the topic of many studies, e.g., [LavMG, Sac, SacMM, SagS1]. Various forms of acyclicity defined in [Fag7] were further examined in [AusDM, DatM, GrhR]. Finally, acyclicity has been used in database scheme design and in the context of universal relation data models. □

2.5 On Logic and the Semantics of Databases

We have, until now, used one basic paradigm: that the database is a finite f.o. relational structure. This *model theoretic* approach is not the only one possible. In fact, a *proof theoretic* approach might be more appropriate for issues beyond the capabilities of the relational data model, such as the problems of incomplete information and updates that we address in Section 5.

Let us reexamine our basic premise, that the database *is* the semantics. Instead, we can view databases as *f.o. theories* of a very special form, called *extended relational theories* by Reiter in [Rei3]. A f.o. theory is a set of sentences of the f.o. predicate calculus with equality, over some vocabulary of constant and relation symbols. For the relational data model this point of view is equivalent to our original premise. An advantage of databases as structures is the natural development of the rich repertoire of algebraic techniques that we have already seen. On the other hand, a fresh point of view may lead to other important data models.

We discuss two such types of data models: deductive and universal relation data models. Our exposition highlights the fact that: at a fundamental level these data models share many ideas, even if they have been developed separately.

The study of data models based on f.o. theories is known as the area of *deductive databases*. Research in this area was greatly influenced by an early collection of papers on the subject [GalM] and many of the original advances are surveyed in [GalMN]. Recently, there has been much work on databases and logic programming, that is related to deductive databases, see [Min2]. There is an intimate connection between deductive databases and the study of query languages beyond relational algebra, in particular, database logic programs. This has been demonstrated in [ChaH3] and is emphasized in [Min2]. We therefore chose to examine these subjects together, in some detail, in Section 4.

In this section we set up the f.o. theory framework for the relational data model. Modifications of this framework lead to nontrivial extensions, both from the point of view of query language expressive power and in terms of representation power for incompletely specified data. We also focus on the *weak universal relation assumption* and its relationship to deductive databases. Given the expressive power of the f.o. theory point of view, the historical choice of the terminology "weak" is rather unfortunate (its justification is that as an assumption it is weaker than the pure universal relation assumption).

2.5.I Deductive Data Models and First-Order Theories

Consider a vocabulary consisting of relation symbols $\{R_1, \ldots, R_m\}$ and of a countably infinite set of constant symbols $\mathcal{C}$. This set $\mathcal{C}$ and the set of values Δ we put into one-to-one and onto correspondence, i.e., a in $\mathcal{C}$ corresponds to a in Δ. A *finite f.o. theory* θ is a finite set of sentences of the f.o. predicate calculus with equality over this vocabulary.

Consider a standard proof system for the f.o. predicate calculus with equality [End]. The consistency of a theory is the proof theoretic property, that not all sentences are provable using the theory and the standard proof system. It is well known that consistency is the proof theoretic analog of satisfiability: a finite f.o. theory is *consistent* iff it is satisfiable [End]. So, let F be an expression of the relational calculus as in Definition 2.2.1: $\{R, x_1, \ldots, x_n : \varphi(x_1, \ldots, x_n)\}$. Then the new framework results from defining $F(\theta)$ as follows:

$$F(\theta) = \begin{cases} \{a_1 \ldots a_n : a_i \text{ appears in } \theta, 1 \le i \le n, \theta \models_f \varphi(a_1, \ldots, a_n)\} & \text{if } \theta \text{ consistent} \\ \emptyset & \text{if } \theta \text{ inconsistent} \end{cases}$$

Note that $\models_f$ is finite implication of sentences, and not satisfaction by a finite model as in Definition 2.2.1. This approach is best exemplified by the work of Reiter, e.g., [Rei1, Rei2, Rei3]. It has been called *proof theoretic*, because, under certain sufficient conditions for the sentences, finite implication $\models_f$ may be replaced by *provability* $\vdash_f$.

Thus, the tuples of $F(\theta)$ correspond to all the variable assignments verifying φ (or the facts) that we can prove from θ in first-order logic. This conservative approach towards what is true, i.e., *facts are true only if they can be proven*, is known as *the closed world assumption* [Rei1]. Now let us try to relate this new approach to the relational data model.

If d is a database we will construct a finite f.o. theory θ_d called an *extended relational theory*. This θ_d is constructed from d using the union of four sets of axioms AT, UN, DO, CO. We use shorthand $\vec{x}$ for $x_1, \ldots, x_n$ and $\vec{a}$ for $a_1, \ldots, a_n$, where n is determined from the context and equality $\vec{x} = \vec{a}$ is componentwise.

AT A sentence $R(\vec{a})$ for each R-tuple $\vec{a}$ in d.

UN A sentence $a \neq a'$ for each pair of distinct values a, a' occurring in d.

DO A sentence $\forall x(x = a_1 \vee \ldots \vee x = a_k)$ if the values occurring in d are exactly $\{a_1, \ldots, a_k\}$.

CO A sentence $\forall \vec{x}(R(\vec{x}) \Rightarrow (\vec{x} = \vec{a}_1 \vee \ldots \vee \vec{x} = \vec{a}_j)$, for each symbol R, where the R-tuples in d are exactly $\{\vec{a}_1, \ldots, \vec{a}_j\}$. If $j = 0$ this sentence is $\forall \vec{x}(\neg R(\vec{x}))$.

The axioms AT (atomic facts) just express the tuples as ground atoms; the axioms UN (uniqueness) just express that different symbols are different objects; and the axioms DO (domain closure) and CO (completion) guarantee that θ_d defines a unique model d, modulo automorphisms on $\mathcal{C}$.

If d is a database and Σ a set of dependencies then $F(\theta_d \cup \Sigma)$ has the following two properties. (Definition 2.2.1 is used for $F(d)$ and ER stands for extended relational theory).

ER1 $F(\theta_d \cup \Sigma) = F(\theta_d) = F(d)$ if d satisfies Σ

ER2 $F(\theta_d \cup \Sigma) = \emptyset$ if d does not satisfy Σ

Note, how in these properties the problem of testing if d satisfies Σ is separated from computing $F(d)$. The facts that can be proven from $\theta_d \cup \Sigma$ are the same as the result of querying the database according to Definition 2.2.1.

Remark 2.5.1: If we assume that $\Sigma = \emptyset$ and only a subset of the UN axioms are present, then we have a situation of incomplete information. There is a bound on the database domain and the tuples of the relations. However, not all constant symbols need represent distinct values. The uniqueness of the model of θ_d is lost, i.e., there might be many "possible worlds". In this case, the definition of $F(\theta_d)$ above has been used to provide semantics for querying incomplete information databases, see [Var9, Rei3].

One way to model incomplete information is to omit some of the axioms of an extended relational theory. In Remark 2.5.1 we deleted some of the UN axioms. An interesting situation arises when we have dependencies and we delete some of the CO axioms.

Let us assume that the vocabulary of relation symbols is $\{R_1, \ldots, R_m\}$. We are given an extended relational theory θ_d^c, *without the* CO *axioms for some of the relation symbols*, and a set of dependencies Σ over this vocabulary. Sentences $\theta_d^c \cup \Sigma$ form a *deductive database.* The theory θ_d^c represents the known facts about the application; Σ can be used to derive other true facts as follows:

Definition 2.5.2: If d is a database and $\theta = \theta_d^c \cup \Sigma$ is a deductive database and R is a relation symbol in its vocabulary then R denotes the query,

$$R[d, \Sigma] = \begin{cases} \{a_1 \ldots a_n : a_i \text{ appears in } \theta, 1 \leq i \leq n, \theta \models_f R(a_1 \ldots a_n)\} & \text{if } \theta \text{ consistent} \\ \emptyset & \text{if } \theta \text{ inconsistent} \end{cases}$$

Note that this definition gives us a data model. Relations (extended relational theories) are manipulated via a query language implicitly defined by $\models_f$. This language makes little distinction between dependency satisfaction and finite implication. For example, a calculus query $F(d)$ can be expressed using a deductive data model by incorporating F's definition into Σ, provided that the set of dependencies allowed is sufficiently expressive.

Example 2.5.3: Let R be a binary relation symbol and θ_r^c be an extended relational theory over R without the CO axiom for R, where r is a directed graph without isolated nodes. *Trans*(r) is the transitive closure of r, which is not expressible in relational calculus, see Theorem 2.2.8. It is not hard to see that $R[r, \{\sigma\}] = \mathit{Trans}(r)$, where,

$$\sigma = \forall x \forall y \forall z (R(xz) \wedge R(zy)) \Rightarrow R(xy). \square$$

Deductive data models are parameterized by the possible syntax of the dependencies of Σ. In Example 2.5.3 the dependencies belong to the language of *universally closed Horn clauses.* The query language, implicitly defined by $\models_f$ for such Horn clauses, is known as *Datalog* [ChaH3, MaiW] and we will examine it in Section 4 in some detail. We will also find universally closed Horn clauses as natural dependencies, called *rule dependencies*, that generalize jd's in Section 3. This Horn form of Σ is sufficient to guarantee that, the semantics based on Definition 2.5.2 are the same as the minimal Herbrand model semantics for the deductive database seen as a logic program, see [Apt, AptV, Min2].

Syntactic classes for Σ can be used to define query languages that are more expressive than relational calculus; see Example 2.5.3 and Theorem 2.2.8. We believe, however, that the study of these formalisms is best conducted as a study of query languages. This allows for a clean separation between issues of querying and of checking integrity of the database. We thus refer the reader to Section 4, where we study expressibility in various logics as opposed to the properties of deduction.

2.5.II Universal Relation Data Models and First-Order Theories

One of the original motivations for introducing the relational data model was to free the programmer from the need to procedurally specify how the data should be accessed; the so-called "navigation problem". When programming at the level of abstraction of relational algebra, the programmer need not worry about the access paths within the data structures that implement relations; the so-called "physical navigation problem". Nevertheless, "logical navigation" is still necessary. Even in the declarative relational calculus one has to specify how to connect various relations (but with less effort than in the algebra). Universal relation data models emerged through the work of many researchers, whose goal was to further simplify "logical navigation".

The ideal goal is to use the attribute notation and the hypergraph structure of database schemes, as a query language that is more declarative than relational calculus. Instead of using a relational calculus expression F (from databases over D to databases over $\{\alpha(F)\}$) just use $\alpha(F)$. The hope is that the database scheme structure and the dependencies have enough information for the system to automatically choose some F by using $\alpha(F)$. This of course involves thoughtful database scheme design and might work only for some expressions F.

We have already seen *the pure universal relation assumption*, first proposed as a simplifying assumption about database scheme design. The set of possible worlds are specified as $< \{U\}, \Sigma >$ and only databases that are decompositions of such *universal* relations are considered. A potential problem is that many databases are not decompositions of any universal relation. Even testing whether there exists a universal relation r such that the given d is a decomposition of r is NP-complete, [HonLY].

These difficulties have led to the formulation of an alternative assumption, the *weak universal relation assumption.* This assumption, instead of the existence of a universal assumption, postulates the existence of a *weak universal relation.* A very good exposition of the subject can be found in [MaiUV]. In our definitions we limit Σ to the full dependencies of [FagV] (see also Section 3). For the embedded ones we refer to [MaiUV].

Definition 2.5.4: Let $d = \{r_1, \ldots, r_m\}$ be a database over $D = \{R_1, \ldots, R_m\}$ with universe U and Σ be a set of full dependencies. A relation r over U is a *weak universal relation* for d w.r.t. Σ if: (1) r satisfies Σ, and (2) $r_i \subseteq \pi_{R_i}(r), 1 \leq i \leq m$. Each subset X of U denotes the query:

$$X[d, \Sigma] = \bigcap \{\pi_X(r) : r \text{ is a weak universal relation for } d \text{ w.r.t. } \Sigma\}$$

Note that this definition gives us data models, which are parametrized by the class of dependencies of which Σ is a subset, e.g., fd's or jd's together with fd's etc. Relations are manipulated and a query language is provided: queries are specified implicitly from the attribute set X, the dependencies Σ and the intersection condition.

Let us determine the precise relationship of universal relation data models with deductive data models. The vocabulary we use has one relation symbol X for each subset X of U and the constant symbols C from the previous subsection. We build a finite f.o. theory θ_d^u, as the union of four sets of sentences AT, UN, IN, CN. The first two sets AT, UN are the same as for extended relational theory θ_d (see Section 5.2.I). The sentences IN express that for each X the relation over X is a subset of the projection on X of the weak universal relation. The sentences CN express the converse, that for each X the projection on X of the weak universal relation is a subset of the relation on X. Let us illustrate these conditions by example, where $U = ABC$ and $X = AB$.

IN For $U = ABC, X = AB$ include the sentence $\forall x \forall y \exists z X(xy) \Rightarrow U(xyz)$.

CN For $U = ABC, X = AB$ include the sentence $\forall x \forall y \forall z U(xyz) \Rightarrow X(xy)$.

An important observation from [MaiUV] is that, if we consider theory θ_d^u instead of θ_d^c in Definition 2.5.2 then we get the $X[d, \Sigma]$ of Definition 2.5.4. Namely, if $\theta = \theta_d^u \cup \Sigma$ then:

$$X[d,\Sigma] = \begin{cases} \{a_1 \ldots a_n : a_i \text{ appears in } \theta, 1 \leq i \leq n, \theta \models_f X(a_1 \ldots a_n)\} & \text{if } \theta \text{ consistent} \\ \emptyset & \text{if } \theta \text{ inconsistent} \end{cases}$$

One difference from deductive databases is the availability of a rather large vocabulary of relation symbols X, namely all the subsets of U. This is to facilitate "logical navigation". Another difference is the special emphasis in universal data models on Σ's consisting of fd's. With fd's query evaluation has a distinctly different flavor than other deductive database query evaluation techniques, e.g., the ones for Σ consisting of rule dependencies.

The two obvious questions arise from Definition 2.5.4. (1) How does one test *consistency*, in this case the existence of some weak universal relation of d w.r.t. Σ. (2) How does one *evaluate* $X[d, \Sigma]$. Since the two characterizations we have are not procedural, there could be an infinite number of candidate weak universal relations of which we need to compute the intersection.

Fortunately, there is a naive but procedural method to answer these questions. The equivalence of the following method with Definition 2.5.4 is shown in [MaiUV, Sag2]:

(i) Construct a *representative universal relation* for database d; this consists of the tuples of d padded with uniquely occurring special symbols called *nulls* that turn these tuples into U-tuples.

(ii) This representative universal relation is viewed as a tableau and is chased by the dependencies in Σ, using a generalization of the chase from Section 2.3.III. Since Σ consists of full dependencies this procedure terminates. If two non-nulls are equated then d, Σ are inconsistent else they are consistent.

(iii) If d, Σ are consistent then $X[d, \Sigma]$ is the projection on X of the result of the chase, restricted to tuples without nulls.

A consequence is that query evaluation becomes dependency implication, where the representative universal relation is viewed as the dependency inferred. The following theorem is from [Hon] for fd's and from [GraMV] for other dependencies. For embedded dependencies, the chase may not terminate and the various questions become undecidable.

Theorem 2.5.5: *Under the weak universal assumption, testing consistency of d and Σ and evaluating $X[d, \Sigma]$, when Σ is a set of fd's, are both in* PTIME. *They are* EXPTIME*-complete, when Σ is a set of full dependencies.*

Constructing and chasing the representative universal relation is not particularly efficient. There has been a great deal of work on better procedural query evaluation methods, e.g., [Gra2, MaiRW, MaiUV, Men, Sag2, Sag3, Sag4, Sag5, Yan1, Yan2]. The goal of these efforts is, given Σ, X and D to find a relational algebra expression E over D such that: for all databases d over D we have

$E(d) = X[d, \Sigma]$. This would reduce the query in the universal data model to a relational algebra query. Such an E need not always exist. If an E exists we call Σ, X, D *algebraic*.

It is easy to see that, the transitive closure query of Example 2.5.3 can be computed in a universal relation data model, one just needs $U = AB$ and a single full dependency; so this case is not algebraic. There are algebraic cases where E exists and can be computed efficiently, for example if Σ is one jd [Sag2, Yan1]. For this case, with X extended to a positive existential query instead of just a projection, see [Yan2].

The algebraic cases (where the difference relational operation is excluded) are characterized in [MaiUV] by a property of the chase procedure, i.e., termination in a fixed number of steps for any d. This property is called *uniform boundedness* of the chase. There are instances where uniform boundedness can be decided: if Σ is a set of fid's without equality and $X = U$, see [Sag5].

Let us close this section with yet another application of dependency implication. We would like to compare the pure and the weak universal relation assumptions. One desirable condition is that: if the pure assumption holds then we get the same result by projecting from the universal relation as we do by following the semantics of Definition 2.5.4. Formally we want that: for all r satisfying Σ and such that $d = \pi_D(r)$ we have $\pi_X(r) = X[d, \Sigma]$. In [MaiUV] this is shown to be true iff $\Sigma \models_f \sigma$, where σ is a *projected-join dependency* (pjd) and relation r satisfies σ if $\pi_X(\bowtie \pi_D(r)) = \pi_X(r)$.

Additional Bibliographic Comments 2.5.6: Weak universal relations have also been used to model incomplete information [Vas]. The weak universal relation assumption can be modified to capture the fine distinctions between values present but unknown, and values whose presence is even unknown [AtzB, CosKS]. Semantics based on set partitions are proposed for the weak universal relation assumption in [CosKS], this shows the close connection between universal relation models and the deductive data models of [Spy]. □

3 Dependencies and Database Scheme Design

3.1 Dependency Classification

In this section we try to classify the many dependencies that have been examined in the literature. We follow the classification framework of embedded dependencies from [FagV]. We highlight the importance of three subclasses of these dependencies: embedded implicational dependencies [BeeV5, BeeV6, Fag6, YanP], inclusion dependencies [CasFP], and rule dependencies [ChaH3, GallMN]. We outline some of the decidability and complexity bounds on their implication problems.

Definition 3.1.1: An *atom* is a formula of the form $R(z_1, \ldots, z_j)$, a *relational atom*, or is a formula of the form $z_1 = z_2$, an *equality atom*; where $z_1, \ldots, z_j$ are not necessarily distinct variables and R is a relation symbol of arity $j \geq 0$. Let $\varphi(x_1, \ldots, x_n)$ and $\psi(y_1, \ldots, y_m)$ be two nonempty conjunctions of atoms, such that $x_1, \ldots, x_n, n \geq 1$ and $y_1, \ldots, y_m, m \geq 1$ are the distinct variables appearing in these conjunctions respectively. An *embedded dependency* is a sentence of the f.o. predicate calculus with equality of the following form:

$$\forall x_1 \ldots \forall x_n \varphi(x_1, \ldots, x_n) \Rightarrow \exists z_1 \ldots \exists z_k \psi(y_1, \ldots, y_m)$$

where $\{z_1 \ldots z_k\} = \{y_1 \ldots y_m\} - \{x_1 \ldots x_n\}$.

A set of embedded dependencies Σ is satisfied by a database d, if d satisfies each sentence in Σ, where the relation r_1 of d interprets R_1 etc. Note that embedded dependencies are satisfied by a database with empty relations and that each embedded dependency is a domain independent sentence. Also a set of dependencies Σ can be written as a sentence in prenex normal form with quantifier prefix $\forall^*\exists^*$. In the definition above conjunction φ is called the *body* and conjunction ψ the *head* of an embedded dependency. Without loss of generality, "the equality symbol need only occur in the head ψ and only between variables that also appear in the body φ"; we will hence assume that this well-formedness condition about equality is true. There are five common restrictions on embedded dependencies that give us five classes of dependencies.

1. The *full* are those without $\exists$ quantifiers.
2. The *unirelational* are those with one relation symbol only.
3. The *1-head* are those with a single atom in the head.
4. The *tuple-generating* are those without the equality symbol.
5. The *equality-generating* are full, 1-head, with an equality atom as head.

For full dependencies, being 1-head is not a constraining assumption, since each full dependency can be replaced by an equivalent set of full 1-head dependencies. One can always replace an embedded dependency by a set of tuple-generating dependencies and equality-generating dependencies, see [Var1]. Within this five-fold classification three special subclasses of dependencies have been identified for their practical significance.

Unirelational Typing and Eid's: Let R be the relational symbol for the unirelational case. Partition the sets of variables into sets corresponding to the arguments of R. A unirelational dependency is *typed* if (1) its equality atoms are between two variables both from a set corresponding to some argument of R, and (2) its relational atoms have in each argument of R a variable from the set corresponding to that argument. The embedded dependencies that are typed unirelational are called *embedded implicational dependencies* (eid's). They were identified, independently and in a variety of formalisms, as the natural closure of many sets of dependencies, [BeeV5, BeeV6, Fag6, YanP]. The full, 1-head eid's are called *full implicational dependencies* (fid's). In Section 2.3.III we have already encountered *functional, join, multivalued dependencies* (fd's, jd's, mvd's) all examples of fid's. Other examples of 1-head eid's that we have encountered in Section 2.3.III are *embedded join, embedded multivalued dependencies* (ejd's, emvd's). All these examples, with the exception of fd's, are tuple-generating. The tuple-generating, 1-head eid's are called *embedded template dependencies* (etd's), [SadU]. Each etd can be described by a tableau; its tuples describing the body and its summary describing the single head atom, where the existentially quantified variables of the head are those distinguished variables missing from the summary. Note that, for dependency satisfaction we are not only interested in the tableau mapping, but in the database being closed under this mapping. When the summary of the tableau has all the distinguished variables then we have *full template dependencies* (ftd's); we have encountered such fid's without $=$ in Section 2.5.II. At the end of Section 2.5.II we also saw a subclass of etd's containing the ejd's, these are the *projected-join dependecies* (pjd's) of [MaiUV]. Finally, the fd's with $|X| = |Y| = 1$ are called *unary* fd's. □

Multirelational Constraints and Ind's: *Inclusion dependencies* (ind's) are all embedded dependencies, such that: the head is one relational atom with no multiple occurrences of variables and the body is one relational atom with no multiple occurrences of variables. For example, $\forall x \forall y \forall z R_1(xyz) \Rightarrow \exists z' R_2(yxz')$ is an ind. Ind's, defined in [CasFP], are very useful in describing multirelational constraint's, e.g., "referential integrity constraints". Note that ind's are tuple-generating, 1-head, but may involve more than one relation symbol and may be untyped. We have already encountered *unary inclusion dependencies* (uind's) for a single relation in Section 2.3.III. Uind's may be defined, in the obvious way, for many relation symbols. The results of Section 2.3.III extend to the multirelational case as long as uind's are the only multirelational statements. There is notational simplicity in denoting single relation uind's as $A \subseteq B$. To extend this notation for ind's one must be careful and sequences instead of sets of attributes must be used. If R_1, R_2 are relation schemes, $< X >$ a sequence of k distinct attributes of R_1 and $< Y >$ a sequence of k distinct attributes of R_2, then $R_1 < X > \subseteq R_2 < Y >$ is an ind, and every ind can be represented in this fashion. For the previous example, this notation would give us $R_1 < AB > \subseteq R_2 < BA >$ where $R_1 = ABC$ and $R_2 = ABA'$. If $k = 1$ we have uind's, $k = 2$ *binary* ind's etc. □

Deduction and Rule Dependencies: The full, 1-head, tuple-generating dependencies are called *rule dependencies* or rules for short. We have already mentioned in Sections 2.5.I–II how rules may be used for querying in deductive and universal relation data models. They play an important part in the area of database logic programs [ChaH3, GalMN]. The analysis and optimization of these programs use many techniques from dependency theory, e.g., [CosK3, Sag6]. A rule may be represented by an untyped tagged tableau with a summary that is full and can have repetitions of variables (see Remark 2.2.15). Ftd's are a special case of rules; for example, the results of [Sag5] for ftd's apply to unirelational and typed database logic programs. □

We have already defined the problems of finite implication and implication of dependencies in Definition 2.3.1. For each class of dependencies we have a special case of these implication problems. For example: an instance of eid implication consists of a set of eid's Σ and an eid σ, the question is whether $\Sigma \models \sigma$ and all these eid's are over one relation symbol R (but there is no a-priori bound on the arity of R).

As we described in Section 2.3.III (finite) implication for a class of embedded dependencies can be identified with (finite) unsatisfiability for a class of sentences in prenex normal form with quantifier prefix $\exists^*\forall^*\exists^*$. For full dependencies this quantifier prefix becomes $\exists^*\forall^*$ and, as a consequence, finite implication and implication are the same and decidable. In fact, this decision problem is in EXPTIME by a chase decision procedure for full dependencies. One corollary is that rules can be checked algorithmically for certain redundancies. For embedded dependencies the situation is qualitatively different: finite implication and implication need not coincide.

The typing in eid's entails a large degree of structure, e.g., algebraic properties such as the *faithfulness under direct product* of [Fag6]. Eid implication can also be axiomatized [YanP], [BeeV6]. There is an elegant semi-decision procedure for eid implication that generalizes the fd-jd chase [BeeV5]. Unfortunately, finite implication and implication are different even for etd's and, as shown independently, in [BeeV3, ChaLM] they are undecidable. These undecidability bounds were strengthened to pjd's independently in [GurL, Var7]. The following two theorems summarize our current understanding about eid implication problems. The first is from [ChaLM] and the second from [GurL, Var7].

Theorem 3.1.2: *Finite implication and implication of fid's coincide and are* EXPTIME*-complete.*

Theorem 3.1.3: *Finite implication and implication of eid's do not coincide and are both undecidable even for pjd's.*

The first result about ind's was that, despite the fact that ind's are not full, finite implication and implication coincide. Theorem 3.1.4 is from [CasFP], where ind implication is also axiomatized.

Theorem 3.1.4: *Finite implication and implication of ind's coincide and are* PSPACE*-complete.*

The implication problems for ind's alone and for ind's together with fd's have a qualitatively different flavor from the eid problems. For example, as illustrated in the axiomatizations of [Mit] and [CosK1, CosK2], equational formal systems are better suited for studying ind and fd implication than is the chase. The chase is still useful, but generally hard to reason about. A careful analysis of the chase in [JohK] suffices to show that implication of an eid from a set of ind's is PSPACE-complete. A sufficient condition for the termination of the chase is *ind acyclicity* [Sci3]. Finite implication and implication of fid's and acyclic ind's coincide; but there are exponential lower bounds for acyclic ind's with fd's [CosK1] and even for acyclic ind's alone implication is NP-complete [CosK2].

As we have seen in Section 2.3.III, uind's and fid's interact in an interesting fashion, [KanCV]. Fid and uind finite implication and implication differ, but are both decidable (Theorem 2.3.15). For fd's, mvd's and uind's all decision questions are in PTIME (Figure 4).

Unfortunately, the good properties of ind's alone, or uind's and fid's, or acyclic ind's and fid's do not extend to the general case. The following theorem was shown independently in [ChaV, Mit].

Theorem 3.1.5: *Finite implication and implication do not coincide and are both undecidable even for unary fd's and binary ind's.*

The relevance of undecidability results, and of other complexity lower bounds, depends on whether arbitrary instances of an implication problem correspond to "real world" situations, see [Sci1, Sci3] for studies of this issue. Pairwise consistency (see Section 2.4) is a natural universal relation assumption, that is expressible using ind's across relations. Let the only dependencies within relations be unary fd's. The *ufd-graph* consists of a node for each attribute of each relation scheme and of an arc $< R.A, R.B >$ for each unary fd $A \rightarrow B$ of R. Unary fd's and pairwise consistency interact in a surprisingly nontrivial fashion. The following is from [Cos2, CosK1].

Theorem 3.1.6: *Implication of unary fd's in the presence of pairwise consistency is undecidable. If the ufd-graph is acyclic, finite implication and implication of ufd's in the presence of pairwise consistency coincide and are decidable.*

There are a number of technical implication problems that remain open, particularly for finite implication of eid's with ind's. For example decidability is open for, (1) finite implication of an eid from a set of ind's [JohK], (2) finite implication of uind's, fd's and emvd's [KanCV], or (3) finite implication of ufd's in the presence of pairwise consistency – Theorem 3.1.6 is for implication. However, the remaining outstanding question is emvd (finite) implication (2.3.16). Dependency theory is a well developed subject, which matured in the late 1970's and early 1980's. Much of the current attention is directed towards rule implication problems, because of their relationship to database logic program optimization.

Additional Bibliographic Comments 3.1.7: There are some classes of dependencies that do not fit in the embedded dependency framework, e.g., in [Var4] arbitrary f.o. sentences are used as dependencies, the embedded dependencies are generalized further in [Cos3], the afunctional dependencies of [DebP] are not embedded ones, and the partition dependencies of [CosKS] are not f.o. expressible (see 2.3.10).

The concept of eid's generalized and unified much of the early work on dependency theory, e.g., [GinZ, GrnJ, MenM, Nic, ParJ, ParP, SadU, SagW]. See [FagV] for the history and the relationship between these eid subclasses. Recent results on special ind and fd implication problems can be found in [CasV, Cos2, CosK1, CosK2, JohK, KanCV, LavMG]. □

3.2 Database Scheme Design

The principal issue in database scheme design can be summarized in the following fashion: replace a specification $< \{U\}, \Sigma >$, of universal relations r satisfying dependencies Σ over U, by a *good* specification $< D, \Sigma' >$, of multirelational databases $d = \{r_1, \ldots, r_m\}$ satisfying dependencies Σ' over $D = \{R_1, \ldots, R_m\}$, where $U = \cup_{i=1}^{m} R_i$. The goal is to replace r by d, where $d = \pi_D(r)$ and π_D is a decomposition (see Section 2.3.III). The intended use of this decomposition is twofold, both for querying and updating the database. We use the notational conventions of this paragraph throughtout this section.

Database scheme design is more of an art than a science. However, certain general principles and crisp mathematical questions have emerged as a result of this endeavor. There are a number of standard expositions [KorS, Mai1, Ull1] and an early survey [BeeBG], where we refer the reader for design-rule motivations such as the elimination of *update anomalies.* Instead we focus on two topics: (I) what is a *good* decomposition π_D of $< \{U\}, \Sigma >$, and (II) what are the most common *normal forms* for $< D, \Sigma' >$.

In (I) we assume that Σ consists of a set of full dependencies. Theorems 3.2.1, 3.2.2 and Definition 3.2.3 can be extended to embedded dependencies verbatim, if we also allow unrestricted relations. In (II) we limit ourselves further to fd's and jd's, since these are the dependencies commonly examined in the context of normalization.

3.2.I Independent Schemes

The least one can require of a decomposition is that it be *injective*, i.e., one-to-one. In Section 2.3.III we have already encountered this requirement, called the *representation principle.* It is also useful that the reconstruction operator, whose existence is implied by injectiveness, be the join. Although the representation principle does not imply join reconstruction, this is so in important special cases, e.g., Theorem 2.4.3.

Let us first assume the pure universal relation assumption and the desirability of injectiveness and join reconstruction. Theorem 3.2.1, from [BeeRi, MaiMSU], reduces these requirements to a dependency implication problem. There is an analogous semantic statement if we assume the weak universal relation assumption, Theorem 3.2.2 from [MaiUV] (see also end of Section 2.5.II): let r be a universal relation satisfying Σ and $d = \pi_D(r)$ then we want the query $U[d, \Sigma]$ to give us back r. Both theorems highlight the importance of having the join dependency $\bowtie [D]$ as part of the design requirements.

Theorem 3.2.1: *Let Σ be a set of full dependencies over U. Decomposition π_D is injective over $< \{U\}, \Sigma >$ with join as the reconstruction operation iff $\Sigma \models \bowtie [D]$.*

Theorem 3.2.2: *Let Σ be a set of full dependencies over U and π_D a decomposition. For all universal relations r satisfying Σ we have that $r = U[\pi_D(r), \Sigma]$ iff $\Sigma \models \bowtie [D]$.*

Decomposing a universal relation into parts is certainly useful from a storage minimization point of view. But it may be wasteful from a querying standpoint. This is an instance of a classical time-space tradeoff. Under the pure universal relation assumption we might have to perform the expensive reconstruction of the universal relation in order to answer a query. Under the weak universal relation assumption we might have to do the same for the representative universal relation; much of the research on querying weak universal relations has focused on avoiding representative relation based evaluation.

Let $\{r_1, r_2\}$ be a decomposition of r and $\{r'_1, r'_2\}$ be a decomposition of r', where r, r' are universal relations satisfying Σ. Another desirable property of decompositions is that the components r_1, r_2, r'_1, r'_2 do not depend on each other; in the sense that $\{r_1, r'_2\}$ and $\{r'_1, r_2\}$ are also decompositions of universal relations that satisfy the given dependencies. This condition, called the *separation principle*, would facilitate integrity and therefore database updating, by limiting integrity checks to the parts instead of the whole. Unfortunately, under the pure universal assumption there are conditions on the whole that one cannot circumvent, e.g., that the parts are projections of a single relation.

Let us formalize the separation principle under the two universal relation assumptions. The sets of databases over D called PGSAT and WGSAT consist of those databases that are decompositions of universal relations satisfying Σ, under the pure and weak assumptions respectively. The G in PGSAT, WGSAT stands for global.

$$\text{PGSAT} = \{d : \exists r \models \Sigma, \forall r' \in d \text{ we have } r' = \pi_{R'}(r)\}$$

$$\text{WGSAT} = \{d : \exists r \models \Sigma, \forall r' \in d \text{ we have } r' \subseteq \pi_{R'}(r)\}$$

Inverting the quantification order we get larger sets PLSAT and WLSAT, where L stands for local. In these sets, every relation is a piece of some decomposition of a universal relation satisfying Σ and these relations are put together into databases with only one additional guarantee. The only additional guarantee about the databases is made in the second argument of the intersection. For the pure assumption we require join consistency. For the weak assumption the analogous statement is always true.

$$\text{PLSAT} = \{d : \forall r' \in d, \exists r \models \Sigma \text{ we have } r' = \pi_{R'}(r)\} \cap \{d : \exists r, \forall r' \in d \text{ we have } r' = \pi_{R'}(r)\}$$

$$\text{WGSAT} = \{d : \forall r' \in d, \exists r \models \Sigma \text{ we have } r' \subseteq \pi_{R'}(r)\} \cap \{d : \exists r, \forall r' \in d \text{ we have } r' \subseteq \pi_{R'}(r)\}$$

We clearly have PGSAT $\subseteq$ PLSAT and WGSAT $\subseteq$ WLSAT. The separation principle can be formalized as *surjectiveness* conditions PGSAT = PLSAT and WGSAT = WLSAT, respectively.

Definition 3.2.3: Given $< \{U\}, \Sigma >$, where Σ is a set of full dependencies and D a database scheme over U, then D is *independent* under the pure (weak) universal relation assumption if:
(1) $\Sigma \models \bowtie [D]$, and (2) PGSAT = PLSAT (WGSAT = WLSAT).

Independence was first proposed in [Ris] for Σ a set of fd's and for D having two relation schemes. It was extended to many relation schemes in [BeeRi, MaiMSU] and generalized further to full dependencies in [Var4]. In all this work the pure assumption is postulated, also injectiveness and surjectiveness are shown equivalent to $\Sigma \models \bowtie [D]$ and surjectiveness. Independence under the weak assumption is from [GraY].

Let us now limit attention to Σ's that are sets of fd's. An important computational notion for testing separation is that: "the relation schemes of D must embed a cover for Σ." We say that: D *embeds a cover* for Σ if there is *dependency preservation* as follows.

Let $D = \{R_1, \ldots, R_m\}$ and Σ^+ be the closure of Σ under fd-implication, then $\pi_{R_i}(\Sigma^+), 1 \leq i \leq m$, are the fd's of Σ^+ with attributes exclusively from R_i. Dependency preservation occurs when Σ and $\cup_{i=1}^{m} \pi_{R_i}(\Sigma^+)$ are equivalent sets of fd's. It is easy to see that dependency preservation is decidable, but it is more involved to show that it is decidable efficiently for fd's. The efficient dependency preservation test of [BeeH] (Theorem 3.2.4) can be used to determine independence under the pure assumption (Theorem 3.2.5). This has be extended by [GraY] to an independence test under the weak assumption (Theorem 3.2.5).

Theorem 3.2.4: *Testing if a database scheme embeds a cover for a set of fd's is in* PTIME.

Theorem 3.2.5: *Testing if a database scheme and a set of fd's are independent is in* PTIME, *under both the pure and the weak universal relation assumptions.*

Additional Bibliographic Comments 3.2.6: There has been a fair amount of work on querying independent schemes under the weak assumption, e.g., [AtzC, ChnH, ChnM, GraY, ItoIK, Sag4].

Injectiveness and $\Sigma \models \bowtie [D]$ can be used in-lieu of each other provided we also require surjectiveness, [Var4]. For a comparison of these two and other conditions (which are inequivalent without surjectiveness) we refer to [AroC, BeeMSU, Mai1]. □

3.2.II Normal Form Schemes

Most of design has dealt with fd's, mvd's and the more general jd's; we thus limit Σ here to a set of fd's and jd's. We can use the chase algorithm of Section 2.3.III to construct Σ^+, the closure of Σ with respect to fd-jd-implication.

The designer is faced with the task of replacing a specification $< \{U\}, \Sigma >$ by the multiple specifications $< \{R_i\}, \Sigma_i >, 1 \leq i \leq m$, where $D = \{R_1, \ldots, R_m\}$ is a database scheme over U and Σ_i is as follows. $\Sigma_i = \pi_{R_i}(\Sigma^+)$, that is Σ_i consists of those fd's and jd's implied by Σ whose attributes are from $R_i, 1 \leq i \leq m$. Note that Σ_i is closed under fd-jd-implication.

A number of *normal forms* have been proposed for $< \{R_i\}, \Sigma_i >, 1 \leq i \leq m$. These are conditions on each individual $< \{R_i\}, \Sigma_i >$. Normal forms guarantee less "redundancy" and fewer "update anomalies", [BeeBG]. In the previous subsection we outlined some properties that would make a decomposition into m normal forms desirable. One would like that for these normal forms the decomposition π_D is lossless. In addition, if possible, one would like that D is independent with respect to Σ.

The *first normal form* 1-NF corresponds to our definition of relation, so it is satisfied by any $< \{R_i\}, \Sigma_i >$. Non 1-NF databases have relations whose values are not indivisible, e.g., values could be sets, tuples, relations etc. Non 1-NF databases have been studied as part of complex object data models (Section 4.3). For further normal forms we need some preliminary definitions.

A most important concept for database scheme design is that of a *key* of R_i: "a subset X of R_i is a key if $X \rightarrow R_i$ is in Σ_i and no proper subset of X has this property". Keys capture the essential relationships between attributes. The maintenance of key integrity constraints is supported by most database systems.

$\Sigma_i(\textit{keys})$ is the set of fd's $X \rightarrow R_i$ in Σ_i, where X is a key.

A *nontrivial* dependency is one that does not hold in all databases, i.e., it is not implied by the empty set of dependencies. $X \rightarrow Y$ over R is trivial iff Y is a subset of X. $X \rightarrow\!\!\!\rightarrow Y$ over R is trivial iff either Y is a subset of X or the union of Y and X is R. We can now define *third normal form* (3-NF), from [Cod1]. 2-NF is a weaker normal form that is subsumed by 3-NF [Cod1, Ull1], so we omit it in our presentation.

Let Σ_i be a set of fd's over R_i, closed under fd-implication.

3-NF $< \{R_i\}, \Sigma_i >$ is in 3-NF if for each nontrivial fd $X \rightarrow A$ in Σ_i we have that either X contains a key of R_i or that A is an attribute of a key of R_i.

Given a $< \{U\}, \Sigma >$, where Σ is a set of fd's, it is possible to construct $< \{R_i\}, \Sigma_i >$, $1 \leq i \leq m$, all in 3-NF, such that the decomposition is independent under the pure assumption, (by losslessness and dependency preservation). For Σ a set of fd's, dependency preserving 3-NF's were first constructed in [Ber], in PTIME, using a method called synthesis. It is possible to also satisfy the lossless join condition, again in PTIME, [BisDB, Osb].

Most database systems support integrity maintainance for keys. Thus, if in a normal form we limit Σ_i to the dependencies implied by Σ_i(keys), integrity maintainance will be easier to implement. This is one reason for trying to remove the last condition of 3-NF: "A is an attribute of a key of R_i".

The *Boyce-Codd normal form* (BC-NF) was first defined in [Cod3] for fd's. We present two equivalent definitions from [Fag3] of BC-NF for fd's and jd's. From these definitions it is clear that BC-NF implies 3-NF. The *fourth normal form* (4-NF) is from [Fag3], for which we also provide two equivalent definitions. The *project-join normal form* (PJ-NF) is the strongest normal form for fd's and jd's from [Fag4].

Let Σ_i be a set of fd's and jd's over R_i closed under fd-jd-implication.

BC-NF $< \{R_i\}, \Sigma_i >$ is in BC-NF if for each nontrivial fd $X \rightarrow Y$ in Σ_i we have:
X contains a key of R_i.

4-NF $< \{R_i\}, \Sigma_i >$ is in 4-NF if for each nontrivial mvd $X \rightarrow\!\!\!\rightarrow Y$ in Σ_i we have:
X contains a key of R_i.

The following two equivalent definitions of BC-NF and 4-NF naturally lead to the definition of PJ-NF.

Let Σ_i be a set of fd's and jd's over R_i closed under fd-jd-implication.

BC-NF $< \{R_i\}, \Sigma_i >$ is in BC-NF if for each fd $\sigma \in \Sigma_i$ we have $\Sigma_i(\text{keys}) \models \sigma$.

4-NF $< \{R_i\}, \Sigma_i >$ is in 4-NF if for each mvd $\sigma \in \Sigma_i$ we have $\Sigma_i(\text{keys}) \models \sigma$.

PJ-NF $< \{R_i\}, \Sigma_i >$ is in PJ-NF if for each jd $\sigma \in \Sigma_i$ we have $\Sigma_i(\text{keys}) \models \sigma$.

PJ-NF $\Rightarrow$ 4-NF $\Rightarrow$ BC-NF $\Rightarrow$ 3-NF $\Rightarrow$ 2-NF $\Rightarrow$ 1-NF, see [Fag3, Fag4]. Moreover, using dependency implication and successive lossless decompositions into two parts, it is possible to transform $< \{U\}, \Sigma >$ losslessly into $< \{R_i\}, \Sigma_i >, 1 \leq i \leq m$ which can be in any of these normal forms.

Unfortunately, even BC-NF is sometimes too strong a condition. Testing if a given scheme is in BC-NF is NP-hard [BeeB]. This implies that constructing such normal forms can be inefficient in the size of Σ. Things are even worse when independence is sought, because dependencies are not necessarily preserved. There are sets of fd's Σ, such that no database scheme D exists which is in BC-NF and independent with respect to Σ, under the pure assumption.

Is there an ultimate normal form? The definition of PJ-NF was further generalized in [Fag4] to the *domain-key normal form.* The semantic definition of domain-key normal form is analogous to the one for PJ-NF, where Σ_i might include arbitrary constraints, including bounds on the domain sizes. In [Fag4], this semantic definition was shown to correspond to a formalization of the absence of update anomalies, and thus represent the ultimate normal form. PJ-NF retains the advantage of being constructible via successive lossless decompositions. The generality of domain-key normal form is at the expense of constructibility.

Additional Bibliographic Comments 3.2.7: Third normal form is extended in [LinTK] to eliminate certain redundancies across relations. Database design has been combined with various types of acyclicity, e.g., [GooT, KifB, YuaO]. A normal form for nested relations appears in [OzsY].

Testing for BC-NF is not the only natural NP-hard problem that arises in scheme design with fd's. Finding a minimum size key of a relation scheme with fd's is NP-complete [LucO]. □

4 Queries and Database Logic Programs

4.1 Query Classification

The study of mappings from databases to databases, or *queries*, is essential in order to understand the limitations and the possible extensions of the relational data model.

We have already established the central role of relational algebra or calculus queries, also referred to as f.o. queries. It is important to distinguish between the queries themselves as mappings and the relational algebra expressions $E(\cdot)$ or calculus expressions $F(\cdot)$, that are programs denoting these mappings. We have also investigated tableaux expressions $T(\cdot)$, which define a proper subset of the f.o. queries; see [ChaH2]. On the other hand we have seen that the transitive closure query is not a f.o. query. Interestingly, using a deductive or a universal relation data model it is possible to express transitive closure via a program $R[\cdot,\Sigma]$ or $X[\cdot,\Sigma]$, where Σ is a set of rule dependencies

In this section we present a systematic view of the classes of queries that are most common in database theory. For this we follow the general classification of [Cha2].

4.1.I The Computable Queries and their Data Complexity

The first natural question in this context is to delimit the queries that one can conceivably compute. For this let us think of the database d over D as a finite relational structure $(\bar{\delta}, r_1, \ldots, r_m)$, where $\bar{\delta}$ is a finite set containing the set of values δ appearing in the database. (This is a more liberal view of what constitutes the database domain, than the use of δ in Section 2.2.II). We are clearly interested in computable mappings between finite relational structures.

Computability for unrestricted relational structures has been extensively studied in mathematical logic, e.g., [Mos], but the emphasis has been on infinite models. The finite case is qualitatively different, even if the syntactic concepts are the same, see [Gur1, Gur2]. In fact, the study of database queries has directly contributed to our understanding of the theory of finite models.

The class of *computable queries* is defined by Chandra and Harel in [ChaH1]. It captures three basic intuitions. First, that the query has to be a partial recursive function f from finite relational structures to finite relational structures. Second, that for each $d = (\bar{\delta}, r_1, \ldots, r_m)$, the values appearing in $f(d)$ are in $\bar{\delta}$. Finally that, isomorphic inputs should be mapped to isomorphic outputs. The third restriction is because relations are unordered collections of tuples; therefore the in [ChaH2] to define a class of computable queries, the *fixpoint queries* FP. Fixpoint queries are an important extension of f.o. queries. They have been studied extensively through universal and deductive data models and, more recently, through database logic programs. QPTIME properly contains FP, since computing the parity of the database domain is not in FP. There are fixpoint queries that are logspace-complete in PTIME and thus most probably not in QNC. This is an important distinction from f.o. queries, since much of the efficiency of the relational data model is due to the large degree of potential parallelism inherent in f.o. queries (Theorem 2.2.7).

The relationship between complexity classes and various logics has been investigated in detail; see the survey [Imm4]. Many of the results in the area assume the ability to test for $<$, which is a special relation that linearly orders the database domain. This change in the rules of the game is certainly possible, since strings are used to represent values and strings are usually equipped with lexicographic order. Also, the ability to test for $<$ can add to the expressive power of the language. The presence of $<$ has allowed the comparison of a variety of extensions of f.o. logic (e.g., especially when the resulting data complexity is in PTIME [Imm3]) by identifying expressibility with computational complexity. On the other hand, $<$ is not a particularly intuitive programming construct.

Most of the analysis of fixpoint queries, within database theory, does not assume the presence of $<$. However, its addition has an interesting effect on expressibility: *the queries with* PTIME *data complexity are precisely the queries expressible by fixpoint formulas over finite structures with* $<$. This was shown independently in [Imm2], [Saz], and [Var5]. In the proofs, all PTIME computations are simulated using $<$ (not only the computations satisfying condition (3) of Definition 4.1.1). An interesting open issue is to design a language for exactly QPTIME.

Additional Bibliographic Comments 4.1.2: The data complexity of many query languages (both procedural and declarative) is determined in [Var5]. Two more measures are defined in [Var5], expression complexity and expression + data complexity. They correspond, respectively, to asymptotic growth in the program only and in both program + data. These measures are typically one exponential higher than data complexity.

The finer structure of f.o. queries is examined in [ChaH2], where a query hierarchy is built based on the alternation of first order quantifiers in a calculus expression $F(\cdot)$.

A large repertoire of programming language constructs can be added to relational algebra and the expressive power of the resulting query languages is examined in [Cha1]. Various *looping* constructs, in particular, lead to interesting query classes, see [ChaH2, Cha1, Cha2, AbiV1, AbiV2]. *Nondeterminate* queries that are relations, as opposed to functions, are studied in [AbiV1, AbiV2], as part of an investigation of update languages. Programming formalisms for fixpoint queries are also examined in [HarK, LakM]. □

4.1.II The Fixpoint Queries

Transitive closure can be expressed by the addition to relational calculus of least fixpoint equations, see [AhoU]. If R_2 is interpreted as a directed graph then this graph's transitive closure is the least, under subset ordering, interpretation of R_1 that satisfies the following equation among query expressions: $R_1(xy) \equiv R_2(xy) \vee \exists z(R_2(xz) \wedge R_1(zy))$.

The right-hand side of this equation is a f.o. formula that is monotone and positive in R_1. In general, let $\varphi(\vec{x}; R_1)$ be a f.o. formula, where $\vec{x}$ is its vector of distinct free variables of size $n_1 \geq 0$ and R_1 is a relation symbol of arity n_1. This formula $\varphi(\vec{x}; R_1)$ might have other relation symbols occurring in it, but we highlight R_1 because it will be on the left-hand side of the equation, (i.e., it will be a variable).

answer to a query should not be affected by the implementation details of how this set is stored, i.e., which is the first tuple etc. More specifically, any automorphism on the input structure should leave the output unchanged; this has also been proposed in [AhoU, Ban1, Par]. The isomorphism requirement is the natural generalization of the automorphism requirement. Let us be more precise.

Consider two databases over D, say $d = (\bar{\delta}, r_1, \ldots, r_m)$ and $d' = (\bar{\delta}', r'_1, \ldots, r'_m)$. These databases are *isomorphic* if there is a bijection $h : \bar{\delta} \rightarrow \bar{\delta}'$, that extends componentwise to tuples and is such that for all $i, 1 \leq i \leq m$, $t \in r_i \Rightarrow h(t) \in r'_i$ and $t' \in r'_i \Rightarrow h^{-1}(t') \in r_i$. The *isomorphism* h is denoted by $d \leftrightarrow^h d'$, and is called an *automorphism* if $d = d'$.

Definition 4.1.1: Let D be a database scheme and R a relation scheme such that $|R| = n$. A *computable query* from D to $\{R\}$ is a function f, which on input database $d = (\bar{\delta}, r_1, \ldots, r_m)$ over D has output database $f(d)$ over $\{R\}$ such that:
(1) f is partial recursive,
(2) $f(d) = (\bar{\delta}, r)$, where $r \subseteq \bar{\delta}^n$,
(3) if $d \leftrightarrow^h d'$ then $f(d) \leftrightarrow^h f(d')$.

It is easy to see that f.o. queries are computable and that the transitive closure query is computable. Therefore, not all computable queries are f.o. ones. A programming language is proposed in [ChaH1] for the computable queries. This is an extension of relational algebra, where the crucial new features are a *while* looping construct and variables that can be assigned relations of any arity. A different programming language with the same expressive power is proposed in [AbiV1]. Unbounded arity variables are replaced there by the ability to create new values. Necessary and sufficient conditions on programming language constructs in order to express the computable queries is an interesting research issue.

Computable queries can be classified according to their data complexity (see Definition 2.2.6). Thus, class QPTIME consists of the computable queries with PTIME data complexity. Note that there is a difference between all queries with PTIME data complexity and QPTIME queries, the latter must satisfy the isomorphism condition of Definition 4.1.1. Similarly, we have computable queries for other complexity classes, (i.e., QLOGSPACE, ...). It is shown in [ChaH2] that, under some weak technical condition (complexity class closure under many-one logspace reducibility), the various computable query classes defined this way are related to each other as the complexity classes. For example, QLOGSPACE = QPTIME iff LOGSPACE = PTIME.

As we saw in Theorem 2.2.7 f.o. queries are contained in QLOGSPACE. There are, however, simple queries in QLOGSPACE that are not f.o. queries, e.g., computing the parity of the database domain [ChaH2]. Thus, we have the following inclusions, where the containments ($\subseteq$) are strongly conjectured to be proper ($\subset$):

F.O. $\subset$ QLOGSPACE $\subseteq$ QNLOGSPACE $\subseteq$ QNC $\subseteq$ QPTIME $\subseteq$

$\subseteq$ QNPTIME $\subseteq$ QPHIER $\subseteq$ QPSPACE $\subset$ QEXPTIME $\subset$ COMPUTABLE

There are close connections between *second order* logic, interpreted over finite structures, and this query classification. As is shown in [Fag1]: *the queries in* QNPTIME *are precisely the queries expressible by existential second order formulas over finite structures* (these formulas consist of a prefix of existential second order quantifiers followed by a f.o. formula). A closely related result was independently shown in [JonS]. The connection to second order logic was extended in [Sto]: *the queries in* QPHIER *are precisely the queries expressible by second order formulas over finite structures.* For other connections to second order logic we refer to [Lei].

A natural (infinitary as opposed to second order) extension of the f.o. predicate calculus with equality is fixpoint logic, also known as the μ calculus [Mos]. This was used as a query language

Formula $\varphi(\vec{x};R_1)$ is *monotone* in R_1 when: given any database d over all the relation symbols of $\varphi(\vec{x};R_1)$ except R_1, then for all r,r' relations over R_1, $r \subseteq r'$ entails $\{\vec{c}: \varphi(\vec{c};r)\} \subseteq \{\vec{c}: \varphi(\vec{c};r')\}$ where the $\vec{c}$ are vectors of size n_1 of values from the domains of d, r, r'.

A sufficient condition for the monotonicity of $\varphi(\vec{x};R_1)$ in R_1 is that every occurrence of R_1 in this formula is under an even number of negations. In that case we say that $\varphi(\vec{x};R_1)$ is *positive* in R_1. Every monotone in R_1 f.o. formula (viewed as a query expression) is unrestricted equivalent to some positive in R_1 f.o. formula. This is known as Lyndon's lemma in model theory. Thus, the syntactic condition of positivity, which is easy to test, matches the semantic condition of monotonicity for unrestricted databases. Unfortunately, the transformation from monotone to positive is not effective because, testing for monotonicity is undecidable in both the unrestricted and finite cases, [Gur1]. A surprising fact for finite structures is that monotone in R_1 f.o. formulas are more expressive (viewed as query expressions) than positive ones. The following is from [AjtG]:

Theorem 4.1.3: *For finite structures, there is a monotone in R_1 f.o. formula that is not equivalent to any positive in R_1 f.o. formula.*

For formulas $\varphi(\vec{x};R_1)$ monotone in R_1, the equation $R_1 \equiv \varphi(\vec{x};R_1)$ has a least fixpoint solution on any database d over all the relation symbols of $\varphi(\vec{x};R_1)$ except R_1. This follows from the Tarski-Knaster Theorem. The *least fixpoint of $\varphi(\vec{x};R_1)$ on d*, denoted r_∞ can be be evaluated in an unbounded but finite number of stages (by the finiteness of d) as follows:

$r_0 = \emptyset, r_{i+1} = \{\vec{c}: \varphi(\vec{c};r_i)\}, r_\infty = \cup_{i\geq 0} r_i$.

If $\varphi(\vec{x};R_1)$ is neither positive nor monotone in R_1 it is still possible to define a solution for equation $R_1 \equiv \varphi(\vec{x};R_1)$ using inflationary semantics. The *inflationary (or inductive) fixpoint of $\varphi(\vec{x};R_1)$ on d*, denoted r_∞ can be be evaluated in an unbounded but finite number of stages as follows:

$r_0 = \emptyset, r_{i+1} = r_i \cup \{\vec{c}: \varphi(\vec{c};r_i)\}, r_\infty = \cup_{i\geq 0} r_i$.

The inflationary fixpoint of $\varphi(\vec{x};R_1)$ on d is the same as the least fixpoint of $(\varphi(\vec{x};R_1) \vee R_1(\vec{x}))$ on d. Also, the inflationary and the least fixpoint on d coincide if $\varphi(\vec{x};R_1)$ is monotone. In both the above fixpoint constructions it is easy to see that: $r_i \subseteq r_{i+1}, i \geq 0$.

The *fixpoint queries* FP are defined in [ChaH2] using fixpoint formulas. Informally, fixpoint formulas are obtained recursively using the f.o. constructors $\exists, \forall, \vee, \wedge, \neg$ as well as a fixpoint constructor μ. The constructor μ binds a relation symbol R_1 appearing free in a fixpoint formula and is applicable only when R_1 appears positively in this formula, i.e., under an even number of negations. The semantics of $\mu R_1 \psi(\vec{x};R_1)$ on d are the same as the least fixpoint of $\psi(\vec{x};R_1)$ on d, (where, recursively, $\psi(\vec{x};R_1)$ is a fixpoint formula). We refer to [ChaH2] for the detailed definitions. We omit them here because by Theorem 4.1.5 below one application of μ suffices.

Given Theorem 4.1.3, one might consider the set of fixpoint formulas, formed under the positivity restriction on the applicability of μ, as a somewhat arbitrary choice. By keeping the least fixpoint semantics, but making the applicability of μ more liberal (μ applicable to all monotone formulas) one can define a set of queries FM. Of course in this case it is undecidable to test when μ is applicable. By adopting the inflationary semantics, one can remove all restrictions on the applicability of μ and define a set of queries FI. It follows from the definitions that FP $\subseteq$ FM $\subseteq$ FI. The following theorem from [GurS] testifies to the robustness of FP; we stress the finiteness because the theorem is not true for unrestricted databases.

Theorem 4.1.4: *For finite structures,* FP = FM = FI.

In [Imm2], Immerman showed another surprising result, also not true for unrestricted databases: *the complement, with respect to the database domain, of the least fixpoint of a f.o. formula on a database can be expressed as the least fixpoint of some other formula on this database.* In fact, fixpoint formulas have equivalent normal forms where the μ constructor is applied only once, [GurS, see also Imm2]. This normal form has been recently simplified in [AbiV2], where it is shown that the $\forall$ constructor is unnecessary. The fixpoint queries can be expressed taking inflationary fixpoints of *existential* f.o. formulas and their projections. Thus, we need to consider only inflationary fixpoints of f.o. formulas with an $\exists^*$ quantifier prefix and $\neg$ limited to the matrix; see also [Remark Section 6 of Gur2].

We summarize the above discussion in Theorem 4.1.5 using a formalism that is closer to database logic programs. We use a system of equations for which we define simultaneous least and inflationary fixpoints on databases. These equations are mutually recursive. We could have eliminated the mutual recursion and have a single equation, as is done with fixpoint formulas. This elimination imposes certain technical restrictions on the vocabulary in [Imm2] and is why $\neq$ is used in [ChaH2]. The use of a system has two advantages: it is close to the notation used for database logic programs and it leads to the definition of equation width.

Let us therefore consider a system of equations $R_i \equiv \varphi_i(\vec{x}_i; R_1, \ldots, R_m)$, $1 \leq i \leq m$, where each $\varphi_i(\vec{x}_i; R_1, \ldots, R_m)$ is a f.o. formula, $\vec{x}_i$ is its vector of n_i distinct free variables and R_i is a relation symbol of arity $n_i \geq 0$. The *equation width* is the maximum n_i for $1 \leq i \leq m$. The notation emphasizes the fact that the left-hand sides are the relation variables (they need not occur in each right-hand side). A database d interprets all the relation symbols occurring in the right-hand sides except for $R_1, \ldots, R_m$. The inflationary fixpoint of a system of equations on d is defined analogously to the inflationary fixpoint of one equation on d; only a sequence of databases $d_0, \ldots, d_\infty$ must be used instead of a sequence of relations $r_0, \ldots, r_\infty$, [ChaH3]. If the $\varphi_i(\vec{x}_i; R_1, \ldots, R_m)$, $1 \leq i \leq m$, are monotone in $R_1, \ldots, R_m$ the same can be done for the least fixpoint on d.

Theorem 4.1.5: *For finite structures, each fixpoint query can be expressed as the projection on R_1 of the inflationary fixpoint of $R_i \equiv \varphi_i(\vec{x}_i; R_1, \ldots, R_m)$, $1 \leq i \leq m$, where each $\varphi_i(\vec{x}_i; R_1, \ldots, R_m)$ in this system of equations is an existential f.o. formula.*

Example 4.1.6: First, let us illustrate inflationary fixpoints of systems of equations using a system that fails to compute the complement of the transitive closure.

$R_1(xy) \equiv \neg R_2(xy)$

$R_2(xy) \equiv \exists z R_3(xy) \vee (R_3(xz) \wedge R_2(zy))$

Given d a database over $\{R_3\}$ (the input graph), let r_∞, r'_∞ be the projections on R_1, R_2 of the inflationary fixpoint on d. Then $r_0 = r'_0 = \emptyset$ and $r'_1 =$ (the input graph) and $r_1 =$ (all possible pairs of nodes). In fact, $r'_\infty =$ (the transitive closure of the input graph) and $r_\infty =$ (all possible pairs of nodes).

The following system (from [AbiV2]) succeeds in computing the complement of the transitive closure, given an input graph as a database over $\{R_5\}$. The semantics used are inflationary.

$R_1(xy) \equiv \exists x' \exists y' \neg R_4(xy) \wedge R_3(x'y') \wedge \neg R_2(x'y')$

$R_2(xy) \equiv \exists x' \exists y' \exists z' R_4(xy) \wedge R_4(x'z') \wedge R_4(z'y') \wedge \neg R_4(x'y')$

$R_3(xy) \equiv R_4(xy)$

$R_4(xy) \equiv \exists z R_5(xy) \vee (R_5(xz) \wedge R_4(zy))$

The idea is that, as the transitive closure is inserted in R_4 by the successive stages of the fixpoint construction it is also copied in R_2 and R_3. The copying is one stage behind the transitive closure construction in R_4 and its last stage is copied in R_3 but not R_2. By finiteness there is a last stage! Only then are tuples inserted in R_1, those that have not been inserted in R_4. □

Additional Bibliographic Comments 4.1.7: The equational formalism used in Theorem 4.1.5 is identical with the language Datalog$^\neg$ defined in [KolP] and [AbiV2]. The existence and uniqueness of least fixpoints of existential f.o. formulas is studied in [KolP], from the point of view of computational complexity.

The fixpoint queries are further classified in [ChaH2] according to *fixpoint width.* Although some progress has been made in understanding the properties of fixpoint width [Gai], there are still many open questions. The fixpoint width of [ChaH2] differs from the equation width used above and in [BeeKBR, Der, Mos]; the difference is in the treatment of mutual recursion. For some recent results on equation width see [AfrC, DubM]. □

4.2 Database Logic Programs

The semantics, the optimization and the evaluation of database logic programs are some of the most active research topics in database theory today. This activity is bound to render obsolete any attempted survey of the area. There are, however, strong ties to other more mature topics, such as the theory of dependencies and the theory of queries. We outline the state of the field by emphasizing these connections.

In database logic programs, Datalog$^\neg$ and the fixpoint queries play the role that relational calculus and the f.o. queries play in the relational data model. Datalog, a sublanguage of Datalog$^\neg$, has received most of the attention in terms of program optimization and evaluation. This is analogous to the importance of positive existential programs in the relational data model. Most of this work has direct applications to the development of new efficient knowledge base systems.

4.2.I From Datalog to Datalog$^\neg$

Datalog and Datalog$^\neg$ Syntax: The vocabulary of a Datalog$^{(\neg)}$ program H is the set of relation symbols occurring in H. It is partitioned into: the intensional database symbols IDB's $\{R_1, \ldots, R_m\}$ and the extensional database symbols EDB's $D = \{R_{m+1}, \ldots, R_k\}$ (we also call the EDB's D the set of *input relation symbols*). The IBD symbol R_1 is called the *output relation symbol.*

H consists of Datalog$^{(\neg)}$ *rules*, where every rule has a *head* and a *body.* Only IDB's occur in the heads of rules and every IDB occurs in the head of some rule.

Each Datalog$^\neg$ rule is of the form $R(\vec{x})$:— φ. (1) The head $R(\vec{x})$ consists of R an IDB of arity $n \geq 0$ and of a vector $\vec{x}$ of variables. Without loss of generality we can take $\vec{x} = xy\ldots$ the vector of the first n distinct variables, in some standard ordering of the variables (i.e., heads are normalized). (2) The body φ is a list of equality atoms, relational atoms and negations of such atoms. This list is terminated by a point (.) and the atoms are separated by commas (,). The variables in $\vec{x}$ must occur in φ.

Datalog rules are Datalog$^\neg$ rules without any negated atoms in their bodies. □

Datalog and Datalog$^\neg$ Fixpoint Semantics: Given Datalog$^{(\neg)}$ program H, construct the following formula for each IDB $R_i, 1 \leq i \leq m$: let $\varphi_i(\vec{x}; R_1, \ldots, R_m)$ be the disjunction of the bodies of all rules with head $R_i(\vec{x})$, where each body is a conjunction of its list of atoms prefixed by existential quantifiers for all variables except $\vec{x}$. That is, we have changed in the bodies comma (,) into and ($\wedge$), point (.) into or ($\vee$), and have existentially quantified variables not in the head.

We form the system $S(H)$ of equations: $R_i(\vec{x}) \equiv \varphi_i(\vec{x}; R_1, \ldots, R_m)$, $1 \leq i \leq m$.

Let d be a database over input symbols D, i.e., over the vocabulary except $\{R_1, \ldots, R_m\}$. If H is a Datalog$^\neg$ program, then $S(H)$ has an inflationary fixpoint on d (see Section 4.1.II), moreover all φ_i's are f.o. existential. If H is a Datalog program then the R_i's appear only positively in the right hand sides of the equations and $S(H)$ has a least fixpoint on d (see Section 4.1.II), which by monotonicity is the same as the inflationary fixpoint.

On input database d over D the output of H is the projection on R_1 of the inflationary fixpoint (or the least fixpoint in the case of Datalog) defined by $S(H)$ on d. $\square$

It is clear that Datalog queries are Datalog$^\neg$ queries, which in turn are fixpoint queries. We have assigned fixpoint semantics to Datalog and Datalog$^\neg$ programs. Such fixpoints semantics can be computed using the standard Tarski construction (see Section 4.1.II). This construction is also known as *naive bottom-up evaluation* and can be viewed as an "operational semantics" for database logic programs.

Note the similarity of Datalog notation with the Prolog programming language syntax and the differences in their "operational semantics". The Datalog rules (negation is excluded here) are Horn clauses [End], when one views (:—) as ($\Leftarrow$) and (,) as ($\wedge$) and (.) as ($\vee$). Recall that there are no negations in the bodies of Datalog rules. In the Horn clause case there is a fundamental fact from logic programming [Apt, AptV]: *that least fixpoint semantics coincide with minimum model semantics.* So let us present the Datalog minimum model semantics.

Datalog Minimum Model Semantics: Given a Datalog program H and a database d over D, the *minimum model* $\mathcal{M}(d, H)$ is defined as follows. The *Herbrand base* of d and H is the set of all possible *ground* relational atoms (i.e., relational atoms with values substituted for the variables) that can be constructed using the vocabulary of H and the values in d. The Herbrand base is finite since d and H are finite.

$\mathcal{M}(d, H)$ is the least, under set inclusion, subset of the Herbrand base containing d and satisfying each rule of H interpreted as a universally closed Horn sentence. That is, in each rule (:—) is changed into ($\Leftarrow$) and comma (,) into ($\wedge$) and *all* the variables of the resulting Horn clause are universally quantified.

On input database d over D the output of H is the projection on R_1 of $\mathcal{M}(d, H)$. $\square$

Theorem 4.2.1: *The least fixpoint and the minimum model semantics of a Datalog program coincide.*

It is clear that rule dependencies are Datalog rules. There are only minor differences in conventions of notation, e.g., in Datalog rules instead of having repeated variables in the head we use $=$ in the body. That is the only nontrivial use of $=$ that we make in Datalog and can be eliminated with repeated variables. We have avoided using impure features in Datalog, such as $\neq$ [ChaH3] or constants in the rules. This is in keeping with our definition of relational calculus without constants and with our use of systems of equations to express mutual recursion. It is simple to add constants to the rules. As long as these constants are added to the Herbrand base, all the definitions generalize in the straightforward fashion. If the only negations are inequalities in the bodies we have a sublanguage of Datalog$^\neg$ called Datalog$^{\neq}$.

Datalog queries are a proper subset of the fixpoint queries, because of the absence of negation. They are incomparable with the f.o. queries, because of the ability to express transitive closure in

Datalog, but they have many interesting properties. They contain the positive existential queries, for which the homomorphism technique was originally developed. On the other hand, it is easy to see that Datalog$^\neg$ queries are exactly the fixpoint queries (see Theorem 4.1.5).

Querying in deductive and universal relation data models with rule dependencies can be reformulated as querying with Datalog. The chase may now be viewed as an *evaluation* algorithm. Consider the input database d as a set of untyped tagged tableaux that are chased using the rules of program H as tuple-generating rule dependencies (see [GraMV]). The result of the chase is the minimum model $\mathcal{M}(d, H)$. Note that, in the construction of least fixpoints in Datalog and of inflationary fixpoints in Datalog$^\neg$, a particular order of rule applications is used. A fine point is that, by the Church-Rosser property of the chase, the order of rule applications is immaterial for Datalog. However, it is significant for Datalog$^\neg$. Finally, as we shall see, another use of the chase algorithm is for the *optimization* of Datalog programs.

Example 4.2.2: Let us rewrite the systems of Example 4.1.6 in Datalog$^\neg$ notation. The first system becomes the following set of rules. Note that the second and third rule form a Datalog program for the transitive closure.

$R_1(xy)$:— $\neg R_2(xy)$.

$R_2(xy)$:— $R_3(xz), R_2(zy)$.

$R_2(xy)$:— $R_3(xy)$.

The second system becomes the following set of rules.

$R_1(xy)$:— $\neg R_4(xy), R_3(x'y'), \neg R_2(x'y')$.

$R_2(xy)$:— $R_4(xy), R_4(x'z'), R_4(z'y'), \neg R_4(x'y')$.

$R_3(xy)$:— $R_4(xy)$.

$R_4(xy)$:— $R_5(xz), R_4(zy)$.

$R_4(xy)$:— $R_5(xy)$. □

Both sets of rules in this example are what is known as *stratified*. Their IDB's can be partitioned into a *linearly ordered set of strata*, such that: (1) a positive atom $R'(\vec{z})$ is in the body of a rule with head $R(\vec{x})$ iff the stratum of R' is less or equal than the stratum of R, and (2) a negative atom $\neg R'(\vec{z})$ is in the body of a rule with head $R(\vec{x})$ iff the stratum of R' is strictly less than the stratum of R. In Example 4.2.2, the ordered strata for the first set of rules are $< \{R_2\}, \{R_1\} >$ and for the second set of rules $< \{R_4, R_3\}, \{R_2\}, \{R_1\} >$ or $< \{R_4\}, \{R_3, R_2\}, \{R_1\} >$ could be possible ordered strata.

Stratified sets of rules can be assigned least fixpoint semantics [ChaH3] or equivalent minimum model semantics [AptBW] by evaluating the strata bottom-up and constructing the least fixpoint for a lower stratum before going on to a higher one. The particular linear order chosen is immaterial and, in addition, within each stratum a Church-Rosser property applies for the evaluation of rules just like in Datalog. Thus, the stratified evaluation of certain Datalog$^\neg$ programs can be used to define the class of *stratified* queries. Note that in Example 4.2.2, both sets of rules express the complement of the transitive closure, when they are evaluated in a stratified way. Under the inflationary semantics only the second set of rules computes the complement of the transitive closure.

Stratified queries have been proposed as a natural and implementable means of introducing negation into database logic programs, see [AptBW, ChaH3, Naq, Vgel]. They properly contain both Datalog and f.o. queries. Operationally they are related to Clark's *negation as failure* principle for general logic programs, [Cla].

A difficulty with this approach is that, the number of strata in a stratified program is bounded. Using the inflationary semantics of Datalog$^\neg$ it is possible to simulate an unbounded number of strata. As shown in [Dah, Kol] the stratified queries are a proper subset of the fixpoint queries. Let P.E. be the positive existential queries, S-DATALOG the stratified queries etc.

Theorem 4.2.3: *For finite databases,* DATALOG *and* F.O. *are incomparable and we have,*
$$\text{P.E.} = (\text{DATALOG} \cap \text{F.O.}) \subset (\text{DATALOG} \cup \text{F.O.}) \subset \text{S-DATALOG} \subset \text{DATALOG}^\neg = \text{FP} \subset \text{QPTIME}$$

This theorem summarizes our understanding of the expressibility of negation. One point in the theorem must be further clarified. It is easy to see that P.E. $\subseteq$ (DATALOG $\cap$ F.O.). For unrestricted databases P.E. = (DATALOG $\cap$ F.O.), by a compactness argument. It has been recently claimed by Ajtai and Gurevich that it also holds in the finite case (see [Cos4] for special cases).

Additional Bibliographic Comments 4.2.4: The introduction of negation into logic programs has been a major topic of research. We have limited our exposition to negation in database queries and stressed the properties of negation in finite model theory. We refer to [Apt, Min2] for more information on negation and minimal models. We refer to [Vge2] for some recent results on minimum model semantics, that also deal with inflationary fixpoints. □

4.2.II Query Optimization and Stage Functions

We have defined minimum model and, equivalent, least fixpoint semantics for Datalog programs (programs for short in this subsection). An alternative and useful way of presenting the construction of least fixpoint semantics is via derivation trees.

A ground atom t is in the minimum model $\mathcal{M}(d,H)$ iff there is a derivation tree for t from d and H. A derivation tree for t is a tree, where each node of the tree is labeled by a ground atom such that: (1) each leaf is labeled by a tuple of d; (2) for each internal node there is a rule of H, whose variables can be instantiated so that the head is the label of that node and the body is the set of labels of its children; and (3) the root is labeled by t.

The depth of a derivation tree is the length of its longest path and its size is the number of nodes. Derivation trees are descriptions of computations of special alternating Turing machines associated with each database logic program, [UllVg, Kan2]. They are particular to Datalog and are commonly used to analyse logic programs.

Definition 4.2.5: Let $H(d)$ be the output of program H on input database d. The *stage function* of H is $\xi(n,H) = \max\{\xi(d,H) :$ where $|d| \leq n$ and $|d|$ is the size of d in a fixed binary encoding$\}$ where $\xi(d,H) = \min\{i :$ for each t in $H(d)$ there is a derivation tree of depth $\leq i$ $\}$.

Stage functions are a major topic in [Mos], where they are defined for unrestricted databases. The name stage function is used, because $\xi(d,H)$ is the first iteration of the naive bottom-up evaluation, by which all the output has been constructed. After this iteration the output does not change, even if other ground atoms are added to the minimum model. Stage functions can also be defined for Datalog$^\neg$ programs, using this intuition about iterations and the construction of inflationary fixpoints.

Recall that, by convention, the input symbols are the EDB's and the output symbol is an IDB. If H is a program let program $\bar{H}$ have the same set of rules, but all the vocabulary as output symbols.

We use the term *program stage function* of H for $\xi(n, \bar{H})$. If a H has only a single IDB then we have that $\xi(n, \bar{H}) = \xi(n, H)$.

The distinction between IDB's and EDB's is somewhat artificial, and is based on the fact that the IDB's are initialized to $\emptyset$ in the iterative construction of fixpoints. It is possible to define *uniform* least and inflationary fixpoint semantics, by initializing the IDB's as arbitrary relations. Then the vocabulary of H, the output and the input symbols are all the same. Under these uniform semantics, we have the *uniform stage function* of H, denoted $\bar{\xi}(n, \bar{H})$. An interesting special case are *single rule programs* or *sirups*: they consist of a single rule and are given the uniform least fixpoint semantics [Kan2]. So they have a single IDB and the uniform stage function.

Let us relate the stage functions of a program H. The *equation width* of H is the largest arity of an IDB in H. It is easy to show that for all n, $\xi(n, H) \leq \xi(n, \bar{H}) \leq \bar{\xi}(n, \bar{H})$. and $\bar{\xi}(n, \bar{H}) = O(\log n^k)$, where k is the equation width of H. Programs with equation width 1 are called monadic, 2 binary, etc. Note that programs can have arbitrary arity EDB's occurring in the bodies of the rules.

Example 4.2.6: There are some other classes of Datalog programs, that have been investigated. One example are *linear* programs: in the bodies of the rules of these programs there can be at most one IDB occurrence. Another example are *chain* programs [UllVg]: their syntax greatly resembles that of context free grammars. Recall that the semantics for sirups are uniform (i.e., all relation symbols are initialized to arbitrary nonempty relations).

Program H_0 is a linear, chain, binary sirup known as the "same generation" program:
$R_1(xy)$:— $R_2(xz), R_1(zz'), R_3(z'y)$.

Program H_1 is a nonlinear, chain, binary sirup known as the "ancestor" program :
$R_1(xy)$:— $R_1(xz), R_1(zy)$.

Note the difference of the ancestor program from the uniformly interpreted sirup H',
$R_1(xy)$:— $R_1(xz), R_2(zy)$.

H_1 expresses the same query as the program consisting of the two rules:
$R_1(xy)$:— $R_1(xz), R_2(zy)$. $R_1(xy)$:— $R_2(xy)$.

Program H_2 is a nonlinear, monadic sirup that is not a chain. It expresses the "path accessibility problem". This is the prototypical logspace-complete problem in PTIME discovered by Cook:
$R_1(x)$:— $R_1(y), R_1(z), R_2(xyz)$.

Program H_3 is a sirup whose uniform stage function is $O(1)$:
$R_1(xy)$:— $R_2(x), R_1(zy)$. □

The conventions on output and uniformity lead to many notions of containment for two programs with the same input and output symbols. The containment that we have previously encountered for other query languages has a direct analog for Datalog programs, called *containment*. We use the term *program containment* when the output symbols are all symbols and the term *uniform containment* under the uniform semantics. The first basic observation is that: *all the kinds of containment are the same as their unrestricted containment counterparts.* It is a special feature of Datalog that containments and unrestricted containments coincide and are therefore co-r.e. The second observation is that: *uniform containment* $\Rightarrow$ *program containment* $\Rightarrow$ *containment*, but the converses need not hold.

Using context free language theory, one can show that program containment and containment are undecidable (Π^0_1 -complete) even for linear, binary, chain programs [CosGKV, Shm]. Using regular language theory, even for nonchain programs, one can show that all the kinds of containment are decidable for monadic programs [CosGKV]. The important connection of program rules and rule dependencies becomes clear when we consider uniform containment. It was first noted in [CosK3] for full template dependencies (ftd's) and in [Sag6] for all rules that, uniform containment is the same as rule dependency implication and therefore EXPTIME-complete. Therefore, the chase is useful as a Datalog program optimization technique. The many possible notions of equivalence between logic programs are investigated in [Mah].

Theorem 4.2.7: *Program H is uniformly contained in program H' iff the set of rule dependencies H is implied by the set H'.*

The presence of recursion in the language generates new questions, beyond containments. The most fundamental of these questions is whether recursion is bounded. As shown in [GaiMSV] undecidablity of recursion boundedness can be translated into undecidability for most other questions concerning recursion. Interestingly, the first bounded recursion definitions and uses appeared for universal models [MaiUV] and deductive data models [MinN].

Definition 4.2.8: A program H is *bounded* if $\xi(n,H) = O(1)$, it is *program bounded* if $\xi(n,\bar{H}) = O(1)$, and it is *uniformly bounded* if $\bar{\xi}(n,\bar{H}) = O(1)$.

Program H_3 in Example 4.2.6 is uniform bounded. Clearly: *uniform boundedness $\Rightarrow$ program boundedness $\Rightarrow$ boundedness*, but the converses need not hold. From the definition one can see that undecidability of uniform boundedness automatically translates into undecidability for program boundedness etc. Also, that decidability of boundedness implies decidability of all the other kinds.

Boundedness, and all the other varieties as well, imply that the query expressed by the program is a positive existential query and therefore a f.o. query. It is also true that uniform unboundedness, and all the other varieties as well, imply that the query expressed by the program is not a positive existential query [NauS]. If we allowed unrestricted databases then, by a compactness argument, unboundedness implies non f.o. expressibility. For databases this has been recently claimed by Ajtai and Gurevich. It is true because of Datalog's special features. For Datalog$^{\neg}$ the situation is different, as Kolaitis has observed, there is an unbounded Datalog$^{\neg}$ program which is f.o. expressible over finite structures.

Another example of the special character of Datalog from [CosK3] is a *gap* theorem: *uniform unboundedness implies $\bar{\xi}(n,\bar{H}) = \Omega(\log n)$.*

Boundedness is decidable for various subclasses of Datalog programs. This was first demonstrated in [Ioa, Nau1], for subclasses of linear programs with PTIME decision procedures. For chain programs, testing for boundedness becomes testing for context free language finiteness. For full template dependencies uniform boundedness is shown decidable in [Sag5]. In this case (without loss of generality [FagMUY]) the program is a sirup and uniform boundedness in NP-hard. Boundedness is shown decidable for monadic programs in [CosGKV]; regular language theory is used to analyze the decidability and the complexity of all monadic boundedness problems. Other decidable cases appear in [NauS, Var11].

Unfortunately, boundedness is an undecidable property for Datalog programs in general. This was shown for uniform boundedness of linear, single IDB, 7-ary programs and for program boundedness of linear, single IDB, 4-ary programs in [GaiMSV]. Even boundedness of linear, single IDB, binary programs is undecidable [Var11].

Uniformity is not the only assumption, under which the complexities of noncontainment and boundedness differ. Whereas program boundedness is r.e. (Σ_1^0-complete) [GaiMSV], boundedness is even harder (Σ_2^0-complete), [CosGKV]. This is because of the possible final projection. Thus,

Theorem 4.2.9: *Uniform boundedness is undecidable, even for linear, single* IDB*, sevenary programs. Program boundedness is undecidable, even for linear, single* IDB*, binary programs. Boundedness is decidable for monadic programs, for chain programs and for ftd programs.*

For the case of a single IDB and a single recursive rule there has been some recent progress. This is slightly more general than sirups, because of the possibility of nonrecursive initialization rules. If the recursive rule is linear then it generalizes both [Ioa, Nau1]. Recently, undecidability has been claimed, by Abiteboul, if the recursive rule is nonlinear and decidability has been claimed, by Plambeck and Vardi, if the recursive rule is linear. Sirup uniform boundedness is still open; an NP-hardness lower bound is known [Kan2].

A definite decidability - undecidability boundary is emerging, even if many questions remain unresolved. The complexity bounds are still not tight for the monadic case. The effects of adding constants to programs are only partly understood, [Var11]. This is also true about the effects of equalities and inequalities in the rule bodies, e.g., monadic, single IDB boundedness is undecidable for Datalog$^{\neq}$ but uniform boundedness in this case is open, [GaiMSV]. Undecidability proofs todate use the fact that rules may be *disconnected*, see [CosGKV]; for the connected case there are only NP-hardness lower bounds, [Var11].

Open Problem 4.2.10: In which cases is boundedness decidable? In particular, what is its status for sirups and for connected programs?

Unbounded stage functions have also been analysed. Some Datalog queries are logspace-complete in PTIME. They are, in a worst case sense, inherently sequential and are not in QNC, unless NC = PTIME. If testing for linear order is not a primitive of the query language, then it is possible to derive unconditional lower bounds for these queries. For example, the following theorem follows from [Imm1].

Theorem 4.2.11: *Any Datalog$^{\neg}$ program H expressing "path system accessibility" has $\xi(n, H) = \Omega(\log^k n)$, for each fixed $k \geq 0$.*

On the other hand, queries in QNC have a lot of inherent parallelism. For example, linear programs express such queries; this is one reason why so much attention has been given to linear program evaluation. Determining if a program expresses a query in QNC is undecidable [UllVg, GaiMSV], but can be decided in special cases, e.g., for chain sirups [AfrP].

Sufficient conditions for detecting implicit parallelism have been proposed. A semantic condition on the size of derivation trees proposed in [UllVg] has the interesting property that (although it is not effectively testable) if it is true for a given program H then H can be transformed automatically into an equivalent H' which is naively bottom-up evaluated in $O(\log n)$ stages and is thus in QNC.

Theorem 4.2.12: *Let H be a program, such that every t in $H(d)$ has a polynomial size derivation tree. Then H can be transformed into an equivalent H', such that $\xi(n, H') = O(\log n)$.*

Additional Bibliographic Comments 4.2.13: Membership in NC, given the condition of Theorem 4.2.12, follows from [Ruz]. The more interesting consequence is that H can be transformed into H' such that $\xi(n, H') = O(\log n)$; this represents a dynamic version of the parallel algorithm technique proposed in [MilR]. Theorem 4.2.12 is extended in [Kan2].

An interesting application is the existence of a nonlinear chain program expressing a QNC query, that is not expressible by any linear chain program [UllVg] or by any linear program [AfrC].

It is possible to use the techniques of parallel computational complexity to analyse not only database logic programs, but logic programs in general. These programs contain f.o. terms (not only variables and constants) and the primitive operation for their evaluation is unification [Rob]. Term unification is in linear time [PatW], but is also logspace-complete in PTIME [DwoKM, Yas] and only restricted cases are known to be in NC [DwoKS]. □

4.2.III Query Evaluation and Selection Propagation

Up to this point we have outlined much of relational database theory without making use of constants and, in particular, of the selection operation $\varsigma_{A=a}$ from relational algebra with constants. We now examine such selections, in order to discuss one query optimization technique that is of great practical significance.

As we mentioned in Section 2.2.III "performing the selections before the joins" is a recurrent theme of relational algebra optimization. It is, in principle, always possible to perform *selection propagation* for a f.o. query. This is typically done by rewriting the parse tree of a relational expression and propagating the selections to its leaves. It is usually accompanied by a dynamic programming analysis of the order of join computations. Query evaluation is commonly performed in two phases: (1) a "compile-time" phase, when selection is propagated to the leaves and the order of joins is planned and (2) a "run-time" phase , when the database is accessed. Of course there is a whole spectrum of possibilities, within this "compile-time" and "run-time" classification.

Selection propagation into logic programs is one of the fundamental issues in the implementation of database logic programs, because of its impact on efficiency. In this more general setting the query language is Datalog$^{\neg}$ (usually Datalog) with constants, instead of the less expressive relational algebra with constants. The importance of this issue and the first partial solutions for database logic programs were proposed in [Ull2] and [HenN].

We refer to [BanR] for a survey and performance comparison of the various evaluation methods, which use selection propagation. As in the f.o. case, query evaluation can be divided into "compile-time" and "run-time" phases. However, many proposals have been made, where these phases are interleaved. It is also typical to use *seminaive* bottom-up evaluation instead of naive bottom-up evaluation, where in the seminaive case the Tarski construction is performed in such a way that the work at iteration i is not duplicated by the work at iteration $i+1$.

We illustrate the formal setting of Datalog with constants using a simple example; syntax and semantics are intuitive generalizations of the constant free case.

Example 4.2.14: Consider the program with constant a,

$R_1(x) :\!-\; R_2(ax).$

$R_2(xy) :\!-\; R_2(xz), R_3(zy).$

$R_2(xy) :\!-\; R_3(xy).$

The input is a directed graph represented by a relation over R_3, and the output (over R_1) is the set of nodes x, such that $< ax >$ is in the transitive closure of the input. A bottom-up evaluation of this program corresponds to a computation of all pairs reachability and then a restriction to the nodes reachable from a. The following program is equivalent (R_2 is no longer needed) but its bottom-up evaluation corresponds to a single source reachability computation from source a.

$R_1(x) :\!-\; R_1(z), R_3(zx).$

$R_1(x) :\!-\; R_3(ax).$ □

Naive or seminaive bottom-up evaluation are ways of constructing the output to a query, that do not take advantage of selections until the minimum model has been constructed. On the contrary, the typical Prolog interpreter would produce the output in "top-down" fashion by reasoning backwards from the selection. Between these two extremes, there is a whole spectrum of evaluation stategies. These are based on a tradeoff between "bottom-up" and "top-down" evaluation. In Example 4.2.14: the second program is produced from the first program via "top-down" reasoning about their equivalence, then it is evaluated via a "bottom-up" method.

As illustrated in [BanR] many other possibilities exist, depending on which optimizations are applied and on whether they are performed at "compile-time" or "run-time." Many of the algorithms are expressed in algebraic terms, i.e., as relational algebra expressions together with *while* constructs. In some cases, arithmetic operations are used to control the number of iterations.

The first sufficient conditions for commuting recursion, in the form of a least fixpoint operator, and selection were introduced by Aho and Ullman in [AhoU]. This approach does not consider other possibilities of equivalence transformations on the program, beyond a straightforward rewriting of the selection operation into the recursion. The magic set strategy of [BanMSU] adopts a more general view of selection propagations. Assume that a relation is defined via a Datalog program and that a selection is applied to this relation; (in Prolog terminology a relation is defined using a Prolog program without function symbols and a variable in the goal is bound to a constant). The magic set strategy propagates the information about the selection by computing *magic* sets of values, which are then used to prune useless rule applications in a "bottom-up" method. The magic sets themselves are computed using rules that are less costly than the original rules.

The following example illustrates the magic set strategy, as a program optimization applied at "compile-time." The "same generation" program is translated into a different, but equivalent, program. This is accomplished by adding atoms to the bodies of already existing rules (typically with binary IDB's). These atoms are of the form $magic(x)$ and are computed using less costly rules, (typically with monadic IDB's). For the details of the transformation we refer to [BanMSU] and for an interpretation of it using quotients of context free languages to [BeeKBR].

Example 4.2.15: Consider the following version of the "same generation" program,

$R_1(x) :\!\!— R_2(ax).$

$R_2(xy) :\!\!— R_3(xz), R_4(zy).$

$R_2(xy) :\!\!— R_3(xz), R_2(zz'), R_4(z'y).$

Think of $R_3(xz)$ as x is-child-of z and of $R_4(z'y)$ as z' is-parent-of y. The $R_2(xy)$ stands for x belongs in same-generation-as y and R_1 contains all those x in a's generation. The magic set transformation produces an equivalent program, where the new monadic IDB MAGIC restricts the application of the old rules to whenever x is instantiated as an ancestor of a.

$R_1(x) :\!\!— R_2(ax).$

$R_2(xy) :\!\!— R_3(xz), R_4(zy), magic(x).$

$R_2(xy) :\!\!— R_3(xz), R_2(zz'), R_4(z'y), magic(x).$

$magic(a) :\!\!—$

$magic(x) :\!\!— magic(y), R_3(yx).$ □

Evaluation based on counting path lengths of ground tuples has been proposed in [BanMSU] and is refined in [SacZ], where it is combined with magic sets. One restriction is that input relations must be acyclic directed graphs. This use of *counting* is a departure from the pure Datalog context, but it is justified by the considerable gain in performance of *magic counting* [SacZ], [MarPS]. Even specific programs, such as the program of Example 4.2.15, are challenging computational questions when the evaluation is allowed to use counters [HadN] and the inputs are arbitrary relations with cycles.

An important issue in this area is the development of a framework in which the plethora of evaluation methods can be analysed and compared. The *capture rule* framework of [Ull2] allows the combination of these methods, provided they have a minimum degree of modularity. A systematic treatment of the methods is attempted in [BeeRa].

The propagation of selections is formalized in [BeeKBR] as the problem of: *finding an equivalent program of smaller equation width.* This formulation allows boundedness to be used as an analytical technique. It also indicates some fundamental differences from selection propagation for f.o. queries.

For *chain* Datalog program H, selection propagation corresponds to a regularity condition [BeeKBR] on the language $L(H)$. This $L(H)$ is generated from the rules of H, without the variables, as context free grammar productions taking the EDB's as terminals and the IDB's as nonterminals. If the regularity condition holds, as in the program of Example 4.2.14, then it is possible to decrease the equation width. If the regularity condition does not hold, as for example in the program of Example 4.2.15, it is not possible to decrease the equation width, even if some heuristic improvements are possible. The analysis of [Nau2] may be viewed as defining regularity conditions for *nonchain* programs. The regularity condition of [Nau2] allows for a clean comparison between a variety of evaluation methods.

"Top-down" evaluation strategies are closer to the spirit of Prolog interpreters. Their analysis presents many nontrivial computational questions, even when the only function symbols are constants. The simplest question here is termination, which is trivially true for "bottom-up" methods. The price to pay for the clever "top-down" propagation of selection information is the possibility of nontermination for these methods. In some cases it is possible to analyse the program and decide termination [AfrEtal, Mor, UllVa].

Much of the query evaluation literature is concerned with linear database logic programs. One reason for this is that, queries defined this way are in QNC and consequently have large amount of potential parallelism. An important question is how to efficiently use many processors to evaluate such queries in parallel. The core combinatorial problem is: *source-sink reachability in directed graphs* [Kan2, Ull3]. This problem is known to be in NC. For undirected graphs, a linear number of processors suffice in order to achieve $O(\log^2 n)$ parallel time on most parallel computation models. (This makes the processor-time product $O(n \log^2 n)$, which is close to the $O(n)$ product for the one processor case). For directed graphs, the best processor-parallel time product for NC algorithms is $O(n^k)$, identical with the sequential time bounds for n by n matrix multiplication. (There is a increase from the $O(n)$ product for the one processor case).

Open Question 4.2.16: Are there processor efficient NC algorithms for source-sink directed reachability?

Additional Bibliographic Comments 4.2.17: For some new, interesting, general program transformations we refer to [RamSUV, Sar].

We refer the reader to [BanR] for an extensive bibliography of evaluation methods. Here we only present a sample from a currently active area of research. It is important to stress that many of the evaluation methods form integral parts of new logic database systems, e.g., [MorUV, BeeNRST, TsuZ].

A question related to the termination of various evaluation methods is that of safety for database logic programs. This is analogous to safety for relational calculus, only now fixpoints add a new dimension to the problem, [KifRS, SagV]. □

4.3 Query Languages and Complex Object Data Models

An important aspect of the relational data model and of its logic programming extensions is their simplicity. Relations are the only data-type. They are the common denominator, which facilitates the comparison and the analysis of many query formalisms. Semantics are introduced using dependencies, which are (as we have seen) closely related to queries. Despite the elegance of the theory, relations are flat structures and the representation of hierarchical database structures via relations is sometimes forced. It is often simpler to deal explicitly with more complex data-types, than to express them indirectly using dependencies.

There has been wide interest in hierarchical database structures. Of particular interest are those built using finite *tuple* and *set* constructors. These constructors are the basis for defining a large repertoire of data-types. Instances of these data-types are manipulated using query languages that are extensions of relational algebra and calculus.

The first proposal to generalize the relational data model by removing the 1-NF assumption was made by Makinouchi [Mak]. Algebras have been proposed for nested (i.e., non 1-NF) relations, e.g., [AbiBe, AbiBi, JaeS, KupV1, RotKS, SchS, ThoF]. These are usually based on the two operations NEST and UNNEST introduced by [JaeS]. A early higher order calculus (but no algebra) is contained in [Jac]. First order calculi can be found in [Abibe, KupV1, RotKS]. A good survey of the area is [Hul2].

The expressive power of relational algebra can be significantly extended by the addition of a *power set operator* (first considered in [KupV1]). It is shown in [AbiBe] that transitive closure can be expressed by an algebra extended with the powerset. It is interesting to consider looping constructs, least fixpoint constructs etc in the context of nested relation algebras, see [GysV]. Most of these enrichments of nested relation algebras turn out to have the same expressive power as the algebra with the powerset operator in [AbiBe]. As shown in [ParV], if the inputs and outputs are 1-NF relations, if there is no powerset operation or looping construct etc, and if the intermediate results are nested relations then, the expressive power of many of these formalisms is still that of relational algebra.

Algebra = calculus theorems are shown in both [KupV1] and [AbiBe]. The calculus of [KupV1] has many similarities with recursive rules, whereas the one in [AbiBe] is a more intuitive extension of the relational calculus. The two settings are somewhat different in terms of their data-types: both have *set,tuple and union types* and [KupV1] also allows *recursive types* (but only for inputs). Also, *object identifiers* are present in [KupV1]; these can be thought of as pointers to values. Both settings are generalized in [AbiK].

Another basic idea, that has been explored recently, is to use levels of nesting of the powerset to classify queries. In this way, queries can be classified into hierarchies going beyond PHIER-queries. See [HulS, KupV2] for the detailed formulation and results.

Much recent attention has been devoted to complex object models, with logic-based query languages, e.g., [AbiBe, BanK, BunDW, KupV1]. In this context the *tuple* constructor leads to first order terms, which can be manipulated using term unification or matching (see Comments 4.2.13). This is a significant step towards full integration of databases and logic programming. On the other hand, adding the *set* constructor leads to nontrivial extensions of conventional logic programming.

Incorporating *sets* into database logic programs is a promising research direction [AbiGr, BanK, BeeNRST, Kup1, Kup2, ShmTZ]. The solution proposed in [ShmTZ] makes use of the difference between data and expression complexity. For a comparison of the various proposed solutions we refer to [Kup2].

An important distinction that can be made in complex object languages is whether the language can be *statically type checked* or not. For example, this is possible in [AbiBe, AbiGr, KupV1, Kup2], but not in [BanK]. The key is the separation of scheme and instance, which is present in database languages but not in logic programming languages like Prolog.

Interestingly, it is possible to have a simple generalization of most complex object models with a language that: (1) has set, tuple, union, and recursive types for both input and output, (2) can be statically type checked, and (3) has full computational capabilities. An example is the language IQL of [AbiK], which is based on recursive rules and object identifiers.

Complex object data models, semantic data models [HulK], and the more recent *object-oriented data models* are all efforts to circumvent the semantic limitations of relations. A variety of new applications require languages, that are much closer to full programming languages than are the current database query languages, [Ban2]. The most challenging open question in this area is the development of an elegant, mathematical model to serve as a foundation for the object-oriented database experimental efforts.

5 Discussion: Other Issues in Relational Database Theory

There are database problems orthogonal to the issues addressed in the theories of dependencies and queries. We close our presentation with two such issues: *incomplete information* and *database updates*.

These are important in a wider context. A theory of incomplete information should be a theory of "knowledge", with many formal connections to modal logics. Updates are an example of dealing with "dynamic change", a problem that has motivated much of the research in the area of the logics of programs. Here, we limit ourselves to formulations and solutions that have a distinctly database character.

5.1 The Problems of Incomplete Information

The implicit assumption of the relational data model is that the database is a completely determined finite structure. Reiter's work on first order theories as databases makes explicit the implicit assumptions about the relational data model, [Rei1, Rei2, Rei3].

To deal with incomplete information we need representations of sets of "possible worlds", i.e., the identification we have been making of a database with a single "possible world" must be relaxed. Consequently, the syntax itself of the database must be used as a specification. This was done in Section 2.5 with extended relational theories for the complete information case, with deductive and universal relation models, and with relaxing the uniqueness axioms (Remark 2.5.1).

In terms of what is a "possible world", there is a fundamental distinction between *open* and the *closed* extensions of relations. Open extensions are best exemplified by the weak universal relation assumption, i.e., we know some tuples completely and any containing universal relation is a "possible world". A closed extension is a *conservative* open extension, i.e., it is an open extension that consists only of tuples whose existence we can infer. A detailed comparison of the two approaches, based on computational complexity, is made in [Var10].

The first order logical framework can be augmented to express facts about the uncertainty itself. This has been done through the addition of *modal* operators, e.g., for possibility or certainty, see [Lev, Lip1, Lip2].

For the rest of this section, we focus on various algebraic mechanisms for modeling uncertainty. The most commonly used one is the mechanism of *null values.* Their presence changes the basic data type of relation as follows.

A *Codd table* ([Cod4]) is the result of replacing some occurrences of values in a relation by *distinct* variables, called *null values.* A *condition* φ is a propositional $(\vee, \wedge, \neg)$ combination of equality atoms $x = y$ or $x = c$, where x, y are variables and c is a value. Conditions may be associated with Codd table T in two ways: (1) a *global condition* φ_T is associated with the whole of T, and (2) a *local condition* φ_t is associated with a tuple t of T, (if no association is specified the default is the trivial true condition $x = x$). A *conditional table* $\mathcal{T}$ is a Codd table T with associated global and local conditions, see [ImiL2]. One can visualize a conditional table as the rows of an untyped tableau with constants, together with a column of conditions.

A valuation h is a function from variables to values, that naturally extends to tuples and Codd tables by defining $h(c) = c$, for each value c. A condition is satisfied or falsified by h in the obvious way. A conditional table $\mathcal{T}$ (with Codd table T) represents the set of relations:

$\mathcal{T}_{rel} = \{\ r :$ there is a $h \models \varphi_T$ and r consists of the tuples $h(t)$ for which $h \models \varphi_t\ \}$.

Why are conditional tables an interesting representation? Codd tables are a simple way of generalizing relations. Between them and conditional tables there is a large repertoire of representations, [AbiKG]. For example, the first order theories in [Var9] can be represented using a Codd table and a conjunctive global condition. In [Grh] global conditions are used to describe dependency satisfaction. The chase may be applied to conditional tables [Grh, Mai1]. The most important reason is that: there is a closure theorem for conditional tables, which allows simple and semantically meaningful querying of incomplete information databases. (One has to expand Codd tables with local conditions in order to realize this closure, [ImiL2, AbiG]). According to these semantics, the output is a representation of *all the possibilities* in the following fashion.

If $\mathcal{T}$ is a conditional table (without global condition) and E a relational algebra expression, then from E and $\mathcal{T}$ it is possible to construct another $\mathcal{T}'$ conditional table (without global condition) such that: $\mathcal{T}'_{rel} = E(\mathcal{T}_{rel}) = \{E(r) : r \in \mathcal{T}_{rel}\}$.

There is a large volume of work on querying incomplete information databases. The goal of most of this research has been to determine the correct semantics of applying a program to an incomplete information database. Closure conditions are one desirable alternative. They often lead to variants of the relational operations with which to process null values. For example, one common problem is how to extend the $\bowtie$ operation to relations with null values. We refer to [ImiL2] for many more references on the subject.

Many of the proposed algorithms focus on *the certain* tuples, i.e., tuples contained in all "possible worlds", as opposed to describing all possibilities. We encountered this *conservative* approach, when we examined querying in deductive and universal relation data models. When querying for the certain tuples: an algorithm is *sound* if it produces certain tuples only and it is *complete* if it produces all the certain tuples.

Unfortunately, it is often the case that completeness implies an exponential increase in data complexity. This was was first shown in [Var9] and further investigated in [AbiKG]. Thus, the sound algorithms in [GrnM, Rei3, Var9] trade completeness for efficiency. There is one important special case, where things are ideal. This was identified in [Var9] and in [ImiL2] for positive existential queries.

To derive the certain answers to a second order positive query on a Codd table with a satisfiable conjunctive global condition it suffices to: (1) *incorporate the equality atoms in the table*, (2) *ignore the inequality atoms*, and (3) *proceed as if the table were a relation.*

The null values we have been discussing are "values present but unknown", sometimes constrained through conditions. Other kinds of null values, e.g., "values whose presence is unknown", have been studied in [Zan2].

Many ideas from mathematical logic have been proposed as modeling tools for uncertainty (e.g., three-valued logic) but their applicability is hard to assess. In summary, for the representation and querying of incomplete information databases there are many partial solutions but no satisfactory full answer. In part this is because, the further away we move from the relational data model the fewer analytical and algebraic tools are available and the more we have to rely on general-purpose theorem-proving heuristic techniques.

5.2 The Problems of Database Updates

Database manipulation languages have primitives for both querying and updating relations. The characteristic that distinguishes database updating from updating the state in more general programming languages is the simplicity of the changes made. Tuples are only *inserted, deleted* or values *modified* in the one underlying relation data-type. It is possible to study these operations in the context of both algebra-like and calculus-like query languages. These investigations span language definition [AbiV1, AbiV2], program optimization [AbiV3], dependencies [Via] and incomplete information [AbiG].

A first order theory approach to updates was initiated in [FagUV] and elaborated in [FagKUV]. The semantics of *insert, delete* for first order theories is formulated in terms of sets of these theories. The broader logical framework and novel notions, such as equivalence that is preserved under common updates or *equivalence forever*, illustrate new challenges to database theory. Algorithmic problems in this setting are resolved in [Win1]; see [Win2] for an overview.

Although broader than the practical use of updates in many database systems, the above concepts are more specialized than a direct use of some form of dynamic logic [Har]. For two recent applications of dynamic logic to database logic programs see [ManW, NaqK].

We close our exposition with the *view update problem.* This is a problem that combines incomplete information with update issues. It is a challenging problem in general, which has been elegantly formalized in relational database theory.

View Updates: Individual users often want to deal only with part of the information of the database. This is why database systems provide the *view* facility. The user would like to query and update the database view in blissful ignorance of the underlying database.

In database systems, a view is generally implemented by storing its definition, i.e., some program in a query language supported by the system. Thus, querying is done by composing the query on the view with the view definition. Sometimes things are simplified further by precomputing and storing the view as a relation, called *materializing* the view. Updating a view is much more challenging. A tuple insertion in or deletion from the view may make the underlying database inconsistent ("no possible world") or ambiguous ("many possible worlds").

A first attempt at resolving these ambiguities was made in [DayB]. An elegant semantic solution was proposed in [BanS], which is quite independent of the relational data model and stated in terms of mappings.

$$\begin{tikzcd}
d \arrow[r, "\bar{f}_u"] \arrow[d, "f_{v\times v'}"'] & d' \\
< f_v(d), f_{v'}(d) > \arrow[r, "f_u"'] & < f_{uv}(d), f_{v'}(d) > \arrow[u, "f^{-1}_{v\times v'}"']
\end{tikzcd}$$

Figure 6: View update semantics

A *view* f_v and an *update* on this view f_u are both mappings of the appropriate types (from the database scheme to the view scheme and from the view scheme to itself respectively). How can f_u be translated into a database update $\bar{f}_u$ (of type from the database scheme to itself)? The basic insight of [BanS] here is that of a *complementary view* $f_{v'}$ of f_v, where the mapping from databases d to databases $< f_v(d), f_{v'}(d) >$ must be injective. This mapping is denoted by $f_{v\times v'}$.

A view can have many complementary views; choosing one that must remain constant assigns unambiguous semantics to a view update.

This is because the database can be reconstructed from the updated view and the constant complement; (recall the theory of decompositions from Section 3.2.I). Once a complementary view is chosen, translating under this constant complement amounts to finding a database d' such that: (1) $f_v(d') = f_u(f_v(d)) = f_{uv}(d)$ and (2) $f_{v'}(d') = f_{v'}(d)$. If such a d' exists then by injectiveness it is unique. The update mapping f_u is translatable iff such a d' exists for any d; (the independent two-part decompositions from Section 3.2.I are a sufficient condition for translatability). If f_u is translatable then its translation is $\bar{f}_u = f^{-1}_{v\times v'} \circ f_{uv\times v'}$.

[BanS] and [CosP] describe additional properties of this approach, in particular for families of updates closed under composition. The semantics of [BanS] are depicted pictorially in the commuting diagram of Figure 6. They are a clarifying (even if initial) step in resolving a challenging problem. Their complexity, for the simplest possible set of dependencies (fd's) and views (projections), is examined in [CosP]. In many cases the computational complexity of a full implementation of view updates is prohibitive. Much of the recent work, e.g., [GotPZ, Kel, KelU, Win1, Win2], has concentrated on feasible algorithmic approximations.

6 Conclusions

Queries and dependencies are major themes, which we have used to present relational database theory. A number of problems remain unresolved. Among the crisp and perhaps hard mathematical problems that have emerged we have highlighted: dependency implication (2.3.16), recursive rule boundedness (4.2.10) and parallel computation (4.2.16) questions.

In the area of logic and databases, we are only beginning to understand the possible uses of negation and the large variety of evaluation methods. Much remains to be done on the subjects of:

complex objects, incomplete information, and database updates. Finally, developing the foundations for object-oriented database languages will (most likely) involve new conceptual contributions, that use the theory of abstract data-types.

Acknowledgements: We would like to thank the many researchers who provided helpful, detailed comments to earlier versions of this paper. In particular, we would like to thank: Serge Abiteboul, Ed Chan, Claude Delobel, Ron Fagin, Marc Gyssens, Michael Kifer, Gabi Kuper, Jack Minker, Jeff Naughton, Jan Paredaens, Oded Shmueli, Dirk Tahon, Allen VanGelder, Dirk VanGucht, Victor Vianu, and Mihalis Yannakakis. We would also like to acknowledge: Ashok Chandra, Stavros Cosmadakis, Phokion Kolaitis, and Moshe Vardi, for many clarifying discussions on the fine points of the theory of queries. Support for the writing of this chapter was provided partly by NSF grant IRI-8617344 and partly by an Alfred P. Sloan Foundation Fellowship.

7 References

[AbiBe] S. ABITEBOUL, C. BEERI, *On the Manipulation of Complex Objects*, Proc. Intl. Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt, (1987).

[AbiBi] S. ABITEBOUL, N. BIDOIT, *Non First Normal Form Relations to Represent Hierarchically Organized Data*, Proc. 3rd ACM PODS, (1984), pp. 191–200.

[AbiG] S. ABITEBOUL, G. GRAHNE, *Update Semantics for Incomplete Databases*, Proc. 11th Conf. on Very Large Databases, (1985), pp. 1–12.

[AbiGr] S. ABITEBOUL, S. GRUMBACH, *COL: A Logic Based Language for Complex Objects*, Proc. 1rst Intl. Conf. on Extending Database Technology, (1988), pp. 271–293.

[AbiK] S. ABITEBOUL, P.C. KANELLAKIS, *Object Identity as a Query Language Primitive*, Proc. ACM SIGMOD, (1989), to appear.

[AbiKG] S. ABITEBOUL, P.C. KANELLAKIS, G. GRAHNE, *On the Representation and Querying of Sets of Possible Worlds*, Proc. ACM SIGMOD, (1987), pp. 10–19.

[AbiV1] S. ABITEBOUL, V. VIANU, *A Transaction Language Complete for Database Update and Specification*, Proc. 6th ACM PODS, (1987), pp. 260–268.

[AbiV2] S. ABITEBOUL, V. VIANU, *Procedural and Declarative Database Update Languages*, Proc. 7th ACM PODS, (1988), pp. 240–250.

[AbiV3] S. ABITEBOUL, V. VIANU, *Equivalence and Optimization of Relational Transactions*, JACM 35, 1, (1988), pp. 130–145.

[AfrC] F. AFRATI, S.S. COSMADAKIS, *Expressiveness of Restricted Recursive Queries*, Proc. 21rst ACM STOC, (1989), to appear.

[AfrP] F. AFRATI, C.H. PAPADIMITRIOU, *The Parallel Complexity of Simple Chain Queries*, Proc. 6th ACM PODS, (1987), pp. 210–213.

[AfrEtal] F. AFRATI, C.H. PAPADIMITRIOU, G. PAPAGEORGIOU, A. ROUSSOU, Y. SAGIV, J.D. ULLMAN, *Convergence of Sideways Query Evaluation*, Proc. 5th ACM PODS, (1986), pp. 24–30.

[AhoBU] A.V. AHO, C. BEERI, J.D. ULLMAN, *The Theory of Joins in Relational Databases*, ACM TODS 4, 3, (1979), pp. 297–314.

[AhoSU1] A.V. AHO, Y. SAGIV, J.D. ULLMAN, *Equivalence of Relational Expressions*, SIAM J. Computing 8, 2, (1979), pp. 218–246.

[AhoSU2] A.V. AHO, Y. SAGIV, J.D. ULLMAN, *Efficient Optimization of a Class of Relational Expressions*, ACM TODS 4, 4, (1979), pp. 435–454.

[AhoU] A.V. Aho, J.D. Ullman, *Universality of Data Retrieval Languages*, Proc. 6th ACM POPL, (1979), pp. 110–117.

[AjtG] M. Ajtai, Y. Gurevich, *Monotone versus Positive*, JACM 34, 4, (1987), pp. 1004–1015.

[Apt] K. Apt, *An Introduction to Logic Programming*, This handbook.

[AptBW] K. Apt, H. Blair, A. Walker, *Towards a Theory of Declarative Knowledge*, Foundations of Deductive Databases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 89–148.

[AptV] K. Apt, M. Van Emden, *Contributions to the Theory of Logic Programming*, JACM 29, 3, (1982), pp. 841–862.

[Arm] W.W. Armstrong, *Dependency Structures of Database Relationships*, Proc. IFIP, North Holland, (1974), pp. 580–583.

[AroC] A.K. Arora, C.R. Carlson, *The Information Preserving Properties of Relational Data base Transformations*, Proc. 4th Conf. on Very Large Databases, (1978), pp. 352–359.

[AstEtal] M.M. Astrahan, etal., *System R: a Relational Approach to Data Management*, ACM TODS 1, 2, (1976), pp. 97–137.

[AtzC] P. Atzeni, E.P.F. Chan, *Efficient Query Answering in the Representative Instance Approach*, Proc. 4th ACM PODS, (1985), pp. 181–188.

[AtzD] P. Atzeni, M.C. De Bernardis, *A New Basis for the Weak Instance Model*, Proc. 6th ACM PODS, (1987), pp. 79–86.

[AusDM] G. Ausiello, A. D'Atri, M. Moscarini, *Chordality Properties on Graphs and Minimal Conceptual Connections in Semantic Data Models*, Proc. 4th ACM PODS, (1985), pp. 164–170.

[AylGSS] A.K. Aylamazyan, M.M. Gigula, A.P. Stolbouskin, G.F. Schwartz, *Reduction of the Relational Model with Infinite Domain to the Case of Finite Domains*, Doklady Akad. Nauk. SSSR 286, 2, (1986), pp. 308–311.

[Ban1] F. Bancilhon, *On the Completeness of Query Languages for Relational Data Bases*, Proc. 7th Symposium on the Mathematical Foundations of Computer Science, Springer-Verlag LNCS 64, (1978).

[Ban2] F. Bancilhon, *Object-Oriented Database Systems*, Proc. 7th ACM PODS, (1988), pp. 152–162.

[BanK] F. Bancilhon, S. Khoshafian, *A Calculus for Complex Objects*, Proc. 5th ACM PODS, (1986), pp. 53–59.

[BanMSU] F. Bancilhon, D. Maier, Y. Sagiv, J.D. Ullman, *Magic Sets and Other Strange Ways to Implement Logic Programs*, Proc. 5th ACM PODS, (1986), pp. 1–15.

[BanR] F. Bancilhon, R. Ramakrishnan, *An Amateur's Introduction to Recursive Query Processing*, Proc. ACM SIGMOD, (1986), pp. 16–52. See also, Foundations of Deductive Databases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 439–518.

[BanS] F. Bancilhon, N. Spyratos, *Update Semantics of Relational Views*, ACM TODS 6, 4, (1981), pp. 557–575.

[Bee] C. Beeri, *On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases*, ACM TODS 5, 3, (1980), pp. 241–259.

[BeeB] C. Beeri, P.A. Bernstein, *Computational Problems Related to the Design of Normal Form Relational Schemas*, ACM TODS 4, 1, (1979), pp. 30–59.

[BeeBG] C. Beeri, P.A. Bernstein, N. Goodman, *A Sophisticate's Introduction to Database*

[Hea] I.J. HEATH, *Unacceptable File Operations in a Relational Data Base*, Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control (1971).

[HenMT] L. HENKIN, J.D. MONK, A. TARSKI, *Cylindric Algebras*, Part I, North Holland, (1971); Part II, North Holland, (1985).

[HenN] L.J. HENSCHEN, S.A. NAQVI, *On Compiling Queries in Recursive First-Order Databases*, JACM 31, 1, (1984), pp. 47–85.

[Hon] P. HONEYMAN, *Testing Satisfaction of Functional Dependencies*, JACM 29, 3, (1982), pp. 668–677.

[HonLY] P. HONEYMAN, R.E. LADNER, M. YANNAKAKIS, *Testing the Universal Instance Assumption*, Inf. Proc. Letters 10, 1 (1980), pp. 14–19.

[Hul1] R. HULL, *Finitely Specifiable Implicational Dependency Families*, JACM 31, 2, (1984), pp. 210–226.

[Hul2] R. HULL, *A Survey of Theoretic Research on Typed Complex Database Objects*, Databases, (J. Paredaens ed.), Academic Press, (1987), pp. 193–256.

[HulK] R. HULL, R. KING, *Semantic Database Modeling: Survey, Applications, and Research Issues*, ACM Computing Surveys, (to appear).

[HulS] R. HULL, J. SU, *On the Expressive Power of Database Queries with Intermediate Types*, Proc. 7th ACM PODS, (1988), pp. 39–51.

[ImiL1] T. IMIELINSKI, W. LIPSKI, *The Relational Model of Data and Cylindric Algebras*, JCSS 28, 1, (1984), pp. 80–102.

[ImiL2] T. IMIELINSKI, W. LIPSKI, *Incomplete Information in Relational Databases*, JACM 31, 4 (1984), pp. 761–791.

[Imm1] N. IMMERMAN, *Number of Quantifiers is Better than Number of Tape Cells*, JCSS 22, 3, (1981), pp. 65–72.

[Imm2] N. IMMERMAN, *Relational Queries Computable in Polynomial Time*, Inf. and Control 68, (1986), pp. 86–104.

[Imm3] N. IMMERMAN, *Languages Which Capture Complexity Classes*, SIAM J. Computing 16, 4, (1987), pp. 760–778.

[Imm4] N. IMMERMAN, *Expressibility as a Complexity Measure: Results and Directions*, Yale Univ. Res. Rep. DCS-TR-538, (1987).

[Ioa] Y.E. IOANNIDIS, *A Time Bound on the Materialization of Some Recursively Defined Views*, Proc. 11th Conf. on Very Large Databases, (1985), pp. 219–226.

[ItoIK] M. ITO, M. IWASAKI, T. KASAMI, *Some Results on the Representative Instance in Relational Databases*, SIAM J. Computing 14, 2, (1985), pp. 334–354.

[Jac] B.E. JACOBS, *On Database Logic*, JACM 29, 2 (1982), pp. 310–332.

[JaeS] G. JAESCHKE, H.-J SCHEK, *Remarks on the Algebra on Non First Normal Form Relations*, Proc. 1st ACM PODS, (1982), pp. 124–138.

[JarK] M. JARKE, J KOCH, *Query Optimization in Database Systems*, ACM Computing Surveys 16, 2, (1984), pp. 111–152.

[JohK] D.S. JOHNSON, A. KLUG, *Testing Containment of Conjunctive Queries Under Functional and Inclusion Dependencies*, JCSS 28, 1 (1984), pp. 167–189.

[JonS] N.G. JONES, A.L. SELMAN, *Turing Machines and the Spectra of First-Order Sentences*, J. Symbolic Logic 39, (1974), pp. 139–150.

[Kan1] P. C. KANELLAKIS, *On the Computational Complexity of Cardinality Constraints in Relational Databases*, Inf. Proc. Letters 11, 2, (1980), pp. 98–101.

[BunDW] P. BUNEMAN, S. DAVIDSON, A. WATTERS *A Semantics for Complex Objects and Approximate Queries*, Proc. 7th ACM PODS, (1988), pp. 302–314.

[CasFP] M.A. CASANOVA, R. FAGIN, C.H. PAPADIMITRIOU, *Inclusion Dependencies and Their Interaction with Functional Dependencies*, JCSS 28, 1, (1984), pp. 29–59.

[CasV] V. CASANOVA, V.M.P. VIDAL, *Towards a Sound View Integration Methodology*, Proc. 2nd ACM PODS, (1983), pp. 36–47.

[Cha1] A.K. CHANDRA, *Programming Primitives for Database Languages*, Proc. 8th ACM POPL, (1981), pp. 50–62.

[Cha2] A.K. CHANDRA, *Theory of Database Queries*, Proc. 7th ACM PODS, (1988), pp. 1–9.

[ChaH1] A.K. CHANDRA, D. HAREL, *Computable Queries for Relational Data Bases*, JCSS 21, 2, (1980), pp. 156–178.

[ChaH2] A.K. CHANDRA, D. HAREL, *Structure and Complexity of Relational Queries*, JCSS 25, 1, (1982), pp. 99–128.

[ChaH3] A.K. CHANDRA, D. HAREL, *Horn Clause Queries and Generalizations*, J. Logic Programming 2, 1, (1985), pp. 1–15.

[ChaLM] A.K. CHANDRA, H.R. LEWIS, J.A. MAKOWSKY, *Embedded Implicational Dependencies and Their Inference Problem*, Proc. 13th ACM STOC, (1981), pp. 342–352.

[ChaM] A.K. CHANDRA, P.M. MERLIN, *Optimal Implementation on Conjunctive Queries in Relational Data Bases*, Proc. 9th ACM STOC, (1977), pp. 77–90.

[ChaV] A.K. CHANDRA, M.Y. VARDI, *The Implication Problem for Functional and Inclusion Dependencies is Undecidable*, SIAM J. Computing 14, 3, (1985), pp. 671–677.

[ChnH] E.P. CHAN, H.J. HERNANDEZ, *Independence Reducible Database Schemes*, Proc. 7th ACM PODS, (1988), pp. 163–173.

[ChnM] E.P. CHAN, A.O. MENDELZON, *Answering Queries on the Embedded Complete Database Schemes*, JACM 34, 2, (1987), pp. 349–375.

[Chi] D.L. CHILDS, *Feasibility of a Set-theoretical Data Structure - A General Stucture Based on a Reconstituted Definition of Relation*, Proc. IFIP, North Holland, (1968), pp. 162–172.

[ChiT] L.H. CHIN, A. TARSKI, *Remarks on Projective Algebras*, abstract, Bulletin AMS, 54 (1948), pp. 80–81.

[Cla] K.L. CLARK, *Negation as Failure Logic and Databases*, Logic and Databases, (H. Gallaire, J. Minker, eds.), Plenum, (1978), pp. 293–322.

[Cod1] E.F. CODD, *A Relational Model of Data for Large Shared Data Banks*, CACM 13, 6, (1970), pp. 377–387.

[Cod2] E.F. CODD, *Further Normalization of the Data Base Relational Model*, Data Base Systems, (R. Rustin, ed.), Prentice-Hall, (1972), pp. 33–64.

[Cod3] E.F. CODD, *Relational Completeness of Database Sublanguages*, Data Base Systems, (R. Rustin, ed.), Prentice-Hall, (1972), pp. 65–98.

[Cod4] E.F. CODD, *Extending the Data Base Relational Model to Capture More Meaning*, ACM TODS 4, 4, (1979), pp. 397–434.

[Cod5] E.F. CODD, *Relational Databases: a Practical Foundation for Productivity*, CACM 25, 2, (1982), pp. 102–117.

[Coo] S.A. COOK, *A Taxonomy of Problems with Fast Parallel Algorithms*, Inf. and Control 64, (1985), pp. 2–22.

[Cos1] S.S. COSMADAKIS, *The Complexity of Evaluating Relational Queries*, Inf. and Control 58, (1983), pp. 101–112.

[Cos2] S.S. COSMADAKIS, *Equational Theories and Database Constraints*, Ph.D. Dissertation, MIT Res. Rep. TR-346, (1985).

[Cos3] S.S. COSMADAKIS, *Database Theory and Cylindric Lattices*, Proc. 27th IEEE FOCS, (1987), pp. 411–420.

[Cos4] S.S. COSMADAKIS, *On the First Order Expressibility of Recursive Queries*, Proc. 8th ACM PODS, (1989), pp. 311–324.

[CosGKV] S.S. COSMADAKIS, H. GAIFMAN, P.C. KANELLAKIS, M.Y. VARDI, *Decidable Optimization Problems for Database Logic Programs*, Proc. 20th ACM STOC, (1988).

[CosK1] S.S. COSMADAKIS, P.C. KANELLAKIS, *Equational Theories and Database Constraints*, Proc. 17th ACM STOC, (1985), pp.273–284.

[CosK2] S.S. COSMADAKIS, P.C. KANELLAKIS, *Functional and Inclusion Dependencies: a Graph Theoretic Approach*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 164–185.

[CosK3] S.S. COSMADAKIS, P.C. KANELLAKIS, *Parallel Evaluation of Recursive Rule Queries*, Proc. 5th ACM PODS, (1986), pp. 280-293.

[CosKS] S.S. COSMADAKIS, P.C. KANELLAKIS, S. SPYRATOS, *Partition Semantics for Relations*, JCSS 32, 2, (1986), pp. 203–233.

[CosP] S.S. COSMADAKIS, C.H. PAPADIMITRIOU, *Updates of Relational Views*, JACM 31, 4, (1984), pp. 742–760.

[Dah] E. DAHLHAUS, *Skolem Normal Forms Concerning the Least Fixpoint*, Computation Theory and Logic, (E. Börger, ed.), Springer-Verlag LNCS 270, (1987), pp. 101–106.

[DahM] E. DAHLHAUS, J.A. MAKOWSKY, *Computable Directory Queries*, Proc. 11th CAAP 86, Springer-Verlag LNCS 214, pp. 254–265.

[DatM] A. D'ATRI, M. MOSCARINI, *Recognition Algorithms and Design Methodologies for Acyclic Database Schemes*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 164–185.

[DayB] U. DAYAL, P.A. BERNSTEIN, *On the Correct Translation of Update Operations on Relational Views*, ACM TODS 8, 3, (1982), pp. 381–416.

[DebP] P. DE BRA, J. PAREDAENS, *Conditional Dependencies for Horizontal Decompositions*, Proc. 10th ICALP, Springer-Verlag LNCS 154, (1983), pp. 67–82.

[Del] C. DELOBEL, *Normalization and Hierarchical Dependencies in the Relational Data Model*, ACM TODS 3, 3, (1978), pp. 201–222.

[DelC] C. DELOBEL, R.C. CASEY, *Decomposition of a Database and the Theory of Boolean Switching Functions*, IBM J. Research and Development 17, 5, (1972), pp. 370–386.

[Der] M. DE ROUGEMONT, *Second-Order and Inductive Definability on Finite Structures*, Z. Math. Logik, 33, (1987), pp. 47–63.

[Dip] R.A. DIPAOLA, *The Recursive Unsolvability of the Decision Problem for a Class of Definite Formulas*, JACM 16, 2, (1969), pp. 324–327.

[DowST] P.J. DOWNEY, R. SETHI, R.E. TARJAN, *Variations on the Common Subexpression Problem*, JACM 27, 4, (1980), pp. 758–771.

[DreG] B.S. DREBEN, W.D. GOLDFARB, *The Decision Problem: Solvable Classes of Qualificational Formulas*, Addison-Wesley, (1979).

[DubM] P. DUBLISH, S.N. MAHESHWARI, *Expressibility of Bounded-Arity Fixed-Point Query Hierarchies*, Proc. 8th ACM PODS, (1989), pp. 324–336.

[DwoKM] C. DWORK, C. KANELLAKIS, P. MITCHELL, *On the Sequential Nature of Unification*,

J. Logic Programming 1, 1, (1984), pp. 35–50.

[DwoKS] C. DWORK, C. KANELLAKIS, L. STOCKMEYER, *Parallel Algorithms for Term Matching*, SIAM J. of Computing, 17,4, (1988), pp. 711–731.

[End] H.B. ENDERTON, *A Mathematical Introduction to Logic*, Academic Press, (1972).

[Fag1] R. FAGIN, *Generalized First-order Spectra and Polynomial-time Recognizable Sets*, Complexity of Computation, (R. Karp, ed.), SIAM-AMS Proc. 7, (1974), pp. 43–73.

[Fag2] R. FAGIN, *Monadic Generalized Spectra*, Z. Math. Logik, 21, (1975), pp. 89–96.

[Fag3] R. FAGIN, *Multivalued Dependencies and a New Normal Form for Relational Databases*, ACM TODS 2, 3, (1977), pp. 262–278.

[Fag4] R. FAGIN, *A Normal Form for Relational Databases that is Based on Domains and Keys*, ACM TODS 6, 3, (1981), pp. 387–415.

[Fag5] R. FAGIN, *Armstrong Databases*, Proc. 7th IBM Symposium on Mathematical Foundations of Computer Science, Japan, (1982). Also see, IBM Res. Rep. RJ3440, (1982).

[Fag6] R. FAGIN, *Horn Clauses and Database Dependencies*, JACM 29, 4, (1982), pp. 952–985.

[Fag7] R. FAGIN, *Degrees of Acyclicity for Hypergraphs and Relational Database Schemes*, JACM 30, 3, (1983), pp. 514–550.

[FagKUV] R. FAGIN, G. KUPER, J.D. ULLMAN, M.Y. VARDI, *Updating Logical Databases*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 1–18.

[FagMU] R. FAGIN, A.O. MENDELZON, J.D. ULLMAN, *A Simplified Universal Relational Assumption and its Properties*, ACM TODS 7, 3, (1982), pp. 343–360.

[FagMUY] R. FAGIN, J.D. ULLMAN, D. MAIER, M. YANNAKAKIS, *Tools for Template Dependencies*, SIAM J. Computing 12, 1, (1983), pp. 36–59.

[FagUV] R. FAGIN, J.D. ULLMAN, M.Y. VARDI, *On the Semantics of Updates in Databases*, Proc. 2nd ACM PODS, (1983), pp. 352–365.

[FagV] R. FAGIN, M.Y. VARDI, *The Theory of Data Dependencies: A Survey*, Mathematics of Information Processing, (M. Anshel and W. Gewirtz, eds.), Symp. in Appl. Math. 34, (1986), pp. 19–72.

[FisT] P.C. FISCHER, D.M. TSOU, *Whether a Set of Multivalued Dependencies Implies a Join Dependencies is NP-hard*, Vanderbilt Univ. Res. Rep., (1981).

[Gai] H. GAIFMAN, *On Local and Nonlocal Properties*, Proc. Logic Colloquium, (J. Sterne, ed.), North Holland, (1981), pp. 105–132.

[GaiMSV] H. GAIFMAN, H. MAIRSON, Y. SAGIV, M.Y. VARDI, *Undecidable Optimization Problems for Database Logic Programs*, Proc. 2nd IEEE LICS, (1987), pp. 106–115.

[GaiV] H. GAIFMAN, M.Y. VARDI, *A Simple Proof that Connectivity of Finite Graphs is not First-order Definable*, Bulletin EATCS 26, (1985), pp. 43-45.

[Gal] Z. GALIL, *An Almost Linear-time Algorithm for Computing a Dependency Basis in a Relational Database*, JACM 29, 1, (1982), pp. 96–102.

[GalM] H. GALLAIRE, J. MINKER, EDS., *Logic and Databases*, Plenum Press, (1978).

[GalMN] H. GALLAIRE, J. MINKER, J.-M. NICOLAS, *Logic and Databases: A Deductive Approach*, ACM Computing Surveys 16, 2, (1984), pp. 153–185.

[GarJ] M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, (1979).

[GinZ] S. GINSBURG, S.M. ZAIDDAN, *Properties of Functional Dependency Families*, JACM 29, 4, (1982), pp. 678–698.

[GooS1] N. GOODMAN, O. SHMUELI, *Tree Queries: A Simple Class of Queries*, ACM TODS 7, 4, (1982) pp. 653–677.

[GooS2] N. GOODMAN, O. SHMUELI, *Syntactic Characterization of Tree Database Schemas*, JACM 30, 4, (1983) pp. 767–786.

[GooS3] N. GOODMAN, O. SHMUELI, *The Tree Projection Theorem and Relational Query Processing* JCSS 28, 1, (1984) pp. 60–79.

[GooT] N. GOODMAN, Y.C. TAY, *Synthesizing Fourth Normal Form Relations from Multivalued Dependencies*, Harvard Univ. Res. Rep., (1983).

[GotPZ] G. GOTTLOB, P. PAOLINI, R. ZICARI, *Properties and Update Semantics of Consistent Views*, ACM TODS, 13, 4, (1988), pp. 486–524.

[Gra1] M.H. GRAHAM, *On the Universal Relation*, Ph.D. Dissertation, Univ. of Toronto Res. Rep., (1979).

[Gra2] M.H. GRAHAM, *Functions in Databases*, ACM TODS 8, 1, (1983), pp. 81–109.

[GraMV] M.H. GRAHAM, A.O. MENDELZON, M.Y. VARDI, *Notions of Dependency Satisfaction*, JACM 33, 1, (1986), pp. 105–129.

[GraY] M.H. GRAHAM, M. YANNAKAKIS, *Independent Database Schemes*, JCSS 28, 1, (1984), pp. 121–141.

[Grh] G. GRAHNE, *Dependency Satisfaction in Databases with Incomplete Information*, Proc. 10th Conf. on Very Large Databases, (1984).

[GrhR] G. GRAHNE, K.-J. RÄIHÄ, *Characterizations for Acyclic Database Schemes*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 19–42.

[GrnJ] J. GRANT, B.E. JACOBS, *On the Family of Generalized Dependency Constraints*, JACM 29, 4, (1982), pp. 986–997.

[GrnM] J. GRANT, J. MINKER, *Answering Queries in Indefinite Databases and the Null Value Problem*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 19–42.

[Gur1] Y. GUREVICH, *Toward a Logic Tailored for Computational Complexity*, Computation and Proof Theory, (M.M. Richter etal. eds.), Springer-Verlag LNM 1104, (1984), pp. 175–216.

[Gur2] Y. GUREVICH, *Logic and the Challenge of Computer Science*, Trends in Theoretical Computer Science, (E. Börger), Computer Science Press, (1988), pp. 1–57.

[GurL] Y. GUREVICH, H.R. LEWIS, *The Inference Problem for Template Dependencies*, Proc. 1st ACM PODS, (1982), pp. 221–229.

[GurS] Y. GUREVICH, S. SHELAH, *Fixed-point Extensions of First-order Logic*, Annals of Pure and Applied Logic 32, North Holland (1986), pp. 265–280. Also, 26th IEEE FOCS, (1985), pp. 346–353.

[GysV] M. GYSSENS, D. VAN GUCHT, *The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra*, Proc. ACM SIGMOD, (1988), pp. 225-232.

[HadN] R. HADDAD, J. NAUGHTON, *Counting Methods for Cyclic Relations*, Proc. 7th ACM PODS, (1988), pp. 333–340.

[HagITK] K. HAGIHARA, M. ITO, K. TANIGUCHI, T. KASAMI, *Decision Problems for Multivalued Dependencies in Relational Databases*, SIAM J. Computing 8, 2, (1979), pp. 247–264.

[Har] D. HAREL, *First-Order Dynamic Logic*, Springer-Verlag LNCS 68, (1979).

[HarK] D. HAREL, D. KOZEN, *A Programming Language for the Inductive Sets and Applications*, Inf. and Control 63, (1984), pp. 118–139.

[Kan2] P. C. KANELLAKIS, *Logic Programming and Parallel Complexity*, Foundations of Deductive Data-bases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 547–586.

[KanCV] P.C. KANELLAKIS, S.S. COSMADAKIS, M.Y. VARDI, *Unary Inclusion Dependencies Have Polynomial Time Inference Problems*, Proc. 15th ACM STOC, (1983), pp. 264–277.

[Kel] A. KELLER, *Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections and Joins* Proc. 4th ACM PODS, (1985), pp. 154–163.

[KelU] A. KELLER, J.D. ULLMAN, *On Complementary and Independent Mappings*, Proc. ACM SIGMOD, (1984).

[KifB] M. KIFER, C. BEERI *An Integrated Approach to Logical Design of Relational Database Schemes* ACM TODS, 11,2, (1986), pp. 134–158.

[KifRS] M. KIFER, R. RAMAKRISHNAN, A. SILBERSCHATZ *An Axiomatic Approach to Deciding Query Safety in Deductive Databases*, Proc. 7th ACM PODS, (1988), pp. 52–60.

[Klu1] A. KLUG, *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions*, JACM 29, 3, (1982), pp. 699–717.

[Klu2] A. KLUG, *On Conjunctive Queries Containing Inequalities*, JACM 35, 1, (1988), pp. 146–160.

[Kol] P.G. KOLAITIS, *The Expressive Power of Stratified Logic Programs*, manuscript, (1987), submitted to Inf. and Computation.

[KolP] P.G. KOLAITIS, C.H. PAPADIMITRIOU, *Why Not Negation by Fixpoint?* Proc. 7th ACM PODS, (1987), pp. 231–239.

[KorS] H.F. KORTH, A. SILBERSCHATZ, *Database System Concepts*, McGraw-Hill, (1986).

[Koz] D. KOZEN, *Complexity of Finitely Presented Algebras*, Proc. 9th ACM STOC, (1977), pp. 164–177.

[Kuh] J.L. KUHNS, *Answering Questions by Computer: a Logical Study*, Rand Corp. Res. Rep., RM-5428-PR, (1967).

[Kup1] G.M. KUPER, *Logic Programming with Sets*, Proc. 6th ACM PODS, (1987), pp. 11–20.

[Kup2] G.M. KUPER, *On the Expressive Power of Logic Programming Languages with Sets*, Proc. 7th ACM PODS, (1988), pp. 10–14.

[KupV1] G. KUPER, M.Y. VARDI, *A New Approach to Database Logic*, Proc. 3rd ACM PODS, (1984), pp. 86–96.

[KupV2] G. KUPER, M.Y. VARDI, *On the Complexity of Queries in the Logical Database Model*, Proc. 2nd Inter. Conf. on Database Theory, (1988), pp. 267–280.

[LakM] V.S. LAKSHMANAN, A.O. MENDELZON, *Inductive Pebble Games and the Inductive Power of Datalog*, Proc. 8th ACM PODS, (1989), pp. 301–311.

[LavMG] K. LAVER, A.O. MENDELZON, M.H. GRAHAM, *Functional Dependencies on Cyclic Database Schemes*, Proc. ACM SIGMOD, (1983), pp. 79–91.

[Lei] D. LEIVANT, *Descriptive Characterizations of Computational Complexity*, CMU Res. Rep., (1988).

[Lev] H.J. LEVESQUE, *Foundations of a Functional approach to Knowledge Representation*, AI 23, (1984), pp. 155-212.

[Lie] E. LIEN, *On the Equivalence of Database Models*, JACM 29, 2, (1982), pp. 333–363.

[LinTK] T. LING, F. TOMPA, T. KAMEDA, *An Improved Third Normal Form for Relational Databases*, ACM TODS 6, 2, (1981), pp. 326–346.

[Lip1] W. LIPSKI, *On Semantic Issues Connected with Incomplete Information Databases*, ACM

TODS 4, 3 (1979), pp. 262–296.

[Lip2] W. LIPSKI, *On Databases with Incomplete Information*, JACM 28, 1, (1981), pp. 41–70.

[LiuD] L. LIU, A. DEMERS, *An Algorithm for Testing the Lossless Join Property in Relational Databases*, Inf. Proc. Letters 11, 2, (1980), pp. 73–76.

[LucO] C.L. LUCCHESI, S.L. OSBORN, *Candidate Keys for Relations*, JCSS 17, 2, (1978), pp. 270–279.

[Mah] M.J. MAHER, *Equivalences of Logic Programs*, Foundations of Deductive Databases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 627–658.

[Mai1] D. MAIER, *The Theory of Relational Databases*, Computer Science Press, (1983).

[Mai2] D. MAIER, *Minimum Covers in the Relational Database Model*, JACM 27, 4, (1980), pp. 664–674.

[MaiMS] D. MAIER, A.O. MENDELZON, Y. SAGIV, *Testing Implications of Data Dependencies*, ACM TODS 4, 4, (1979), pp. 455–469.

[MaiMSU] D. MAIER, A.O. MENDELZON, F. SADRI, J.D. ULLMAN, *Adequacy of Decompositions of Relational Databases*, JCSS 21, 3, (1980), pp. 368–379.

[MaiRW] D. MAIER, D. ROZENSHTEIN, D.S. WARREN, *Window Functions*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 213–246.

[MaiSY] D. MAIER, Y. SAGIV, M. YANNAKAKIS, *On the Complexity of Testing Implications of Functional and Join Dependencies*, JACM 28, 4, (1981), pp. 680–695.

[MaiUV] D. MAIER, J.D. ULLMAN, M.Y. VARDI, *On the Foundations of the Universal Relation Model*, ACM TODS 9, 2, (1984), pp. 283–308.

[MaiW] D. MAIER, D.S. WARREN, *Computing with Logic: Logic programming with Prolog*, Benjamin Cummings, (1988).

[Mak] A. MAKINOUCHI, *A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model*, Proc. 3rd Conf. on Very Large Databases, (1977), pp. 447–453.

[ManR] H. MANNILA, K.-J. RÄIHÄ , *Small Armstrong Relations for Database Design*, Proc. 4th ACM PODS, (1985), pp. 245–250.

[ManW] S. MANCHANDA, D.S. WARREN, *A Logic-Based Language for Database Updates*, Foundations of Deductive Databases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 363–394.

[MarPS] A. MARCHETTI-SPACCAMELA, A. PELAGGI, D. SACCA *Worst-case Complexity Analysis of Methods for Logic Query Implementation* Proc. 6th ACM PODS, (1987) pp. 294–301.

[Men] A.O. MENDELZON, *Database States and Their Tableaux*, ACM TODS 9, 2, (1984), pp. 264–282.

[MenM] A. O. MENDELZON, D. MAIER, *Generalized Mutual Dependencies and the Decomposition of Data base Relations*, Proc. 5th Conf. on Ver Large Databases, (1979), pp. 75–82.

[Min1] J. MINKER, *Perspectives in Deductive Databases*, J. Logic Programming 5, 1, (1988), pp. 33–60.

[Min2] J. MINKER, ED., *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, (1988).

[MinN] J. MINKER, N. NICOLAS, *On Recursive Axioms in Relational Databases*, Inf. Systems 8, (1982), pp. 1–13.

[MilR] G.L. MILLER, J.H. REIF, *Parallel Tree Contraction and its Applications*, Proc. 26th IEEE FOCS, (1985), pp. 478–489.

[Mit] J.C. MITCHELL, *The Implication Problem for Functional and Inclusion Dependencies*, Inf.

and Control 56, 3, (1983), pp. 154–173.

[Mor] K. MORRIS *An Algorithm for Ordering Subgoals in NAIL!* Proc. 7th ACM PODS, (1988), pp. 82–88.

[MorUV] K. MORRIS, J.D. ULLMAN, A. VAN GELDER, *Design Overview of the NAIL! System*, Proc. 3rd Int. Conf. on Logic Programming, Springer-Verlag LNCS 225, (1986), pp. 554–568.

[Mos] Y.N. MOSCHOVAKIS, *Elementary Induction on Abstract Structures*, North Holland, (1974).

[Naq] S. NAQVI, *A Logic for Negation in Database System*, Proc. Workshop on Logic Databases, Washington, D.C. (1986).

[NaqK] S. NAQVI, R. KRISHNAMURTHY, *Database Updates in Logic Programming*, Proc. 7th ACM PODS, (1988), pp.

[Nau1] J.F. NAUGHTON, *Data Independent Recursion in Deductive Databases*, Proc. 5th ACM PODS, (1986), pp. 267–279.

[Nau2] J.F. NAUGHTON *One Sided Recursions*, Proc. 6th ACM PODS, (1987), pp. 340–348.

[NauS] J.F. NAUGHTON, Y. SAGIV, *A Decidable Class of Bounded Recursions*, Proc. 6th ACM PODS, (1987), pp. 227–236.

[Nic] J-M. NICOLAS, *First Order Logic Formalization for Functional, Multivalued, and Mutual Dependencies*, Proc. ACM SIGMOD, (1978), pp. 40–46.

[Osb] S.L. OSBORN, *Normal Forms for Relational Databases*, Ph.D. Dissertation, Univ. of Waterloo Res. Rep., (1977).

[OzsY] Z.M. OZSOYOGLOU, L.-Y. YUAN, *A New Normal Form for Nested Relations*, ACM TODS 12, 1, (1987), pp. 111–136.

[Pal] F.P. PALERMO, *A Database Search Problem*, Information Systems COINS IV, (J.T. Tou, ed.), Plenum Press, (1974).

[Pap] C.H. PAPADIMITRIOU, *The Theory of Concurrency Control*, Computer Science Press, (1986).

[Par] J. PAREDAENS, *On the Expressive Power of the Relational Algebra*, Inf. Proc. Letters 7, 2, (1978).

[ParJ] J. PAREDAENS, D. JANSSENS, *Decompositions of Relations: a Comprehensive Approach*, Advances in Data Base Theory, vol. 1, (H. Gallaire, J. Minker, J-M. Nicolas, eds.), Plenum (1981), pp. 73–100.

[ParV] J. PAREDAENS, D. VAN GUCHT, *Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions*, Proc. 7th ACM PODS, (1988), pp. 29–38.

[ParP] D.S. PARKER, K. PARSAYE-GHOMI, *Inference Involving Embedded Multivalued Dependencies and Transitive Dependencies*, Proc. ACM SIGMOD, (1980), pp. 52–57.

[PatW] M.S. PATERSON, M.N. WEGMAN, *Linear Unification*, JCSS 16, (1978), pp. 158–167.

[RamSUV] R. RAMAKRISHNAN, Y. SAGIV, J.D. ULLMAN, M.Y. VARDI, *Proof-tree Transformation Theorems and their Applications*, Proc. 8th ACM PODS, (1989), pp. 172-182.

[Rei1] R. REITER, *On Closed World Data Bases*, Logic and Databases, (H. Gallaire, J. Minker, eds.), Plenum, (1978), pp. 55–76.

[Rei2] R. REITER, *Towards a Logical Reconstruction of Relational Database Theory*, On Conceptual Modeling, (M.L. Brodie, J.L. Mylopoulos, J.W. Schmidt, eds.), Springer-Verlag, (1984), pp. 163–189.

[Rei3] R. REITER, *A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values*, JACM 33, 2, (1986), pp. 349–370.

[Ris] J. RISSANEN, *Independent Components of Relations*, ACM TODS 2, 4 (1977), pp. 317–325.

[Rob] J. A. ROBINSON, *A Machine Oriented Logic Based on the Resolution Principle*, JACM 12, 1, (1965) pp. 23–41.

[RotKS] M.A. Roth, H.F. Korth, A. Silberschatz, *Theory of Non-First-Normal-Form Relational Databases*, Univ. of Texas Austin Res. Rep. TR-84-36, (1986).

[Ruz] W.L. Ruzzo, *Tree-size Bounded Alternation*, JCSS 21, 2, (1980), pp. 218–235.

[Sac] D. Sacca, *Closures of Database Hypergraphs*, JACM 32, 4, (1985), pp. 774–803.

[SacMM] D. Sacca, F. Manfredi, A. Mecchia, *Properties of Database Schemata with Functional Dependencies*, Advances in Computing Research, vol. 3, (P.C. Kanellakis, F. Preparata, eds.), JAI Press, (1986), pp. 105–137.

[SacZ] D. Sacca, C. Zaniolo *On the Implementation of a Simple Class of Logic Queries for Databases* Proc. 5th ACM PODS, (1986), pp. 16–23.

[SadU] U. F. Sadri, J.D. Ullman, *Template Dependencies: A Large Class of Dependencies in Relational Database and Their Complete Axiomatization*, JACM 29, 2, (1981), pp. 363–372.

[Sag1] Y. Sagiv, *An Algorithm for Inferring Multivalued Dependencies with an Application to Propositional Logic*, JACM 27, 2, (1980), pp. 250–262.

[Sag2] Y. Sagiv, *Can We Use the Universal Assumption Without Using Nulls?*, Proc. ACM SIGMOD, (1981), pp. 108–120.

[Sag3] Y. Sagiv, *A Characterization of Globally Consistent Database and Their Correct Access Paths*, ACM TODS 8, 2 (1983), pp. 266–286.

[Sag4] Y. Sagiv, *Evaluation of Queries in Independent Database Schemes*, Hebrew Univ. of Jerusalem Res. Rep., (1984).

[Sag5] Y. Sagiv, *On Computing Restricted Projections of Representative Instances*, Proc. 4th ACM PODS, (1985), pp. 171–180.

[Sag6] Y. Sagiv, *Optimizing Datalog Programs*, Foundations of Deductive Databases and Logic Programming, (J. Minker, ed.), Morgan-Kaufmann, (1988), pp. 659–698.

[SagDPF] Y. Sagiv, C. Delobel, D.S. Parker, R. Fagin, *An Equivalence between Relational Database Dependencies and a Fragment of Propositional Logic*, JACM 28, 3, (1981), pp. 435–453.

[SagS1] Y. Sagiv, O. Shmueli, *On Finite FD-Acyclicity*, Proc. 5th ACM PODS, (1986), pp. 173–182.

[SagS2] Y. Sagiv, O. Shmueli, *The Equivalence of Solving Queries and Producing Tree Projections*, Proc. 5th ACM PODS, (1986), pp. 160–172.

[SagV] Y. Sagiv, M.Y. Vardi, *Safety of Datalog Queries over Infinite Databases*, Proc. 8th ACM PODS, (1989), pp. 160-172.

[SagW] Y. Sagiv, S. Walecka, *Subset Dependencies and a Completeness Result for a Subclass of Embedded Multivalued Dependencies*, JACM 29, 1, (1982), pp. 103–117.

[SagY] Y. Sagiv, M. Yannakakis, *Equivalence Among Expressions with the Union and Difference Operators*, JACM 27, 4, (1980), pp. 633–655.

[Sar] Y. Saraiya, *Linearising Nonlinear Recursions in Polynomial Time*, Proc. 8th ACM PODS, (1989), pp. 182-190.

[Saz] V. Sazonov, *A Logical Approach to the Problem of "P=NP,"*, Springer-Verlag LNCS 88, (1980), pp. 562–575.

[SchS] H.-J. Scheck, M. Scholl, *The Relational Model with Relation-valued Attributes*, Information Systems, (1986).

[Sci1] E. Sciore, *Real-World MVDs*, Proc. ACM SIGMOD, (1981), pp. 121–132.

[Sci2] E. Sciore, *A Complete Axiomatization of Full Join Dependencies*, JACM 29, 2, (1982), pp.

373–393.

[Sci3] E. SCIORE, *Comparing the Universal Instance and Relational Data Models*, Advances in Computing Research, vol. 3: The Theory of Databases, (P.C. Kanellakis and F. Preparata, eds.) JAI Press, (1986), pp. 139–163.

[Sel] P. SELINGER, *Chickens and Eggs - The Interrelationship of Systems and Theory*, Proc. 6th ACM PODS, (1987), pp. 250–253.

[SelEtal] P. SELINGER, M.M. ASTRAHAN, D.D. CHAMBERLIN, R.A. LORIE, T.G. PRICE, *Access Path Selection in a Relational Database Management System*, Proc. ACM SIGMOD, (1979), pp. 23–34.

[Shm] O. SHMUELI, *Decidability and Expressiveness Aspects of Logic Queries*, Proc. 6th ACM PODS, (1987), pp. 237-249.

[ShmTZ] O. SHMUELI, S. TSUR, C. ZANIOLO *Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL)* Proc. 7th ACM PODS, (1988), pp. 15–28.

[Spy] N. SPYRATOS, *The Partition Model: A Deductive Database Model*, ACM TODS 12, 1, (1987), pp. 1-37.

[Sto] L. STOCKMEYER, *The Polynomial-time Hierarchy*, TCS 3, (1977), pp. 1–22.

[TarY] R. E. TARJAN, M. YANNAKAKIS, *Simple Linear-time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs*, SIAM J. Computing 13, 3, (1984), pp. 566–579.

[TarT] A. TARSKI, F.B. THOMPSON, *Some General Properties of Cylindric Algebras*, (abstract), Bulletin of the AMS, 58 (1952), p. 65.

[ThoF] S.J. THOMAS, P.C. FISCHER, *Nested Relational Structures*, Advances in Computing Research, vol.3, (P.C. Kanellakis, F. Preparata, eds.) JAI Press, (1986), pp. 269–307.

[Tra] B.A. TRAHTENBROT, *Impossibility of an Algorithm for the Decision Problem in Finite Classes*, Doklady Akad. Nauk. SSSR 70, (1950), pp. 569–572.

[TsuZ] S. TSUR, C. ZANIOLO, *LDL: A Logic-based Data-Language*, Proc. 12th Conf. on Very Large Databases, (1986).

[Ull1] J.D. ULLMAN, *Principles of Database and Knowledge Base Systems: Volume I*, Computer Science Press, (1988).

[Ull2] J.D. ULLMAN, *Implementation of Logical Query Languages for Databases*, ACM TODS, 10, 3, (1985), pp. 289–321.

[Ull3] J.D. ULLMAN, *Database Theory – Past and Future* , Proc. 6th ACM PODS, (1987), pp. 1–10.

[UllVa] J. ULLMAN, M. VARDI *The Complexity of Ordering Subgoals* Proc. 7th ACM PODS, (1988), pp. 74–81.

[UllVg] J.D. ULLMAN, A. VAN GELDER, *Parallel Complexity of Logical Query Programs*, Algorithmica, 3, 1, (1988), pp. 5–42.

[Var1] M.Y. VARDI, *The Implication Problem for Data Dependencies in Relational Databases*, Ph.D. Dissertation, Hebrew Univ. of Jerusalem Res. Rep., (1981).

[Var2] M.Y. VARDI, *The Decision Problem for Database Dependencies*, Inf. Proc. Letters 12, 5, (1981), pp. 251–254.

[Var3] M.Y. VARDI, *Global Decision Problems for Relational Databases*, Proc. 22nd IEEE FOCS, (1981), pp. 198–202.

[Var4] M.Y. VARDI, *On Decomposition of Relational Databases*, Proc. 23rd IEEE FOCS, (1982), pp. 176–185.

[Var5] M.Y. VARDI, *The Complexity of Relational Query Languages*, Proc. 14th ACM STOC, (1982), pp. 137–146.

[Var6] M.Y. VARDI, *Inferring Multivalued Dependencies from Functional and Join Dependencies*, Acta Informatica, 19 (1983), pp. 305–324.

[Var7] M.Y. VARDI, *The Implication and Finite Implication Problems for Typed Template Dependencies*, JCSS 28, 1 (1984), pp. 3–28.

[Var8] M.Y. VARDI, *Fundamentals of Dependency Theory*, IBM Res. Rep. RJ4858, (1985).

[Var9] M.Y. VARDI, *Querying Logical Databases*, JCSS 32, 2, (1986).

[Var10] M.Y. VARDI, *On the Integrity of Databases with Incomplete Information*, Proc. 5th ACM PODS, (1986), pp. 252–266.

[Var11] M.Y. VARDI, *Decidability and Undecidablity Results for Boundedness of Linear Recursive Queries*, Proc. 7th ACM PODS, (1988), pp. 341–351.

[Vas] Y. VASSILIOU, *A Formal Treatment of Imperfect Information in Data Management*, Ph.D. Dissertation, Univ of Toronto Res. Rep. CSRG-TR-123, (1980).

[Vge1] A. VAN GELDER, *Negation as Failure Using Tight Derivations for General Logic Programs*, Proc. 3rd IEEE Symp. on Logic Programming, (1986), pp. 127–139.

[Vge2] A. VAN GELDER, *The Alternating Fixpoint of Logic Programs with Negation*, Proc. 8th ACM PODS, (1989), pp. 1-11.

[VgeT] A. VAN GELDER, R. TOPOR, *Safety and Correct Translation of Relational Calculus Formulas*, Proc. 6th ACM PODS, (1987), pp. 313–328.

[Vgu] D. VAN GUCHT, *On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model*, Proc. 6th ACM PODS, (1987), pp. 302–312.

[Via] V. VIANU, *Dynamic Functional Dependencies and Database Aging*, JACM 34, 1, (1987), pp. 28–59.

[Win1] M. WINSLETT, *A Model-Theoretic Approach to Updating Logical Databases*, Proc. 5th ACM PODS, (1986), pp. 224–234.

[Win2] M. WINSLETT, *A Framework for Comparison of Update Semantics*, Proc. 7th ACM PODS, (1988), pp. 315–324.

[WonY] E. WONG, K. YOUSSEFI, *Decomposition - a Strategy for Query Processing*, ACM TODS 1, 3, (1976), pp. 223–241.

[Yan1] M. YANNAKAKIS, *Algorithms for Acyclic Database Schemes*, Proc. 7th Conf. on Very Large Databases, (1981), pp. 82–94.

[Yan2] M. YANNAKAKIS, *Querying Weak Instances*, Advances in Computing Research, vol. 3, (P.C. Kanellakis and F. Preparata, eds.), JAI Press, (1986), pp. 185–212.

[YanP] M. YANNAKAKIS, C. PAPADIMITRIOU, *Algebraic Dependencies*, JCSS 25, 2, (1982), pp. 3–41.

[Yas] H. YASUURA, *On the Parallel Complexity of Unification*, Yajima Lab. Res. Rep. ER-83-01, (1983).

[YuO] C.T. YU, M.Z. OZSOYOGLU, *An Algorithm for Tree-query Membership of a Distributed Query*, Proc. IEEE COMPSAC, (1979), pp. 306–312.

[YuaO] L.Y. YUAN, M.Z. OZSOYOGLU, *Logical Design of Relational Databases Schemes*, Proc. 7th ACM PODS, (1987), pp. 38–47.

[Zan1] C. ZANIOLO, *Analysis and Design of Relational Schemata for Database Systems*, Ph.D. Dissertation, UCLA Res. Rep. ENG-7669, (1976).

[Zan2] C. ZANIOLO, *Database Relations with Null Values*, JCSS 28, 1, (1984), pp. 142–166.

Index

Normalization Theory, Proc. 4th Conf. on Very Large Databases, (1978), pp. 113–124.

[BeeDFS] C. BEERI, M. DOWD, R. FAGIN, R. STATMAN, *On the Structure of Armstrong Relations for Functional Dependencies*, JACM 31, 1, (1984), pp. 30–46.

[BeeFH] C. BEERI, R. FAGIN, J.H. HOWARD, *A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations*, Proc. ACM SIGMOD, (1977), pp. 47–61.

[BeeEtal] C. BEERI, R. FAGIN, D. MAIER, A.O. MENDELZON, J.D. ULLMAN, M. YANNAKAKIS, *Properties of Acyclic Database Schemes*, Proc. 13th ACM STOC, (1981), pp. 355–362.

[BeeFMY] C. BEERI, R. FAGIN, D. MAIER, M. YANNAKAKIS, *On the Desirability of Acyclic Database Schemes*, JACM 30, 3, (1983), pp. 479–513.

[BeeH] C. BEERI, P. HONEYMAN, *Preserving Functional Dependencies*, SIAM J. Computing 10, 3, (1981), pp. 647–656.

[BeeKBR] C. BEERI, P.C. KANELLAKIS, F. BANCILHON, R. RAMAKRISHNAN, *Bounds on the Propagation of Selection into Logic Programs*, Proc. 6th ACM PODS, (1987), pp. 214–227.

[BeeMSU] C. BEERI, A.O. MENDELZON, Y. SAGIV, J.D. ULLMAN, *Equivalence of Relational Database Schemes*, SIAM J. Computing 10, 2, (1981), pp. 352–370.

[BeeNRST] C. BEERI, S. NAQVI, R. RAMAKRISHNAN, O. SHMUELI, S. TSUR, *Sets and Negation in a Logic Database Language (LDL1)*, Proc. 6th ACM PODS, (1987), pp. 21–37.

[BeeRa] C. BEERI, R. RAMAKRISHNAN, *On the Power of Magic*, Proc. 6th ACM PODS, (1987), pp. 269–283.

[BeeRi] C. BEERI, J. RISSANEN, *Faithful Representation of Relational Database Schemes*, IBM Res. Rep. RJ2722, (1980).

[BeeV1] C. BEERI, M.Y. VARDI, *On the Complexity of Testing Implications of Data Dependencies*, Hebrew Univ. of Jerusalem Res. Rep., (1980).

[BeeV2] C. BEERI, M.Y. VARDI, *Formal Systems for Join Dependencies*, Hebrew Univ. of Jerusalem Res. Rep., (1981).

[BeeV3] C. BEERI, M.Y. VARDI, *The Implication Problem for Data Dependencies*, Proc. 8th ICALP, Springer-Verlag LNCS 115, (1981), pp. 73–85.

[BeeV4] C. BEERI, M.Y. VARDI, *On the Properties of Join Dependencies*, Advances in Database Theory, vol. 1, (H. Gallaire, J. Minker, J.-M. Nicolas, eds.) Plenum Press, (1981), pp. 25–72.

[BeeV5] C. BEERI, M.Y. VARDI, *A Proof Procedure for Data Dependencies*, JACM 31, 4, (1984), pp. 718–741.

[BeeV6] C. BEERI, M.Y. VARDI, *Formal Systems for Tuple and Equality Generating Dependencies*, SIAM J. Computing 13, 1, (1984), pp. 76–98.

[BeeV7] C. BEERI, M.Y. VARDI, *On Acyclic Database Decompositions*, Inf. and Control 61, 2, (1984), pp. 75–84.

[Ber] P.A. BERNSTEIN, *Synthesizing Third Normal Form Relations from Functional Dependencies*, ACM TODS 1, 4, (1976), pp. 277–298.

[BerC] P.A. BERNSTEIN, D.W. CHIU, *Using Semi-Joins to Solve Relational Queries*, JACM 28, 1, (1981), pp. 25–40.

[BerG] P.A. BERNSTEIN, N. GOODMAN, *The Power of Natural Semi-Joins*, SIAM J. Computing 10, 4, (1981), pp. 751–771.

[BerHG] P.A. BERNSTEIN, V. HADZILACOS, N. GOODMAN, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, (1987).

[BisDB] J. BISKUP, U. DAYAL, P.A. BERNSTEIN, *Synthesizing Independent Database Schemas*, Proc. ACM SIGMOD, (1979) pp. 143–152.