Databases and Finite-Model Theory

Victor Vianu

Abstract. Databases provide one of the main concrete scenarios for finite-model theory within computer science. This paper presents an informal overview of database theory aimed at finite-model theorists, emphasizing the specificity of the database area. It is argued that the area of databases is a rich source of questions and vitality for finite-model theory.

1. Introduction

There is an intimate connection between finite-model theory and database theory. In a very real sense, finite-model theory provides the backbone of database theory. And databases provide a *concrete scenario* for finite-model theory – perhaps the main one within computer science. This scenario contributes two things to finite-model theory. First, it induces a specific measure of relevance on finite-model theory questions and results. Second, it is itself a rich source of questions, some of which would be unlikely to arise independently. This paper is an attempt to present database theory to finite-model theory researchers, emphasizing the specificity of database theory. We hope to convince finite-model theorists that database theory is a source of vitality for their area.

We begin with an introduction to the database scenario, and continue with a brief overview of the general landscape of classical database theory. We emphasize several areas in database theory that have been largely overlooked by finite-model theory and that we believe are a rich source of questions. Chief among these are dynamic aspects of databases – update languages, view updates and maintenance, temporal databases and constraints, etc. Next, we look at the theory of query languages, where most of the overlap between database theory and finite-model theory occurs. We avoid covering the overlap, which is familiar terrain to finite-model theory researchers. Instead, we focus on the idiosyncrasies of the database scenario, its specificity, and the differences with finite-model theory. We next describe several developments in advanced database systems, and some of the accompanying theory. These include object-oriented databases, databases with incomplete information, and multimedia databases. Lastly, we discuss briefly some aspects of the interaction between databases and finite-model theory.

A word of warning is in order. We cover here many aspects of databases, and opt for breadth rather than depth in the presentation. Consequently, the exposition is informal. As an attenuating circumstance, we invoke the modest

The author was supported in part by NSF Grant IRI-9221268.

aims of the paper: to give a flavor of the database area, and to incite curiosity. We refer the reader to [**Kan91, AHV95**] for more detailed and somewhat formal presentations of database theory. Early database theory is covered in [**Mai83**]. A concise overview of the field aimed at computer science theoreticians is also provided in [**Yan95**]. The relationship of database theory and practice is explored in [**Pap95**]. A general introduction to the database area is provided in [**KS91**], and an in-depth presentation in [**Ull88, Ull89**].

2. The Database Scenario

A database system is a software and hardware system whose purpose is to store and manipulate data. The definition is even more ambiguous than it seems at first. The meaning of "data" changes with the demands of the market and the increased capabilities of computers. So does the meaning of "manipulate". However, some aspects of the database scenario are relatively constant. They include the basic architecture of a database. We discuss this first.

The logical level. Users–who may be people as well as programs–interact with a database using the interface it provides. The abstraction provided by the interface is called the *logical* level of the database. Data is represented here using a *data model*. Intuitively, a data model provides a uniform way to organize and manipulate data. Examples of data models are the relational, hierarchical, network, entity-relationship, and object-oriented models. By far the most popular remains the relational model, in which data is presented as a set of tables. Each table is identified by a name; columns also have names, called *attributes*. The structure of the tables in a relational database, given by their names and attributes, is referred to as the database *schema*. The schema provides the "skeleton" of the database, without its data. The content of each table at any given time is a set of tuples–a relation. The relations contained by the tables form an *instance* of the database. Data manipulation capabilities provided at the logical level include a *query language* (used to extract information from the database) and an *update language* (used to modify the content of the database).

A relational database schema is essentially a first-order vocabulary without function symbols or constants. A database instance can be viewed as a finite relational structure providing a finite interpretation for the vocabulary. This analogy is the basis for the connection between databases and finite-model theory.

Besides the basic functionality of the logical level, described above, there are many bells and whistles. We mention two important ones: *data dependencies*, and *views*.

Clearly, the ability of relations to capture interesting semantics of data is limited. One way to augment the descriptive capabilities of the model is to allow explicitly specifying additional properties that must be satisfied by all correct instances. This is usually done using first-order sentences, called *data dependencies* (or *integrity constraints*). The most common of these are *functional dependencies* and *inclusion dependencies*. Suppose *account* and *name* are attributes in a relation storing data about bank accounts. The expression

$$name \rightarrow account$$

is an example of a functional dependency. It states that no two tuples in the relation can have the same *name* value but distinct *account* values. Next, consider relations *Student* and *TA*, each of which has the attribute *name*. The expression

$$TA[name] \subseteq Student[name]$$

is an inclusion dependency stating that every TA is a student. Data dependencies serve several purposes. Their primary purpose it to provide a rough filter for incorrect data. As such, they must be checked following each modification of the database. Another use of dependencies is in the design of database schemas with desirable properties. A third application is in query optimization. This is quite intuitive – data dependencies provide more information about the data to which a query is applied, and this in turn may allow faster evaluation of the query. We discuss this later in more detail.

Another functionality commonly provided at the logical level is the ability to define *views* of the data. Suppose we have a database. A view of the database is a set of tables whose contents are defined from the given database by a query (one for each table in the view). There are many reasons for providing views. The database may be very complex, and different users may prefer to have different ways of looking at the data. Views typically involve both hiding of irrelevant information, and restructuring of data that is retained. They may be very simple, as in the case of automatic teller machines, or highly intricate, as in the case of computer-aided design systems. Another aspect is security: views can be used to hide information from certain users. The set of views provided by a database is sometimes referred to as the *external level* of the database.

Below the logical level. The implementation of the database occurs below the logical level in a hierarchy of levels of abstraction. At the bottom of the hierarchy lies the *physical level*, which consists of the data stored on some physical medium (say, on disk), as a sequence of bits. Access to data stored on the physical medium is facilitated by *indexes*, which act as directories providing physical addresses for tuples with a given value for some attribute. The processing of queries and updates takes place in between the logical and physical levels. Different stages in this process require different information, and therefore occur at different levels of abstraction. We next outline the main stages in the journey of a query from the logical level to the physical level. We limit the discussion to relational databases.

A query is formulated at the logical level in a language which is typically a syntactic variant of first-order logic over relations (FO). Such is the case for the most widely used commercial relational query languages such as SQL [**C$^+$76**]. Below the logical level, the query is processed in several stages:

- *translation to algebra:* the query is reformulated in an intermediate language, called *relational algebra*. This specifies the evaluation of the query in terms of some simple algebraic operations on relations (see Section 3.1).
- *query rewriting:* the relational algebra query is rewritten into a simpler query. This stage uses no information about the instance to which the query is applied. It uses static information such as the database schema and data dependencies. The result is a query which can be evaluated faster and is equivalent to the original on all instances satisfying the data dependencies.
- *query evaluation plan:* the plan for evaluating the query is generated. This can be represented in a graph-like manner and specifies the order in which

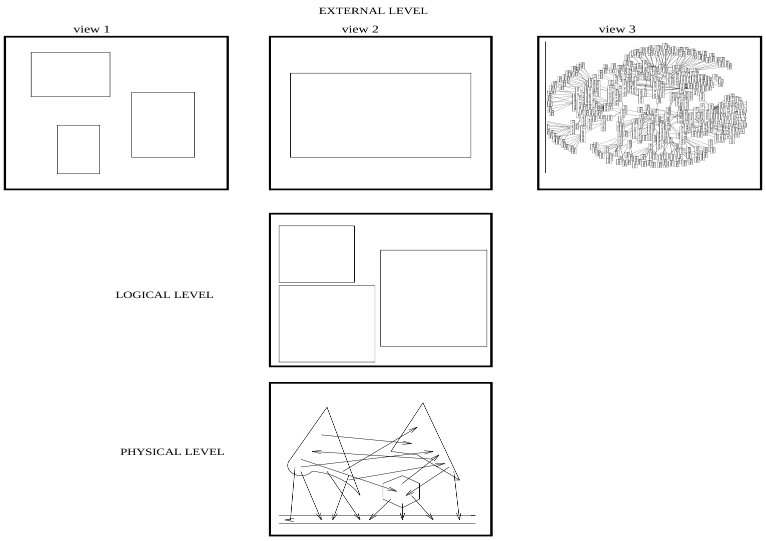

FIGURE 1. Three-level architecture of database systems

the algebra operations will be performed, the reuse of intermediate results, etc. Typically, this requires information about the instance to which the query is applied, including physical level information. Such information may include statistical information about the data (size of relations, distribution of data, etc.), and the availability of indexes for certain attributes. Sometimes indexes may be created dynamically.

- *query evaluation:* the algebra operations are executed as specified by the evaluation plan. The evaluation of each operation is highly optimized and may also use physical level information such as indexes.

The separation between the logical level and the physical level is perhaps the most fundamental idea underlying the field of databases. This is referred to as the *data independence principle*. It allows the user to think of the database in terms of its logical representation alone, without any knowledge of what happens underneath: the user need not know about disks, indexes, intermediate representations, etc. This capability is arguably the most important single distinction between file systems and database systems. As we shall see, the data independence principle has important consequences for database theory as well.

3. Landscape of Database Theory

Before embarking upon more detailed discussions of some selected parts of database theory, we briefly survey the general landscape.

In attempting to be useful, database theory is engaged in an intricate dance with database systems. Sometimes, database theory leads developments in systems. The most notable example is the birth of relational database systems, which followed Codd's theoretical work on the relational calculus and algebra [**Cod70, Cod72b**]. Other times, database theory follows developments in systems. One example is in the area of object-oriented databases, where formal work was late to emerge. In either case, there is a tension between the natural tendency of database theory to produce a unified, elegant formalism, and the whimsical reality of database systems. The systems rarely evolve under the impulse of intellectual or esthetic considerations, but are instead subject to powerful market, technological, and even political forces. There ensues a remarkably opportunistic lack of loyalty to any particular paradigm or formalism, much to the despair of theoreticians. One of the challenges of database theory is to follow (or lead) the Brownian motion of database systems, to stay relevant, and to discern the essential from the noise. But this is also an opportunity. Databases act as a vital soup where various paradigms interact: logic, complexity theory, algorithms, programming languages, artificial intelligence, logic programming, etc. This is one of the sources of richness and intellectual excitement in the database area.

The overlap of database theory with finite-model theory occurs primarily in the area of query languages. We discuss this in detail in the next section. Here we focus on the other areas of database theory, some of which we believe are sources of interesting new questions for finite-model theory.

Most of database theory deals with the logical level of databases. The main topics besides the theory of query languages are dependency theory, and dynamic aspects of databases. The latter include update languages, view maintenance, view updates, and temporal databases. Some database theory also deals with what happens below the logical level of the database. This includes query rewriting techniques, physical access structures, and concurrency control. We begin with a description of the basic formalism of relational databases, then discuss very briefly each of these topics.

3.1. Basic Formalism of Relational Databases. The basic notation and terminology of relational databases varies quite a bit. The reader should expect such variations when consulting different sources.

We assume the existence of four infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, the set **var** of *variables*, and the set **rel** of *relation names*. A *relational schema* is a relation name R with an associated finite set *att(R)* of attributes in **att**. A *tuple* over a relational schema R is a mapping from $att(R)$ into **dom**. An *instance* over a relation schema R is a *finite* set of tuples over R. A *database schema* is a finite set of relational schemas. An *instance* $\mathbf{I}$ over a database schema $\mathbf{R}$ is a mapping associating to each R in $\mathbf{R}$ an instance over R, denoted $\mathbf{I}(R)$. The set of constants occurring in an instance $\mathbf{I}$ is called the *active domain of* $\mathbf{I}$, denoted $adom(\mathbf{I})$. The set of all instances over a schema $\mathbf{R}$ is denoted by $inst(\mathbf{R})$.

Note that, in logic terms, a database schema supplies a vocabulary consisting of a finite set of predicates (however, a database schema additionally specifies a set of attributes for each predicate). A database instance provides a finite interpretation of the predicates. The domain of the interpretation is not given explicitly. It consists of the active domain of the instance. Also note the use of the word "constant". This is different from the notion of a constant in a first-order vocabulary. In databases, a constant is just a domain element.

The standard query languages for relational databases are based on a variation of first-order logic on relations, called *relational calculus* (which we will denote FO despite the minor differences with first-order logic). Suppose $\mathbf{R}$ is a database schema. A relational calculus query is an expression

$$\{\langle x_1, \ldots, x_n\rangle : A_1 \ldots A_n \mid \varphi(x_1, \ldots, x_n)\}$$

where the x_i are distinct variables, the A_i are distinct attributes, and φ is a first order formula with free variables $x_1, \ldots, x_n$, which allows atoms $x = c$ where c is a constant in **dom**. This defines a relation with attributes $\{A_1, \ldots, A_n\}$. Each constant used in a query is always interpreted by the constant itself[1].

Relational calculus has a simple algebraization called *relational algebra*. The operations of the algebra are the following:

- set operations: $\cup, -$;
- selection: $\sigma_{A=c}(R)$, for $A \in att(R)$, selects all tuples in R for which $A = c$;
- projection: $\pi_X(R)$ projects R on a subset X of its attributes;
- renaming: $\delta_{A \to B}(R)$ leaves the content of R unchanged and renames attribute A to B;
- join: $R \bowtie S$ returns all tuples t over $att(R) \cup att(S)$ such that $\pi_{att(R)}(t) \in R$ and $\pi_{att(S)}(t) \in S$.

The fact that FO has a simple algebraization had been known to logicians as Tarski's Algebraization Theorem [**HMT71**]. It was brought to the attention of the database community by Ted Codd [**Cod70, Cod72b**] (who went on to receive the Turing Award in 1981 for his development of the relational model [**Cod87**]). This result played a central role in the birth of practical relational databases.

For example, consider a database whose schema consists of the following relations[2]: *serves* with attributes *bar, beer*; and *frequents* with attributes *drinker* and *bar*. Assume for simplicity that every bar in the database serves some beer and is frequented by some drinker. Consider the query

"find all drinkers who frequent only bars serving Coors".

The query is expressed in the calculus as:

$$\{\langle d \rangle : \mathit{drinker} \mid \forall b[\mathit{frequents}(d,b) \to \mathit{serves}(b, \mathit{Coors})]\}$$

and in the algebra as:

$$\pi_{drinker}(\mathit{frequents}) - \pi_{drinker}[\mathit{frequents} \bowtie (\pi_{bar}(\mathit{serves}) - \pi_{bar}(\sigma_{beer=Coors}(\mathit{serves})))].$$

The importance of the equivalence of the calculus and algebra in the development of databases cannot be overstated. It is likely to be viewed by most mainstream database people as the main result in the theory of query languages.

3.2. Dependency Theory. Data dependencies are typically specified by FO sentences. Dependency theory has focused on very limited classes of FO sentences. This is due to two facts. First, most dependencies encountered in practice are quite simple. Second, it is important to be able to perform certain manipulations of the dependencies in a class of interest. This includes deciding if a dependency in the class is implied by a given set of dependencies in the same class. Of course, this is undecidable for arbitrary FO sentences. Finite axiomatizability is also important. Interestingly, dependency theory is the area within database theory where the difference between general model theory and finite-model theory was first considered. It introduced the notion of *finite implication* (implication over finite structures) and contrasted it to *unrestricted implication* over arbitrary structures (e.g., see [**JK84**]).

[1] The rationale for this difference with logic is simple enough: if the query asks for Joe's address, it actually means Joe and not Smith. Therefore there is no need to interpret Joe as anything but itself.

[2] This example is from [**Ull82**].

These notions were shown to coincide for some classes of dependencies, and to differ for others. In the remainder of the section, the term "implication" means *finite* implication, unless otherwise stated.

Two classes of dependencies have emerged as central: *functional dependencies (fds)* [**Cod72a**] and *inclusion dependencies (incds)* [**Dat81**]. We have seen informal examples of both in the previous section. Although inclusion dependencies often involve single attributes (as in the example), they generally involve several attributes. Specifically, an inclusion dependency is an expression $P[X] \subseteq Q[X]$ where P and Q are relations and X is a set of attributes common to P and Q. The meaning is that $\pi_X(P) \subseteq \pi_X(Q)$.

It turns out that the implication problem for fds and incds taken separately is decidable in linear time [**BB79**] and polynomial space [**CFP84**], respectively. (The problem is PSPACE-complete for incds [**CFP84**].) The implication problem becomes undecidable for fds and incds taken together [**CV85, Mit83**]. The proof uses an elegant reduction of the finite monoid word problem [**Pos47, Gur66**]. However, implication remains decidable for special cases of practical importance, such as fds together with *acyclic incds* [**CK85**]. A set of incds is acyclic if the following directed graph is acyclic. Its vertices are the relation names of the schema and there is an edge from P to Q if $P[X] \subseteq Q[X]$ is an incd in the set. Implication is also decidable when the incds are restricted to be *unary*, i.e. of the form $P[X] \subseteq Q[X]$ where X is a singleton [**KCV90**].

Dependency theory developed fast and deep. It studied many classes of FO sentences, which we do not describe here. The most general are *tuple generating dependencies* and *equality generating dependencies* [**BV81, Fag82**]. The first are generalizations of incds and the second are generalizations of fds. Several deep questions arose that remained open for some time, such as the decidability of implication for the class of *embedded multi-valued dependencies* [**Fag77**]. The question was recently settled in the negative [**Her95**].

Comprehensive presentations of dependency theory can be found in [**Var87, FV86**]. A more concise presentation is provided in [**Kan91**]. Dependency theory is also the topic of the book [**Tha91**].

3.3. Update Languages. The content of a database changes over time. This raises a host of problems related to dynamic aspects of databases, not usually considered in finite-model theory.

The first question is to provide languages for modifying the content of a database. These are dynamic analogs of query languages, and are called *update languages*. The difference between query and update languages is subtle but important. To specify an update, one could indeed define the new database as the answer to a query posed against the old database. However, this misses an essential characteristic of updates: most often, they involve small changes to the current database. And, query languages are not naturally suited to explicitly speak about *change*. In contrast, update languages use as building blocks simple statements expressing change, such as insertions, deletions, and modifications of tuples in the database. Other subtle differences arise from the fact that updates modify their input. This results in a built-in recursion which may be hard to circumvent in some languages. The difficulty is illustrated by a result of [**Don88**] involving the language Datalog (conjunctive queries augmented with an inflationary fixpoint operator). It is shown that Datalog is not closed under composition as an update language, although it is closed under composition as a query language. Specifically, there is no Datalog update equivalent to the Datalog update closing R transitively, followed by the Datalog update closing R symmetrically.

3.4. View Maintenance. As we have seen in the previous section, databases provide different users with different *views* of the database. Consider a database with schema $\mathbf{R}$. A view over $\mathbf{R}$ is a database schema $\mathbf{V}$ together with a mapping $v : inst(\mathbf{R}) \rightarrow inst(\mathbf{V})$. The mapping v is often defined by one FO query for each relation in $\mathbf{V}$. There are two main options in managing views: the view can be *materialized*, i.e. computed from the database and stored explicitly, or the view can be *virtual*, i.e. never explicitly stored. Queries on a virtual view are translated into a query on the database by substituting into the query the definition of each table in the view in terms of the database.

Materialized views must be updated whenever the database is updated. How to do this efficiently is the *view maintenance* problem. The object is to avoid recomputing the view from scratch every time an update occurs. Some work in this area has focused on detecting situations when the database update does not affect the view, therefore making it unnecessary to recompute it [**BC79, BLT86**]. The cost of checking relevance of the update to the view must be weighed against the cost of recomputing the view. Other recent work has considered the incremental maintenance of views. The idea is to compute the new view from the old view and the update to the database, possibly using some additional auxiliary information. Since computing a view really means evaluating a query, this has led to very interesting work on incremental query evaluation, ranging from pragmatic heuristics to new notions of "incremental" complexity. These are relevant to a much broader context than view maintenance. The incremental computational complexity of problems is considered in [**MSVT94**]. Dong, Su and Topor [**DT92, DS92, DS93, DST94**] bring in a descriptive complexity viewpoint. They consider certain queries not definable in FO (such as graph transitive closure) and ask if they can be maintained using FO queries (for example, transitive closure can be maintained under insertions by FO means). It is open whether there exists a *fixpoint* query which cannot be incrementally maintained in FO, in their sense. In the same spirit, Immerman and Patnaik [**PI94**] define the class Dyn-FO of properties that can be checked incrementally in FO. They show that many properties not definable in FO are in fact in Dyn-FO, including graph reachability, bipartiteness, minimum spanning trees, multiplication, and regular languages. The connection between Dyn-FO and classical complexity classes remains open. For example, it is not known whether Dyn-FO contains NC^1, L, or NL. Dyn-FO is contained in P [**PI94**], but it is not known if the containment is strict. In a broader context, extensive work on incremental computation has been done in the algorithms and programming languages communities (e.g., see [**CT91, Fre85, LT95, Rau94**]).

3.5. View Updates. To a user accessing a view, the view *is* the database. Therefore, the user must be allowed to query and update the view as if it were a database in its own right. Queries do not pose much of a problem: a query against the view is translated into a query against the database by substituting into the query the definition of each table in the view in terms of the underlying database. Updates are a different matter. What does an update of a view mean in terms of the database? Generally, there is no unambiguous translation. For example, consider a view V defined as the projection of a database table R, and an update consisting of the insertion of a new tuple in V. Obviously, there is no unambiguous way of translating this into an update of R, since no values are supplied for the additional attributes of R. This problem has given rise to considerable research, along two lines. The first school holds that the semantics of view updates must be specified explicitly as part of the definition of the view. This is rather straightforward. The second school attempts to identify reasonable situations when the translation can be done automatically and unambiguously. This has given rise to an elegant theory which we briefly describe [**BS81**]. The main idea is to disambiguate updates to a given view v using a second view v', called a *complement* of v. In order for v' to be a complement of v, the two views taken together must uniquely identify the underlying database. That is, the mapping that associates to each database the combined view $\langle v, v' \rangle$ must be injective. Now suppose that v' is kept fixed in an update. It follows that there is at most one way to translate an update of v into an update of the database. Thus, specifying a complement view v' implicitly specifies the semantics of updates of v. Unfortunately, not all views have complements and not all complements guarantee the existence of a solution to the view update. In fact, it turns out that this approach has very limited applicability (despite its elegance!) [**KU84**].

3.6. Temporal Databases. Classical databases model *static* aspects of data. Thus, the information in the database consists of data (presumed to be) currently true in the world. However, in many applications, information about the history of data is just as important as static information. When history is taken into account, queries can ask about the evolution of data through time; and, constraints may restrict the way changes occur.

Temporal databases borrow heavily from temporal logic [**Eme91, Gal87**]. They typically use a time model where time is viewed as discrete and linear. Thus, the database can be viewed as a sequence of instances. Queries can make explicit reference to time, or may use constructs such as *since, until, before, after, eventually, overlaps*. Several temporal extensions of SQL use a **when** clause to express a temporal condition. For example, consider a temporal database consisting of one table *Schedule* with attributes *theater* and *film*. To each tuple in *Schedule* is associated a temporal interval (or union of intervals) indicating its period of validity. Consider the query:

"Find the pairs of theaters that have shown the same film on the same date."

This can be expressed using the **when** clause as follows:

$$\begin{array}{ll} \textbf{select} & t_1.\mathit{theater}, t_2.\mathit{theater} \\ \textbf{from} & \mathit{Schedule}\ t_1\ t_2 \\ \textbf{where} & t_1.\mathit{title} = t_2.\mathit{title}\ \textbf{and} \\ \textbf{when} & t_1.\mathit{interval}\ \textbf{overlaps}\ t_2.\mathit{interval} \end{array}$$

The **when** clause above is true for tuples t_1, t_2 iff the intervals indicating their periods of validity have nonempty intersection.

Questions specific to databases focus on the efficient management of historical information and query evaluation. For example, what is a good representation of historical information? What information must be kept in order to be able to answer a specific class of temporal queries? Conversely, what temporal queries can be answered if resource bounds are placed on the historical information that can be kept? Such issues have only recently begun to be explored [**Cho92, LS87**], and remain largely unresolved. A survey of temporal database research, emphasizing theoretical aspects, is provided in [**Cho94**].

3.7. Below the Logical Level. While most database theory deals with the logical level, some of it concerns the levels below. This includes techniques for query processing, the design of physical access structures, and concurrency control.

Query rewriting has been the subject of some elegant theoretical work. Recall that relational calculus queries are translated below the logical level into relational algebra queries. It is easy to see that *join* is the most expensive operation of the algebra. A technique has been developed in [**CM77, ASU79, MMS79, SY80**] for minimizing the number of joins. Furthermore, the technique can take into account the fds satisfied by the database. The technique is guaranteed to produce a query equivalent to the original on all databases satisfying the fds, and which has the minimum number of joins among all such queries. We illustrate this with a simple example. Suppose R has attributes ABC and satisfies the fd $B \rightarrow C$. Consider the algebra query $\pi_{AB}(R) \bowtie \pi_{BC}(R)$. It turns out that this query is the identity on all instances of R satisfying $B \rightarrow C$. To see this, suppose first that $\langle a, b, c\rangle$ is a tuple in R. Clearly, $\langle a, b, c\rangle$ is also in the answer (this is not dependent on the fd). Conversely, suppose $\langle a, b, c\rangle$ is a tuple in the answer. Then there must be tuples $\langle a, b, c'\rangle$ and $\langle a', b, c\rangle$ in R. These agree on B. Since R satisfies $B \rightarrow C$, it must be that $c = c'$. But then $\langle a, b, c\rangle$ is in R. This kind of reasoning about fds and queries illustrates a technique called the *chase* [**MMS79**].

Much algorithmic and data structures work has gone into the design of indexes, such as B$^+$-trees [**BU77**]. Roughly speaking, a B$^+$-tree allows $log(n)$-time access to one of n records, if the tree is balanced. The main difficulty is how to efficiently keep the tree balanced as records are inserted or deleted.

Concurrency control develops means to ensure that concurrent access to the database by multiple users does not produce nonsense. The central idea is the following. Suppose n programs $T_1, \ldots, T_n$ are run on the database. These can query as well as update the database. For efficiency reasons, it is desirable to interleave the execution of the instructions of the programs. An interleaving of the instructions is a *schedule* for the programs. Which are the schedules that should be allowed? If the programs are simply executed one after the other in some order, there is clearly no problem. Such a schedule is called *serial*. It should then be the case that schedules which are *equivalent* to some serial schedule should also be allowed. Such schedules are called *serializable*. Serializability is the most commonly used correctness criterion for schedules. Of course, this cannot be checked for arbitrary programs. Therefore, concurrency control is done by choosing an appropriate abstraction for programs and enforcing sufficient conditions for serializability. The simplest abstraction consists of the sequence of lock and unlock operations performed by programs on data entities (such as tuples, relations, or pages). To see an example, consider two transactions:

$$\begin{array}{llll} T_1: & lockA & T_2: & lockA \\ & lockB & & unlockA \\ & unlockA & & lockB \\ & unlockB & & unlockB \end{array}$$

and the schedules

S_1	$T_1 : lockA$	S_2	$T_1 : lockA$	S_3	$T_2 : lockA$
	$T_1 : lockB$		$T_1 : lockB$		$T_2 : unlockA$
	$T_1 : unlockA$		$T_1 : unlockA$		$T_1 : lockA$
	$T_1 : unlockB$		$T_2 : lockA$		$T_1 : lockB$
	$T_2 : lockA$		$T_2 : unlockA$		$T_1 : unlockA$
	$T_2 : unlockA$		$T_1 : unlockB$		$T_1 : unlockB$
	$T_2 : lockB$		$T_2 : lockB$		$T_2 : lockB$
	$T_2 : unlockB$		$T_2 : unlockB$		$T_2 : unlockB$

Schedule S_1 is serial (it consists of the instructions of T_1 followed by those of T_2). Schedule S_2 is not serial, but is serializable (it is equivalent to T_1 followed by T_2). Schedule S_3 is neither serial, nor serializable (it is not equivalent to T_1 followed by T_2, nor to T_2 followed by T_1).

Generally, the more information is captured by the abstraction the more accurate the serializability test, and the higher its complexity. This highlights a basic tradeoff between the degree of concurrency allowed by a concurrency control mechanism, and its cost. The theory of concurrency control is presented in [**Pap86, BHG87**].

4. Theory of Query Languages

As discussed earlier, the main overlap between database theory and finite-model theory occurs in the theory of query languages. The common ground is well known to finite-model theorists. It consists of the investigation of languages that extend FO with recursion, such as Datalog (see [**MW88**]), the *fixpoint* queries (expressed by FO+LFP [**CH82**] and FO+IFP [**GS86**]) and the *while* queries[3] (expressed by FO+PFP, partial fixpoint logic [**AV91a**]). The connection between languages and complexity classes is of great interest to both fields. In particular, the existence of a language expressing precisely the queries in P remains the major open problem [**Gur84, Gur88**]. We omit here any further description of the common ground (see Section 6 for some remarks on the interaction with finite-model theory). Instead, we focus on certain aspects of the theory of query languages that illustrate the specificity of the database approach. The aspects we discuss are the following:

- safety and domain independence;
- the diversity of database query language paradigms;
- the impact of the data independence principle;
- complete languages; and,
- new notions of query complexity.

4.1. Safety and Domain Independence. The basis for the connection between finite-model theory and databases is that instances are essentially finite structures. However, there is a small difference between the two that gives rise to a large problem. Finite interpretations of first-order structures specify explicitly a finite domain, and formulas are evaluated relative to this domain. Database instances do not make explicit mention of any particular domain. So we really have two domains to contend with: the infinite domain **dom**, and the implicit finite domain of the

[3]The names *fixpoint* and *while* are used for historical reasons. The term *fixpoint* is derived from the fixpoint logics that express this class of queries, and the term *while* is derived from a language introduced in [**Cha81**] which expresses the *while* queries.

instance consisting of its active domain. This is not as abstract as it may seem. In a real database recording information about persons, a user would naturally distinguish between the potentially infinite set of all persons, and the finite set of persons mentioned in the database. This gives rise to an ambiguity. Suppose the user asks for all persons who are not students. Does he refer to all persons *in the active domain*, or to all persons? Obviously, if the interpretation is relative to **dom**, some queries will have infinite answers. This raises the problem of *safety*. An FO query is safe if its evaluation relative to **dom** is always finite. One can show that safe queries can be effectively evaluated, despite the fact that quantifiers range over **dom**. Not surprisingly, safety of FO queries is undecidable. However, there are syntactic conditions that guarantee safety and that cover all safe queries [**Ull82**]. However, this is not always possible for domains equipped with more predicates than just equality [**ST95**].

The obvious alternative is to evaluate queries relative to the active domain of the input instance. This raises a more subtle problem. Databases often contain a large number of relations. Some users have only partial information about the database. Suppose a user poses a query that only mentions some relation R. This is nonetheless evaluated against the active domain of the entire database. In general, the answer will depend on database relations not mentioned in the query, that the user may know nothing about. As a simple example, consider the beer drinker's database used earlier. Consider the sentence $\forall b\ (frequents(Joe, b))$ (Joe frequents all bars). This may be true if only bars occurring in *frequents* are considered. It may be false if there are additional bars elsewhere in the database, say in relation *serves*. This is disturbing. One would like the answer to be independent of the presence of unknown additional relations in the database. This is captured by the notion of *domain independence*. An FO query is domain independent if its evaluation is the same when taken relative to any two finite domains lying between the active domain and **dom**. For example, the query

$$\forall b\ [\exists d\ (\mathit{frequents}(d, b)) \rightarrow \mathit{frequents}(Joe, b)]$$

is domain independent. Intuitively, the syntax ensures that elements of the domain which do not occur in *frequents* are irrelevant, which implies domain independence. Note that domain independence implies safety, but the converse is false. Like safety, domain independence of FO queries is undecidable. There are syntactic restrictions that guarantee domain independence and cover all domain independent queries. The restrictions are in the spirit of the above example. They ensure that the range of all variables is explicitly given in the query. In the above, it is sufficient to consider only values of b and d occurring in *frequents*.

The notion of domain independence was introduced to the database community by [**Fag82, Mak81**]. Notions related to domain independence are found as early as [**Low15**] in the logic community. Surveys on domain independence and variations can be found in [**Kif88, GT91**].

4.2. Diversity of Database Language Paradigms. The standard relational database query languages have at their core some syntactic variation of FO. It is interesting to briefly consider their characteristics. One principle guiding the design of practical languages such as SQL is to make it very easy to express simple queries. The building blocks of SQL queries are essentially conjunctive queries, i.e.

FO queries of the form

$$\{x_1 : A_1, \ldots, x_n : A_n \mid \exists \mathbf{u}(L_1 \wedge \ldots \wedge L_k)\}$$

where L_i are of the form $R(x_1, \ldots, x_k)$ where R is a relation of arity k and each x_i, $1 \leq i \leq k$, is a variable or domain element. Furthermore, the SQL syntax for these building blocks ensures domain independence of the query. Arbitrary FO queries can be expressed as the composition of several conjunctive queries, and negation. In general, no explicit universal quantification is provided. For an example of an SQL query, the earlier query on the beer drinker's database "find all drinkers who frequent only bars serving Coors" is expressed in SQL as follows:

```
select   drinker
from     frequents
where    drinker not in
           (select   drinker
            from     frequents
            where    bar not in
                     (select bar from serves
                      where beer = Coors));
```

This corresponds to a decomposition of the query into this sequence of existentially quantified queries:

$$\begin{array}{lcl} P & = & \{b : bar \mid serves(b, Coors)\}; \\ Q & = & \{d : drinker \mid \exists b(frequents(d, b) \wedge \neg P(b)\}; \\ answer & = & \{d : drinker \mid \exists b(frequents(d, b)) \wedge \neg Q(d)\}. \end{array}$$

Syntax is a constant preoccupation in query language design. The quest for convenient, appealing languages that are likely to be accepted by a wide variety of users goes beyond FO, and is one of the reasons for the diversity of programming paradigms considered by database researchers. It turns out that logic is not always the preferred paradigm for a particular class of queries. Rather oddly, one might even say that databases have no particular allegiance to logic!

The inability of FO to express certain useful queries, such as connectivity of finite graphs, was noted early on by database theoreticians ([**Fag75**], and independently [**AU79**]). Such facts have led to the introduction of a variety of extensions of FO with recursion. Some of the proposed paradigms are illustrated next.

- **Logic:** Extensions of FO with logical flavor include the well known FO+IFP and FO+LFP, and the partial fixpoint logic FO+PFP. Recall that FO+LFP extends FO with an operator LFP that iterates a positive formula up to a fixpoint; FO+IFP extends FO with an operator IFP that iterates an arbitrary FO formula up to a fixpoint. Convergence is ensured by forcing the iteration to be cumulative (whence the name of inflationary fixpoint operator). And, FO+PFP extends FO with an operator PFP that iterates an arbitrary FO formula, and may or may not reach a fixpoint (whence the name of partial fixpoint operator).
- **Imperative programming:** Languages have variables that denote relations, assignment of FO queries to relation variables, and an explicit looping construct of the form **while change do** *body*. The *body* is iterated as long as there is a change to some relation. Assignments come in two flavors: inflationary and noninflationary. Inflationary assignment, denoted $R+ = \varphi$, is

cumulative: the result of φ is added to the content of R. Noninflationary assignment, denoted $R := \varphi$, is destructive: the old content of R not preserved. The language with inflationary semantics is called *while*$^+$ [**AV90**], and the language with noninflationary semantics *while* [**Cha81, CH82**]. The following *while* program computes the complement of the transitive closure of G:

> $T := G$;
> **while change do**
> **begin**
> $T := \{\langle x, y\rangle \mid T(x,y) \vee \exists z(G(x,z) \wedge T(z,y))\}$
> **end**
> $\overline{T} := \{\langle x, y\rangle \mid \neg T(x,y)\}$

- **Logic programming:** This uses a rule-based formalism. The language of choice in databases is Datalog$^\neg$, Datalog with negation. A Datalog$^\neg$ program is a set of rules of the form

$$A_0(t_0) \leftarrow A_1(t_1), \ldots, A_n(t_n)$$

where A_0 is a relation, each A_i with $i > 0$ is of the form R or $\neg R$ where R is a relation, the t_i are tuples of variables of appropriate arities, and each variable in t_0 occurs in some t_i where A_i is positive (of the form R), with $i > 0$. In a Datalog$^\neg$ rule as above, $A_0(t_0)$ is called the *head* and $A_1(t_1), \ldots, A_n(t_n)$ the *body* of the rule.

There are several versions of Datalog$^\neg$, differing in the semantics for negation. The main semantics for negation are the *stratified* semantics [**CH85, ABW88, Lif88, Gel86**] and the *well-founded* semantics [**GRS88, GRS91, Gel89, BF88, Prz89, Prz90**], which we do not describe here. The following is a Datalog$^\neg$ program which defines the complement of transitive closure under both semantics:

$$\begin{array}{l} T(x,y) \leftarrow G(x,y) \\ T(x,y) \leftarrow G(x,z), T(z,y) \\ \overline{T}(x,y) \leftarrow \neg T(x,y) \end{array}$$

Intuitively, the semantics of negation ensures that T is computed before the last rule is used to compute $\overline{T}$. The negation is taken relative to the finite active domain.

- **Production systems:** These are also rule-based languages, but with a procedural fixpoint semantics. One common semantics is to apply the rules in parallel as follows. The computation consists of a sequence of stages. At each stage, for each valuation of the variables that makes true the body of a rule, the head of the rule under that valuation is added to the database. The semantics of negation is straightforward: $\neg R(u)$ is true at a particular stage if $R(u)$ is not in the current state of the database. This is repeated until a fixpoint is reached. This semantics is referred to as the *inflationary fixpoint semantics* for Datalog$^\neg$ [**AV88, KP88**]. Note that the above program no longer computes the complement of transitive closure under the inflationary fixpoint semantics. Indeed, the last rule causes $\overline{T}$ to contain all edges after the first stage. Instead, the following program computes the complement of the transitive closure of G under inflationary fixpoint semantics (it is assumed that G is not empty):

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y) \\ T(x,y) & \leftarrow & G(x,z), T(z,y) \\ \textit{old-}T(x,y) & \leftarrow & T(x,y) \\ \textit{old-}T\textit{-except-final}(x,y) & \leftarrow & T(x,y), T(x',z'), T(z',y'), \neg T(x',y') \\ \overline{T}(x,y) & \leftarrow & \neg T(x,y), \textit{old-}T(x',y'), \\ & & \neg \textit{old-}T\textit{-except-final}(x',y'). \end{array}$$

Note that *old-T* follows the computation of T, but is one step behind it. The relation *old-T-except-final* is identical to *old-T*, but includes a clause which prevents it from firing when T has reached its last iteration. Thus, *old-T* and *old-T-except-final* differ only in the iteration after the transitive closure T reaches its final value. In the subsequent iteration, the program recognizes that the fixpoint has been reached, and fires the rule computing the complement $\overline{T}$.

- **Object-oriented programming:** Query languages for object-oriented databases are strongly influenced by object-oriented programming. In object-oriented databases, objects with common structure and behavior are grouped into *classes*. Programs called *methods* are attached to classes and can be applied to each object in the class. An abstract language modeling side-effect-free methods, called *method schemas*, has been proposed in [**AKRW92, HKR93**]. We further discuss some aspects of object-oriented database languages in Section 5.

The diversity of query language paradigms beyond FO may be disconcerting at first. It may appear that there is no hope for unity. However, there is good news. Despite the wide variations, many of the languages are equivalent. In fact, we are led back to familiar ground. Indeed, the *fixpoint* and *while* queries emerge as central. Among the languages described above, FO+IFP, FO+LFP, *while*$^+$, Datalog$^\neg$ (with either well-founded or fixpoint semantics), and method schemas are all equivalent and express the *fixpoint* queries [**GS86, AV88, Gel89, HKR93**]. The languages FO+PFP, *while*, and Datalog$^\neg$ augmented with the ability to delete previously inferred facts, are also equivalent and express the *while* queries [**AV91a**].

As discussed earlier, convenient syntax and semantics is a constant preoccupation in query language design. Logic is not always the formalism of choice. Among languages expressing the *fixpoint* queries, the rule-based languages are the most popular. Among those expressing *while*, imperative languages in the style of *while* are competing with rule-based languages in the style of Datalog$^\neg$ with fixpoint semantics, augmented with the ability to delete facts.

Coming up with the "right" language for a target class of queries is more than just a matter of syntactic sugaring. For example, Datalog$^\neg$ with inflationary fixpoint semantics is much simpler at first glance than the full FO+IFP, and their equivalence is nontrivial (and special to finite structures) [**AV88, AV91a**].

4.3. Data independence principle. As discussed earlier, the data independence principle is one of the defining characteristics of database systems. The separation of the logical and physical database levels has yielded a distinct point of view on several aspects of the theory of query languages.

Expressiveness results in database theory, as in finite-model theory, come in two main flavors: those that assume the presence of an order, and those that make no such assumption. In the context of databases, computation without order can be viewed as a mathematical metaphor for the data independence principle. The argument goes as follows. The database as viewed at the logical level is generally an unordered structure. At the physical level however, the constants are encoded and stored on some physical medium as sequences of bits. This induces an order among them. If the physical level cannot be accessed, this information cannot be used and no order is available. However, the order provided by the physical level can always be accessed by breaking the data independence principle.

The fact that the answer to a query is dependent only on information present at the logical level is captured by the notion of *genericity* [**AU79, CH80b, HY84**]. A query is generic if it commutes with isomorphisms of **dom**. Genericity basically says that the query is well defined at the logical level. This is almost trivial in the absence of several levels of abstraction. In the context of multiple levels of abstraction, genericity becomes much more meaningful. It says that, although the *evaluation* of the query uses physical-level information, the answer to the query may not depend on the additional information.

Genericity has important consequences for the expressiveness of languages and the complexity of queries. For example, the evenness query on a unary relation[4] is not expressible in seemingly powerful languages such as *while* [**Cha81**]. There is a mismatch between the classical notion of complexity and the difficulty of computation without order, arising from the fact that classical complexity is defined using machines that work on encodings of structures, rather than the structures themselves. The encodings are analogous to representations of data at the physical level of a database. Thus, abstraction is absent.

Relational machines. Computation without order is modeled by a device called a *relational machine*, operating directly on structures [**AV91b**]. (A closely related idea, of generalizing Turing machines to operate on general structures, goes back to [**Fri71**] and was investigated extensively in [**Lei89a, Lei89b**] for ordered structures.) A relational machine[5] is a Turing machine augmented with a *relational store*. The relational store consists of a set of relations of certain arities. Some of these relations are designated as input relations and some are designated as output relations. The *arity* of the machine is the maximal arity of the relations in its store. The tape of the machine is a work tape and is initially empty. Transitions depend on the current state, the content of the current tape cell, and a test of emptiness of a relation of the relational store. Transitions involve the following actions: move right or left on the tape, overwrite the current tape cell with some tape symbol, and replace a relation of the store with the result of a relational algebra operation on relations of the store. For example, the machine can have instructions such as:

> If the machine is in state s_3, the head is reading the symbol 1, and relation R_1 is empty, then change the state to s_4, replace the symbol 1 by 0, move the head to the right, and replace R_2 by $R_2 \cap R_3$.

Relational machines are closely related to a language very familiar to finite-model theorists: infinitary logic with finitely many variables, $L^{\omega}_{\infty\omega}$ [**Bar77**]. It

[4]Evenness is true iff the relation has an even number of elements.

[5]Relational machines were initially called *loosely coupled generic machines* in [**AV91b**]. The term *relational machine* is used in [**AVV92**] and subsequent papers.

turns out that in a reasonable sense, relational machines are precisely the effective fragment of $L^{\omega}_{\infty\omega}$ [**AVV95**].

Defining complexity classes based on relational machines poses the following problem. For Turing machines, the most natural measure of complexity is in terms of the size of the input. This measure is not the most natural one for relational machines, since such machines cannot measure the size of their input. Indeed, a relational machine of arity k cannot distinguish between two structures that are equivalent relative to $L^{k}_{\infty\omega}$, although these may have widely different sizes. In fact, relational machines have a limited *discerning* power, i.e., the power to distinguish between different pieces of their input, since they are able to manipulate their input only relationally. This suggests that it is natural to measure the size of the input to a relational machine with respect to the discerning power of the machine. The new measure, based on the number of equivalence classes of k-tuples with respect to equivalence relative to $L^{k}_{\infty\omega}$, gives rise to a new notion of computational complexity, called *relational complexity* [**AV91b, AV95, AVV92**]. This results in classes such as P_r (relational polynomial time) and NP_r (relational nondeterministic polynomial time). It turns out that there is a nice match between relational complexity classes and query languages. It is shown in [**AV91b, AV95**] that P_r = *fixpoint* and PSPACE_r = *while*. This is extended in [**AVV92**] to variations of fixpoint logic and the relational complexity classes NP_r, $\mathrm{EXPTIME}_r$, and $\mathrm{EXPSPACE}_r$. Furthermore, the containment structure of these relational complexity classes is the same as that of their classical counterparts. This yields results relating languages to *classical* complexity classes, such as: *fixpoint* = *while* iff P = PSPACE [**AV91b, AV95**].

Nondeterministic query languages. It is well known that the presence of order has dramatic impact on the expressiveness of languages. For example, *fixpoint* = P and *while* = PSPACE on ordered structures [**Imm86, Var82**]. Now recall our view of order as a metaphor for accessing physical level information in a database. Suppose that a query language breaks the data independence principle by making reference to physical level information. The answer will generally depend on such information. Viewed at the logical level, the query then appears to be nondeterministic: it returns different answers when repeatedly posed against the same logical-level database.

This suggest an alternative to breaking the data independence principle to circumvent limitations in expressive power: making nondeterminism a first-class construct in the language. The expected benefits in terms of expressiveness are similar to the presence of an order.

We illustrate one way in which a deterministic language can be augmented with a nondeterministic construct. Consider FO+IFP. We augment it with a nondeterministic construct, called the *witness operator*, denoted W. Intuitively, $W\vec{y}\phi(\vec{x}, \vec{y})$ indicates that one "witness" $\vec{y}_x$ is chosen for each $\vec{x}$ satisfying $\exists\ \vec{y}\ \phi(\vec{x}, \vec{y})$. For example, if R= {[1,1],[1,2],[2,3]}, the formula $Wy(R(x,y))$ denotes the set of two relations {[1,1],[2,3]} and {[1,2],[2,3]}. More precisely, for each formula $\phi(\vec{x}, \vec{y}))$ (where $\vec{x}$ and $\vec{y}$ are vectors of the variables which are free in ϕ), $W\vec{y}\phi(\vec{x}, \vec{y}))$ is a formula (where the $\vec{y}$ remain free) defining the *set* of relations[6] $\mathbf{I}$ such that for some $\mathbf{J}$ defined by ϕ: $\mathbf{I} \subseteq \mathbf{J}$; and for each $\vec{x}$ for which $[\vec{x}, \vec{y}]$ is in $\mathbf{J}$ for some $\vec{y}$, there

[6] Each relation in the set represents one possible answer.

exists a *unique* $\vec{y}_x$ such that $[\vec{x}, \vec{y}_x]$ is in **I**. FO+IFP augmented with the witness operator is denoted FO+IFP+W. This language defines both deterministic and nondeterministic queries, and it is undecidable whether a given FO+IFP+W query is deterministic. Let *det*(FO+IFP+W) denote the set of queries of FO+IFP+W that are deterministic. It was shown in [**AV91a, ASV90**] that *det*(FO+IFP+W) = P. This confirms the intuition that nondeterminism is an alternative to the order assumption. The connection between nondeterminism and order emerges very naturally in the context of the data independence principle.

4.4. Complete languages. A query language is said to be *complete* if it expresses precisely the computable and generic queries [**CH80a**]. The quest for a complete query language was an early preoccupation in database theory.

Note that all query languages we have considered so far have complexity within PSPACE. The most powerful of these is *while*. To break the PSPACE barrier, one would be tempted to enrich *while* with full computing power *outside* the database. This can be done by augmenting *while* with integer variables, *increment* and *decrement* instructions, and a loop construct **while** $i > 0$ **do**. This yields a language denoted $while_N$ [**Cha81**]. Note that this captures the computational style of practical relational languages like C+SQL where an FO language (SQL) is embedded in a full programming language (C). This would seem to provide the simplest "cure" to the computational limitations of the languages exhibited so far. However, $while_N$ is not complete. In fact, it turns out to be equivalent to relational machines, and so cannot express queries such as evenness. However, $while_N$ *is* complete on ordered databases.

We next discuss other possibilities. Consider why *while* cannot go beyond PSPACE: it uses, throughout the computation, (i) only values from the input, and (ii) relations of fixed arity. The addition of integers as in $while_N$ is one way to break the space barrier. Another is to relax (i) or (ii). Relaxing (i) is done by allowing the creation of new values, not present in the input. Relaxing (ii) yields an extension of *while* with *untyped algebra*, i.e. an algebra of relations with variable arities. The latter approach was historically first to produce a complete query language [**CH80a**].

We briefly describe (i), i.e. the extension of *while* with value creation, denoted $while_{new}$. The language *while* is augmented with a new instruction $R := new(S)$, where R and S are relational variables and $arity(R) = arity(S) + 1$. The semantics is the following. Relation R is obtained by extending each tuple of S by one distinct new value from **dom**, not occurring in the input, the current state, or in the program. For example, if the value of S is the relation in Figure 2 then R is of the form shown in the same figure. The values α, β, γ are distinct new values in **dom**.

Note that the *new* construct is, strictly speaking, nondeterministic. Indeed, the new values are arbitrary, so several possible outcomes are possible depending on the choice of values. However, the different outcomes differ *only* in the choice of new values. If the final answer to the query contains no new values (which can be guaranteed by a simple syntactic restriction), the query is deterministic. It is shown in [**AV90**] that $while_{new}$ with the above mentioned syntactic restriction is complete.

S

a	b
a	c
c	a

R

a	b	α
a	c	β
c	a	γ

FIGURE 2. An application of *new*.

4.5. New notions of query complexity. The database scenario has led to several new approaches to query complexity. We already discussed *relational complexity*, which takes into account the abstraction inherent in the data independence principle. We mention two other points of view.

Although research in database theory often focuses on one particular level of abstraction in the database architecture, some of the most interesting questions concern the interaction of the different levels. These questions are also some of the hardest. The classical definition of query complexity does not capture various factors present in a real database that may nonetheless have tremendous impact on the difficulty of query evaluation. Some research has tried to remedy this. For example, Ullman and Yannakakis [**UY90**] consider a notion of complexity based on the number of I/O operations required to evaluate the query, and study the I/O complexity of transitive closure. Other graph algorithms in the context of external memory are considered in [**CGG**$^+$**95**].

Another approach to complexity arises from the uniform nature of query processing. Classical complexity considers all algorithms for solving a given problem. But in the case of evaluating a query, the corresponding algorithm has to be generated by a compiler from the specification of the query in a query language. The compiler can rarely improve by more than a polynomial amount on the natural evaluation which follows from the specification of the query in the language. This has given rise to work considering the complexity of queries relative to a language. For example, although the *evenness* query can be expressed in $while_{new}$, any program expressing it takes exponential space in its natural evaluation [**AV91a**]. Intuitively, the *evenness* query is computed in $while_{new}$ by first producing all orderings of the active domain (each of which is identified by a distinct new value), then computing *evenness* for each of the orderings. This uses exponential space, since exponentially many orderings are generated. The lower bound follows from the observations that (i) the PSPACE fragment of $while_{new}$ collapses to FO on unary relation inputs, and (ii) any $while_{new}$ program not in PSPACE must use exponential space. Results of similar flavor are obtained in [**Yan86**] for queries in the "weak universal relation" model. Lower bounds for expressing certain queries in Datalog are shown in [**Afr94**]. Exponential lower bounds for expressing transitive closure in a complex value algebra are shown in [**SP94**].

5. The Frontier: New Applications, New Models

Most of the discussion so far was set within the classical relational database framework. In recent years, the database field has gone beyond the classical framework, bringing into play a combination of formalisms and tools from artificial intelligence, programming languages, logic programming, information retrieval, etc. We briefly discuss a few of these developments, and the flavor of the accompanying theory. We focus on object-oriented databases, databases with incomplete information, and multimedia databases.

5.1. Object-oriented databases. Object-oriented databases provide a data model that is considerably richer than the relational model. It is strongly influenced by semantic networks in AI and by object-oriented programming. Roughly speaking, the world consists of things called *objects*. Objects have structure and behavior. The structure consists of attributes whose values are sets or tuples of other objects or domain elements. The behavior consists of programs which can be run on the object, called *methods*. Objects with the same structure and behavior are grouped together in *classes*. There is a subclass hierarchy among classes; for example, it may be that *student* is a subclass of *person*. A subclass *inherits* the structure and behavior of the superclass. To illustrate, this means that all attributes that apply to the class *person* also apply to the class *student* and any method that can be run on objects of class *person* can also be run on objects of class *student*.

Many of the features of object-oriented databases are programming conveniences. However, some new fundamental problems arise. We discuss two: complex values, and new notions of complete languages.

Complex values. As we have seen, in object-oriented databases attribute values are no longer atomic values. They can have complex structure, obtained by repeated applications of tuple and set constructors. This gives rise to so called *complex values*. Figure 3 gives an example of a complex-value relation. As in relational databases, the relation schema specifies a set of attributes, and an instance consists of a finite set of tuples providing values for each attribute. However, those values are now complex rather than atomic.

In order to manipulate complex values, extensions of the relational calculus have been introduced, together with corresponding algebraizations. Note that inputs and outputs to queries may have complex structure, as can intermediate results. One notable special case is that where inputs and outputs are classical "flat" relational instances but the query (say, in the calculus) uses variables over complex values. This is analogous to higher-order logics. For example, the complex value calculus with flat input and output, but using variables ranging over *sets* of tuples, is essentially second-order logic. Thus, one use of complex values in query languages is to increase expressiveness. However, in object-oriented databases the primary goal is to manipulate complex values in a tractable way. This has given rise to numerous proposals to restrict the complex value calculus and algebra so as to guarantee tractability. The main idea is to restrict how new sets are constructed. One way to ensure this is by syntactic restrictions, in both the calculus and the algebra. For example, the *powerset* operator, present in the algebra, is forbidden in various restrictions. An example of a tractable algebraic operation that constructs sets is *nesting*. Suppose R is a relation with attributes A, B. The result of $nest_A(R)$

Director	*Movies* — *Title*	*Movies* — *Actors*
Hitchcock	The Trouble with Harry	John Forsythe Edmund Gwenn Shirley MacLaine Alfred Hitchcock
	The Birds	Tippi Hedren Rod Taylor Suzanne Pleshette Alfred Hitchock
	Psycho	Anthony Perkins Janet Leigh Alfred Hitchcock
Bergman	Cries and Whispers	Harriet Andersson Kari Sylwan Ingrid Thulin Liv Ullman
	The Seventh Seal	Max von Sydow Gunnar Björnstrand Bengt Ekerot Mils Poppe

FIGURE 3. A relation with complex values

is the relation with attributes A, B consisting of the tuples

$$\{\langle a, X\rangle \mid a \in \pi_A(R), X = \{b \mid \langle a, b\rangle \in R\}\}.$$

Thus, the values of the attribute B are now sets. There are syntactic restrictions on the calculus and algebra that guarantee that all queries can be evaluated in polynomial-time.

The earliest proposal to introduce complex values appears to be [**Mak77**]. Equivalent complex value calculus and algebra are proposed in [**KV84, KV93, AB95**]. Tractable restrictions are considered by many authors, including [**JS82, TF86, RKS86, AB95, PG88, GV91**].

Extended notions of completeness. Classical relational queries return answers built from the elements of **dom** present in the input. In contrast, queries in object-oriented databases often return structures that contain newly constructed objects. These are represented by new "identifiers" (values in **dom**), not previously in the database. This extended notion of query yields an extended notion of completeness of a query language. One might think that a language such as $while_{new}$, which allows for value creation and is complete in the classical sense, would also be complete in the extended sense. It turns out that this is not the case. This is illustrated by the following example from [**AK89**]. Consider a query whose input consists of a graph with vertices $\{a, b\}$ and no edges, and whose output consists of a graph with vertices $\{a, b\}$, additional new vertices $\{\psi_0, \psi_1, \psi_2, \psi_3\}$, and with edges as in Figure 4.

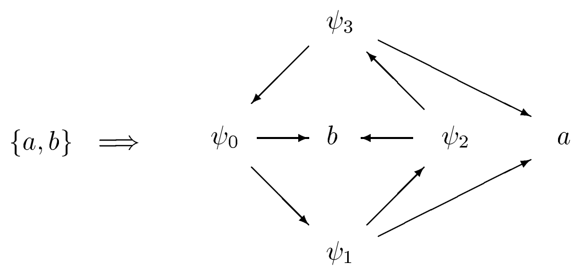

FIGURE 4. A query not expressible in $while_{new}$.

It turns out that this query is not expressible in $while_{new}$, and even in more powerful languages using complex values. To understand why, we present an elegant necessary condition in order for queries to be expressible in $while_{new}$ (this becomes also sufficient for certain more powerful languages) [**dBGAG92**]. For an instance K, let $Aut(K)$ denote the set of automorphisms of K. For a pair K, K' of instances, $Aut(\langle K, K'\rangle)$ denotes the bijections on $\mathbf{adom}(K \cup K')$ that are automorphisms of both K and K'. An *extension homomorphism* from $Aut(K)$ to $Aut(\langle K, K'\rangle)$ is a mapping $h : Aut(K) \rightarrow Aut(\langle K, K'\rangle)$ such that for each $\tau, \mu \in Aut(K)$,

(i) τ and $h(\tau)$ coincide on K;
(ii) $h(\tau \circ \mu) = h(\tau) \circ h(\mu)$; and,
(iii) $h(\mathbf{id}_K) = \mathbf{id}_{\langle K,K'\rangle}$.

The necessary condition for a query to be expressible in $while_{new}$ is the following:

For each input-output pair $\langle I, J\rangle$ in q there exists an extension homomorphism from $Aut(I)$ to $Aut(\langle I, J\rangle)$.

This condition can be used to show that the above query is not expressible in $while_{new}$ as follows. Let $\langle I, J\rangle$ be the input-output pair of Figure 4. Suppose there is a $while_{new}$ query which produces J on input I. By the necessary condition above, there is an extension homomorphism h from $Aut(I)$ to $Aut(\langle I, J\rangle)$. Let μ be the automorphism of I exchanging a and b. Note that $\mu^{-1} = \mu$, so $\mu \circ \mu = \mathbf{id}_I$. Consider $h(\mu)(\psi_0)$. Since $\langle \psi_0, b\rangle$ is an edge in J and $h(\mu)$ is an automorphism of J, $\langle h(\mu)(\psi_0), h(\mu)(b)\rangle$ must also be an edge in J. Since $h(\mu)(b) = \mu(b) = a$ it follows that, $h(\mu)(\psi_0) \in \{\psi_1, \psi_3\}$. Suppose $h(\mu)(\psi_0) = \psi_1$ (the other case is similar). Then clearly, $h(\mu)(\psi_1) = \psi_2$. Consider now $h(\mu \circ \mu)(\psi_0)$. We have, on one hand,

$$\begin{array}{rcl} h(\mu \circ \mu)(\psi_0) & = & (h(\mu) \circ h(\mu))(\psi_0) \\ & = & h(\mu)(\psi_1) \\ & = & \psi_2 \end{array}$$

and on the other hand

$$\begin{array}{rcl} h(\mu \circ \mu)(\psi_0) & = & h(\mathbf{id}_I)(\psi_0) \\ & = & \mathbf{id}_{\langle I,J\rangle}(\psi_0) \\ & = & \psi_0, \end{array}$$

which is a contradiction, since $\psi_0 \neq \psi_2$. So, q is not expressible in $while_{new}$.

Several languages are known to be complete with respect to the extended notion. However, they all use primitive constructs that either have high complexity

[**AK89**] or are not very natural [**DV93**]. No complete language is known that uses only tractable and natural primitives.

5.2. Incomplete information. Classical databases are completely determined finite structures. In reality, we often have to deal with incomplete information. This can be of many kinds. There can be missing information, as in: "John bought a car but I don't know which one". In the case of John's car, the information exists but is not available. In other cases, some attributes may be relevant only to some tuples, and irrelevant to others. Alice is single, so the spouse field is irrelevant in her case. Or, some information may be imprecise: "Heather lives in a large and cheap apartment" where the values of *large* and *cheap* are fuzzy.

We briefly describe simple forms of incompleteness, represented by "null values". These partially specify the state of the world. Instead of completely identifying one state of the world, its content is compatible with many "possible worlds". In this spirit, we define an *incomplete database* simply as a set of possible worlds, i.e. a set of instances. What is actually stored is a *representation* T of an incomplete database $rep(T)$. For example, *conditional tables* allow for tuples with variables subject to conditions attached to the table or to each row [**IL84, Gra84**]. A conditional table T represents the set of tables obtained by assigning values to the variables in ways that satisfy the conditions. Thus, the following conditional table represents the information that Sally is taking Math or CS (but not both), and another course; Alice takes Biology if Sally takes Math, and Math or Physics (but not both) if Sally takes Physics.

$Student$	$Course$	
	$(x \neq Math) \wedge (x \neq CS)$	
$Sally$	$Math$	$(z = 0)$
$Sally$	CS	$(z \neq 0)$
$Sally$	x	
$Alice$	$Biology$	$(z = 0)$
$Alice$	$Math$	$(x = Physics) \wedge (t = 0)$
$Alice$	$Physics$	$(x = Physics) \wedge (t \neq 0)$

Choosing appropriate representations is a central issue. The main problem is how to answer queries on such databases. In order for a representation system for incomplete information to be adequate in the context of a query language, it must also be capable of representing *answers* to queries. This leads to a desirable closure property of representations of incomplete information with respect to query languages. Consider some particular representation system. Such a system involves a language for describing representations and a mapping rep that associates a set of instances to each representation. Suppose that we are interested in a particular query language L (e.g., FO). We would like to always be capable of representing the result of a query in the same system. More precisely, for each representation T and query q, there should exist a representation $\overline{q}(T)$ computable from T such that:

$$rep(\overline{q}(T)) = q(rep(T)).$$

In other words, $\overline{q}(T)$ represents the possible answers of q, i.e. $\{q(I) \mid I \in rep(T)\}$.

If some representation system τ has the property described above for a query language L, τ is called a *strong representation system* for L. Clearly, we are particularly interested in strong representation systems for FO. It turns out that conditional tables form a strong representation system for FO [**IL84**]. In fact, it remains a strong representation system for the *fixpoint* and *while* queries. It is easy to see that tables with variables but no conditions do *not* form a strong representation system for FO. Less obvious is the fact that FO does not form a strong representation system for FO.

5.3. Multimedia databases. Recent database applications require the manipulation of nontraditional information stored on various media: images, sound, text, etc. The database then has to handle hybrid data. Part of it is a traditional finite instance, processed by the well-known means provided by relational databases. Another part has to be processed using techniques specific to the particular media, such as image processing and string matching algorithms. This raises a host of interesting new questions. We consider two of them here[7]

1. the finite representation of infinite information, and
2. the representation of media-specific information in classical relational form.

We illustrate the two problems using the area of spatial databases. These handle information about regions in space. Such information is conceptually infinite, but must be finitely represented. One solution is provided by *constraint databases* [**KKR90**]. These introduce an extension of the relational model where tuples are replaced by constraints over points in space, specified by boolean combinations of polynomial inequalities with integer coefficients. Then each tuple specifies a region in space. Regions specified in this fashion are called *semi-algebraic*. Suppose we wish to answer queries on constraint databases. The problem here is similar to that encountered in the previous section on incomplete information. A representation system for spatial information is adequate relative to a query language if it is closed under queries in the language. It turns out that constraint databases have this property for FO and for some extensions with recursion [**KKR90**]. The proof uses quantifier elimination techniques. See [**Kan95**] for a survey of the area.

Constraint databases provide an approximate representation for spatial regions, which is adequate for many purposes. Although they are finite representations, they are not traditional finite relational structures. (Of course, they could be encoded as such, but this is rather artificial.) A natural problem arises: are there useful situations when the relevant information can be represented as a classical finite relational structure? The answer is positive. Suppose we are interested in topological queries on spatial regions, that is queries which are invariant under homeomorphisms. These are particularly relevant to applications such as geographic information systems. It it shown in [**PSV96**] that topological information about semi-algebraic regions can be precisely and naturally described using a finite structure. (A similar invariant is exhibited in [**KPdB95**] for isotopy-invariant information.) Such a structure acts as an *invariant* characterizing a class of topologically equivalent spatial instances. It is an abstraction capturing exactly the topological properties of a set of regions. For semi-algebraic regions, the invariant can be computed in polynomial time (and NC). Moreover, once this structure is computed, topological queries can be answered by classical database queries posed against that

[7] For other work on the foundations of multi-media database systems, see [**Fag96, MS**].

structure, of complexity polynomially related to the original query. This provides a bridge between the spatial and classical database domains.

Such techniques have been used in connection with other types of information, such as text [**CM95**]. The approach is the same: for a target class of queries, the information needed to answer these queries is extracted from the rough data and summarized as a classical finite relational structure. Note that such techniques are not foreign to finite-model theory. The invariants describing equivalence classes of structures with respect to $L^k_{\infty\omega}$ serve a similar purpose [**AV95, DLW95**]. They summarize precisely the information needed to answer $L^k_{\infty\omega}$ queries on those structures. Since these invariants are ordered, they provide a bridge between computation on arbitrary structures and computation on ordered structures.

6. Interaction with Finite Model Theory

The interaction between databases and finite-model theory has been a fruitful one. Most of the common ground lies in the theory of query languages. Many of the questions on expressiveness and complexity are shared by the two fields. Fmt has answered some of the questions, and short of that has produced a sophisticated array of tools for approaching them (games, 0-1 laws, etc.). However, it is important to keep in mind that even here the match between finite-model theory and database theory has its rough edges.

We mentioned the need for accessible, convincing syntax in database languages. Some languages proposed in finite-model theory, very appealing from an expressiveness viewpoint, need careful tuning in order to become palatable to database people. As discussed in Section 4.2, fixpoint logics such as FO+IFP and FO+PFP have not been adopted in databases, most likely because of their unappealing syntax. Instead, equivalent rule-based languages were preferred. These have the advantage of simple syntax with no explicit fixpoint operator, and the ability to express simultaneous induction in a very natural way. As another example, consider *fixpoint* extended with counting, as introduced in [**CFI89, GO93**]. The semantics bounds the values of the integers to the size of the active domain of the database. This is however quite inconvenient in practical languages, and raises issues akin to domain independence. The connection with languages more in tune to the database style is not obvious. Before this connection is clarified, it seems unlikely that the theory developed around *fixpoint* with counting will acquire in databases the same prominent role it has acquired in finite-model theory.

Similar remarks apply to certain expressiveness results. As a case in point, consider some of the recent work [**Hel92, Daw93, DH95, Daw95**], studying whether FO+LFP can be augmented with finitely many generalized quantifiers to yield a language expressing P. It would be tempting to assimilate this to the following natural question in databases: is it possible to obtain a language for P by augmenting *fixpoint* with some finite set of additional programming constructs? However, the analogy is not that straightforward. For example, a *counting* construct corresponds to infinitely many generalized quantifiers. Further clarification of the connection between generalized quantifiers and programming constructs in database languages is needed. This would help understand the impact of the elegant results on generalized quantifiers to the database domain.

The shape of problems produced by the database scenario does not always fit the taxonomy developed by finite-model theory. For example, the class of sentences corresponding to fds+incds is not subsumed by any of the classes for which undecidability of finite implication has been known in finite-model theory [**BGG96**]; undecidability had to be proven from scratch.

However, results and formalisms from finite-model theory are sometimes unexpectedly and beautifully reinforced by databases. From the apparent cacophony of query language paradigms, the *fixpoint* and *while* queries emerge once again as fundamental, robust classes of queries. Even the newer, unsettled areas of databases yield surprising connections. A class of queries and constraints in object-oriented databases, expressed by so called *path expressions*, turns out to be subsumed by FO^2 and inherits its nice properties [**LSW**]. A constraint expressed by path expressions might be $person.mother \cap person.father = \emptyset$, stating that the set of objects reachable from *persons* by following the *mother* attribute is disjoint from the set reached from *persons* by following the *father* attribute. This constraint is expressible in FO^2. Path expressions with recursion (where paths are described by regular expressions) are subsumed by *fixpoint* with 2 variables, which brings up the open question of its decidability. The notion of *deep equality*, also arising in object-oriented databases, turns out to provide possibly the most convincing example of a query separating stratified Datalog$^\neg$ and *fixpoint* [**AdB95**]. The query used in the original proof involved rather complex game trees [**Kol91**].

Occasionally, questions and results originating in database theory have contributed to finite-model theory. The very concept of query was first articulated in database theory [**CH80b**] (although this is closely related to generalized quantifiers). The question of the recursive enumeration of the queries in P again originated in database theory [**CH82**]. As discussed in Section 4.3, the abstraction inherent in the data independence principle has motivated the study of relational machines and relational complexity, which has yielded results relating the inclusion structure of fixpoint logics to that of classical complexity classes.

7. Conclusion

The aim of this paper is to give finite-model theorists a flavor of the database field. We avoided covering the well-known common ground of database theory and finite-model theory, and focused instead on what is specific to databases. Rather than providing a list of database questions that finite-model theorists could work on, we attempted to show that the area of databases is a live, continuous source of interesting problems for finite-model theory. The best way to take advantage of this source is to become directly involved with it, to keep abreast of new developments in the database scenario, and to avoid relying solely on second-hand accounts.

There are several obstacles that a logician is likely to face in becoming involved with an area such as databases. Many logicians live in splendid isolation. They are used to intellectual rigor, clearly formulated ideas and very stable formalism. In contrast, the most interesting ideas in databases often arise in a shift of the scenario and start out in messy, half-baked form. This is nonetheless where the richness lies. There is also a tremendous diversity of formalism. But then again, this brings along its own set of questions, and a healthy confrontation and marriage of paradigms.

There are also barriers of temperament and style. Much of database research is driven by market forces and funding agencies, rather than by intellectual necessity. There is a certain amount of hard selling being done, an inflation of flashy, unnecessary terms. Some database researchers even wear ties. This goes against the grain of many logicians.

But logicians can take advantage of the richness and vitality of the database field and remain logicians. In contemplating the possibility, it may be useful to remember that others have been faced with similar predicaments, only more daunting [**Jun79**]:

> *"The alchemists sought their* prima materia *in dirt, earth, chaos, the most despised substances found in filth, from which it was hoped that the mystic figure would emerge."*

Acknowledgement The author wishes to thank Serge Abiteboul, Anuj Dawar, Ron Fagin, Neil Immerman, and Phokion Kolaitis for carefully reading the manuscript and providing numerous insightful comments. Special thanks to Neil Immerman and Phokion Kolaitis for instigating this paper.

References

[AB95] S. Abiteboul and C. Beeri, *The power of languages for the manipulation of complex values*, VLDB Journal **4(4)** (1995), 727–794.

[ABW88] K.R. Apt, H. Blair, and A. Walker, *Towards a theory of declarative knowledge*, Foundations of Deductive Databases and Logic Programming (Los Altos, CA) (J. Minker, ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 89–148.

[AdB95] S. Abiteboul and J. Van den Bussche, *Deep equality revisited*, Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD), Springer Verlag, Lecture Notes in Computer Science 1013, 1995, pp. 213–228.

[Afr94] F. Afrati, *Bounded arity Datalog($\neq$) queries on graphs*, Proc. ACM Symp. on Principles of Database Systems, 1994, pp. 97–106.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*, Addison-Wesley, Reading, MA, 1995.

[AK89] S. Abiteboul and P. C. Kanellakis, *Object identity as a query language primitive*, Proc. ACM SIGMOD Symp. on the Management of Data, 1989, To appear in *J. ACM*, pp. 159–173.

[AKRW92] S. Abiteboul, P. Kanellakis, S. Ramaswamy, and E. Waller, *Method schemas*, Tech. Report CS-92-33, Brown University, 1992, (An earlier version appeared in Proceedings 9th ACM PODS, 1990).

[ASU79] A. V. Aho, Y. Sagiv, and J. D. Ullman, *Equivalence of relational expressions*, SIAM J. on Computing **8(2)** (1979), 218–246.

[ASV90] S. Abiteboul, E. Simon, and V. Vianu, *Non-deterministic languages to express deterministic transformations*, Proc. ACM Symp. on Principles of Database Systems, 1990, pp. 218–229.

[AU79] A. V. Aho and J. D. Ullman, *Universality of data retrieval languages*, Proc. ACM Symp. on Principles of Programming Languages, 1979, pp. 110–117.

[AV88] S. Abiteboul and V. Vianu, *Procedural and declarative database update languages*, Proc. ACM Symp. on Principles of Database Systems, 1988, pp. 240–250.

[AV90] S. Abiteboul and V. Vianu, *Procedural languages for database queries and updates*, Journal of Computer and System Sciences **41** (1990), 181–229.

[AV91a] S. Abiteboul and V. Vianu, *Datalog extensions for database queries and updates*, Journal of Computer and System Sciences **43** (1991), 62–124.

[AV91b] S. Abiteboul and V. Vianu, *Generic computation and its complexity*, Proc. ACM SIGACT Symp. on the Theory of Computing, 1991, pp. 209–219.

[AV95] S. Abiteboul and V. Vianu, *Computing with first-order logic*, Journal of Computer and System Sciences **50** (1995), no. 2, 309–335.

[AVV92] S. Abiteboul, M. Y. Vardi, and V. Vianu, *Fixpoint logics, relational machines, and computational complexity*, Conf. on Structure in Complexity Theory, 1992, pp. 156–168.

[AVV95] S. Abiteboul, M. Y. Vardi, and V. Vianu, *Computing with infinitary logic*, Theoretical Computer Science **149** (1995), no. 1, 101–128.

[Bar77] J. Barwise, *On Moschovakis closure ordinals*, J. of Symbolic Logic **42** (1977), 292–296.

[BB79] C. Beeri and P. A. Bernstein, *Computational problems related to the design of normal form relational schemas*, ACM Trans. on Database Systems **4** (1979), no. 1, 30–59.

[BC79] O.P. Buneman and G.K. Clemons, *Efficiently monitoring relational databases*, ACM Trans. on Database Systems **4** (1979), no. 3, 368–382.

[BF88] N. Biboit and C. Froidevaux, *General logic databases and programs: Default logic semantics and stratification*, Tech. report, LRI, Université de Paris-Sud, Orsay, 1988, To appear in *J. Information and Computation*.

[BGG96] E. Borger, E. Graedel, and Y. Gurevich, *The classical decision problem*, Springer-Verlag, Berlin, 1996.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley, 1987.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa, *Efficiently updating materialized views*, Proc. ACM SIGMOD Symp. on the Management of Data, 1986, pp. 61–71.

[BS81] F. Bancilhon and N. Spyratos, *Update semantics of relational views*, ACM Trans. on Database Systems **6(4)** (1981), 557–575.

[BU77] R. Bayer and K. Unterauer, *Prefix b-trees*, ACM Trans. on Database Systems **2** (1977), no. 1, 11–26.

[BV81] C. Beeri and M. Y. Vardi, *The implication problem for data dependencies*, Proc. Intl. Conf. on Algorithms, Languages and Programming, 1981, Springer-Verlag LNCS 115, pp. 73–85.

[C+76] D. D. Chamberlin et al., *Sequel 2: a unified approach to data definition, manipulation and control*, IBM J. Research and Development **20(6)** (1976), 560–575.

[CFI89] J. Cai, M. Fürer, , and N. Immerman, *An optimal lower bound on the number of variables for graph identification*, IEEE Conf. on Foundations of Computer Science, 1989, pp. 612–617.

[CFP84] M. A. Casanova, R. Fagin, and C. H. Papadimitriou, *Inclusion dependencies and their interaction with functional dependencies*, Journal of Computer and System Sciences **28** (1984), no. 1, 29–59.

[CGG+95] Y.J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamasia, D.E. Vengroff, and J.S. Vitter, *External-memory graph algorithms*, Proc. ACM Symp. on Principles of Database Systems, 1995, pp. 1–10.

[CH80a] A. K. Chandra and D. Harel, *Structure and complexity of relational queries*, IEEE Conf. on Foundations of Computer Science, 1980, pp. 333–347.

[CH80b] A.K. Chandra and D. Harel, *Computable queries for relational data bases*, Journal of Computer and System Sciences **21(2)** (1980), 156–178.

[CH82] A.K. Chandra and D. Harel, *Structure and complexity of relational queries*, Journal of Computer and System Sciences **25(1)** (1982), 99–128.

[CH85] A.K. Chandra and D. Harel, *Horn clause queries and generalizations*, J. Logic Programming **2(1)** (1985), 1–15.

[Cha81] A. K. Chandra, *Programming primitives for database languages*, Proc. ACM Symp. on Principles of Programming Languages, 1981, pp. 50–62.

[Cho92] J. Chomicki, *History-less checking of dynamic integrity constraints*, Proc. IEEE Intl. Conf. on Data Engineering, 1992.

[Cho94] J. Chomicki, *Temporal query languages: A survey*, Proc. 1st Intl. Conf. on Temporal Logic, 1994.

[CK85] S. S. Cosmadakis and P. C. Kanellakis, *Equational theories and database constraints*, Proc. ACM SIGACT Symp. on the Theory of Computing, 1985, pp. 73–284.

[CM77] A. K. Chandra and P. M. Merlin, *Optimal implementation on conjunctive queries in relational data bases*, Proc. ACM SIGACT Symp. on the Theory of Computing, 1977, pp. 77–90.

[CM95] M.P. Consens and T. Milo, *Algebras for querying text regions*, Proc. ACM Symp. on Principles of Database Systems, 1995, pp. 11–22.

[Cod70] E. F. Codd, *A relational model of data for large shared data banks*, Comm. of the ACM **13(6)** (1970), 377–387.

[Cod72a] E.F. Codd, *Further normalization of the data base relational model*, Courant Computer Science Symposium 6: Data Base Systems (R. Rustin, ed.), Prentice-Hall, 1972, pp. 33–64.

[Cod72b] E.F. Codd, *Relational completeness of database sublanguages*, Data Base Systems (R. Rustin, ed.), Prentice-Hall, 1972, pp. 65–98.

[Cod87] T. Codd, *Relational database: a practical foundation for productivity*, ACM Turing Award Lectures, ACM Press, 1987, pp. 391–410.

[CT91] R. Cohen and R. Tamassia, *Dynamic expression trees and their applications*, Proc. 2nd ACM-SIAM SODA, 1991.

[CV85] A. K. Chandra and M. Y. Vardi, *The implication problem for functional and inclusion dependencies is undecidable*, SIAM J. on Computing **14(3)** (1985), 671–677.

[Dat81] C. J. Date, *Referential integrity*, Proc. of Intl. Conf. on Very Large Data Bases, 1981, pp. 2–12.

[Daw93] A. Dawar, *Feasible computation through model theory*, Ph.D. thesis, University of Pennsylvania, 1993.

[Daw95] A. Dawar, *Generalized quantifiers and logical reducibilities*, Journal of Logic and Computation **5** (1995), no. 2, 213–226.

[dBGAG92] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens, *On the completeness of object-creating query languages*, IEEE Conf. on Foundations of Computer Science, 1992, pp. 372–379.

[DH95] A. Dawar and L. Hella, *The expressive power of finitely many generalized quantifiers*, Information and Computation **123** (1995), no. 2, 172–184.

[DLW95] A. Dawar, S. Lindell, and S. Weinstein, *Infinitary logic and inductive definability over finite structures*, Information and Computation **119** (1995), no. 2, 160–175.

[Don88] G. Dong, *On the composition and decomposition of Datalog program mappings*, Proc. of Intl. Conf. on Database Theory, 1988, pp. 87–101.

[DS92] G. Dong and J. Su, *Incremental and decremental evaluation of transitive closure by first-order queries*, Tech. Report TRCS 92-18, University of California, Santa Barbara, 1992, To appear in Information and Computation.

[DS93] G. Dong and J. Su, *First-order incremental evaluation of datalog queries*, Proc. of Intl. Workshop on Database Programming Languages, 1993.

[DST94] G. Dong, J. Su, and R. Topor, *First-order incremental evaluation of datalog queries*, 1994, to appear.

[DT92] G. Dong and R. Topor, *Incremental evaluation of datalog queries*, Proc. of Intl. Conf. on Database Theory, 1992, pp. 282–296.

[DV93] K. Denninghoff and V. Vianu, *Database method schemas and object creation*, Proc. ACM Symp. on Principles of Database Systems, 1993, pp. 265–275.

[Eme91] E.A. Emerson, *Temporal and modal logic*, Handbook of Theoretical Computer Science (J. Van Leeuwen, ed.), Elsevier, 1991, pp. 997–1072.

[Fag75] R. Fagin, *Monadic generalized spectra*, Z. Math. Logik **21** (1975), 89–96.

[Fag77] R. Fagin, *Multivalued dependencies and a new normal form for relational databases*, ACM Trans. on Database Systems **2** (1977), 262–278.

[Fag82] R. Fagin, *Horn clauses and database dependencies*, J. ACM **29(4)** (1982), 952–985.

[Fag96] R. Fagin, *Combining fuzzy information from multiple systems*, Proc. ACM Symp. on Principles of Database Systems, 1996, To appear.

[Fre85] G.M. Frederickson, *Data structures for on-line updating of minimum spanning trees*, SIAM J. on Computing **14** (1985), 781–798.

[Fri71] H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory*, Logic Colloquium '69 (R.O. Gangy and C.M.E. Yates, eds.), North Holland, 1971, pp. 361–389.

[FV86] R. Fagin and M. Y. Vardi, *The theory of data dependencies: A survey*, Mathematics of Information Processing: Proceedings of Symposia in Applied Mathematics (Providence, RI) (M. Anshel and W. Gewirtz, eds.), vol. 34, American Mathematical Society, Providence, RI, 1986, pp. 19–71.

[Gal87] A. Galton, *Temporal logics and their applications*, Academic Press, London, San Diego, Sydney, 1987.

[Gel86] A. Van Gelder, *Negation as failure using tight derivations for general logic programs*, IEEE Symp. on Logic Programming, 1986, pp. 127–139.

[Gel89] A. Van Gelder, *The alternating fixpoint of logic programs with negation*, Proc. ACM Symp. on Principles of Database Systems, 1989, pp. 1–11.

[GO93] E. Grädel and M. Otto, *Inductive definability with counting on finite structures*, 6th Workshop on Computer Science Logic CSL 92, Springer Lecture Notes in Computer Science Nr. 702, 1993, pp. 231–247.

[Gra84] G. Grahne, *Dependency satisfaction in databases with incomplete information*, Proc. of Intl. Conf. on Very Large Data Bases, 1984, pp. 37–45.

[GRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf, *The well-founded semantics for general logic programs*, Proc. ACM Symp. on Principles of Database Systems, 1988, pp. 221–230.

[GRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf, *The well-founded semantics for general logic programs*, J. ACM **38** (1991), 620–650.

[GS86] Y. Gurevich and S. Shelah, *Fixed-point extensions of first-order logic*, Annals of Pure and Applied Logic **32** (1986), 265–280.

[GT91] A. van Gelder and R. Topor, *Safety and tranlsation of relational calculus queries*, ACM Trans. on Database Systems **16** (1991), 235–278.

[Gur66] Y. Gurevich, *The word problem for certain classes of semigroups (in Russian)*, Algebra and Logic **5** (1966), 25–35.

[Gur84] Y. Gurevich, *Toward a logic tailored for computational complexity*, Computation and Proof Theory (M.M. Richter etal, ed.), Springer-Verlag LNM 1104, 1984, pp. 175–216.

[Gur88] Y. Gurevich, *Logic and the challenge of computer science*, Trends in Theoretical Computer Science (E. Borger, ed.), Computer Science Press, 1988, pp. 1–57.

[GV91] S. Grumbach and V. Vianu, *Tractable query languages for complex object databases*, Proc. ACM Symp. on Principles of Database Systems, 1991.

[Hel92] L. Hella, *Logical hierarchies in ptime*, Proc. IEEE Conf. on Logic in Computer Science, 1992.

[Her95] C. Herrmann, *On the undecidability of implications between embedded multivalued database dependencies*, Information and Computation **122** (1995), 221–235.

[HKR93] G. Hillebrand, P. Kanellakis, and S. Ramaswamy, *Functional programming formalisms for OODB methods*, Proc. NATO ASI Summer School on OODBs, Kasadaci, Turkey, 1993.

[HMT71] L. Henkin, J.D. Monk, and A. Tarski, *Cylindric algebras*, North Holland, 1971.

[HY84] R. Hull and C. K. Yap, *The Format model: A theory of database organization*, Journal of the ACM **31** (1984), no. 3, 518–537.

[IL84] T. Imielinski and W. Lipski, *The relational model of data and cylindric algebras*, Journal of Computer and System Sciences **28(1)** (1984), 80–102.

[Imm86] N. Immerman, *Relational queries computable in polynomial time*, Inf. and Control **68** (1986), 86–104.

[JK84] D. S. Johnson and A. Klug, *Testing containment of conjunctive queries under functional and inclusion dependencies*, Journal of Computer and System Sciences **28** (1984), 167–189.

[JS82] G. Jaeschke and H.-J Schek, *Remarks on the algebra on non first normal form relations*, Proc. ACM Symp. on Principles of Database Systems, 1982, pp. 124–138.

[Jun79] C.G. Jung, *Collected works, vol. 5*, Princeton University Press, Princeton, NJ, 1979.

[Kan91] P. C. Kanellakis, *Elements of relational database theory*, Handbook of Theoretical Computer Science (J. Van Leeuwen, ed.), Elsevier, 1991, pp. 1074–1156.

[Kan95] P.C. Kanellakis, *Constraint programming and database languages: A tutorial*, Proc. ACM Symp. on Principles of Database Systems, 1995, pp. 46–53.

[KCV90] P. C. Kanellakis, S.S. Cosmadakis, and M. Y. Vardi, *Polynomial-time implications problems for unary inclusion dependencies*, J. ACM **37** (1990), 15–46.

[Kif88] M. Kifer, *On safety, domain independence, and capturability of database queries*, Proc. 3rd Intl. Conf. on Data and Knowledge Bases (C. Beeri, J.W. Schmidt, and U. Dayal, eds.), Morgan-Kaufmann, 1988, pp. 405–415.

[KKR90] P. Kanellakis, G. Kuper, and P. Revesz, *Constraint query languages*, Proc. 9th ACM Symp. on Principles of Database Systems (Nashville), 1990, pp. 299–313.

[Kol91] P. G. Kolaitis, *The expressive power of stratified logic programs*, Information and Computation **90** (1991), no. 1, 50–66.

[KP88] P. G. Kolaitis and C.H. Papadimitriou, *Why not negation by fixpoint?*, Proc. ACM Symp. on Principles of Database Systems, 1988, pp. 231–239.

[KPdB95] B. Kuijpers, J. Paredaens, and J. Van den Bussche, *Lossless representation of topological spatial data*, Fourth Int'l. Symp. on Large Spatial Databases, 1995.

[KS91] H.F. Korth and A. Silberschatz, *Database system concepts (2nd edition)*, McGraw-Hill, 1991.

[KU84] A. Keller and J.D. Ullman, *On complementary and independent mappings*, Proc. ACM SIGMOD Symp. on the Management of Data, 1984, pp. 143–148.

[KV84] G. Kuper and M. Y. Vardi, *A new approach to database logic*, Proc. ACM Symp. on Principles of Database Systems, 1984, pp. 86–96.

[KV93] G. M. Kuper and M. Y. Vardi, *The logical data model*, ACM Trans. on Database Systems **18** (1993), 379–413.

[Lei89a] D. Leivant, *Descriptive characterization of computational complexity*, Journal of Computer and System Sciences **39** (1989), 51–83.

[Lei89b] D. Leivant, *Monotonic use of space and computational complexity over abstract structures*, Tech. Report CMU-CS-89-212, Carneggie-Mellon University, 1989.

[Lif88] V. Lifschitz, *On the declarative semantics of logic programs with negation*, Foundations of Deductive Databases and Logic Programming (Los Altos, CA) (J. Minker, ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 177–192.

[Low15] L. Lowenheim, *Uber Möglichkeiten im relativekalkul*, Math. Ann. **76** (1915), 447–470.

[LS87] U. W. Lipeck and G. Saake, *Monitoring dynamic integrity constraints based on temporal logic*, Information Systems **12** (1987), no. 3, 255–269.

[LSW] J-B. Lagorce, A. Stockus, and E. Waller, *Object-oriented database evolution*, To appear.

[LT95] Y. Liu and T. Teitelbaum, *Systematic derivation of incremental programs*, Science of Computer Programming **24** (1995), 1–39.

[Mai83] D. Maier, *The theory of relational databases*, Computer Science Press, 1983.

[Mak77] A. Makinouchi, *A consideration of normal form of not-necessarily-normalized relations in the relational data model*, Proc. of Intl. Conf. on Very Large Data Bases, 1977, pp. 447–453.

[Mak81] J. A. Makowsky, *Characterizing data base dependencies*, 8th Colloquium on Automata, Languages and Programming, Springer-Verlag, 1981.

[Mit83] J. C. Mitchell, *The implication problem for functional and inclusion dependencies*, Information and Control **56** (1983), 154–173.

[MMS79] D. Maier, A.O. Mendelzon, and Y. Sagiv, *Testing implications of data dependencies*, ACM Trans. on Database Systems **4(4)** (1979), 455–469.

[MS] S. Marcus and V.S. Subrahmanian, *Foundations of multi-media information systems*, J. ACM, To appear.

[MSVT94] P.B. Miltersen, S. Subramanian, J.S. Vitter, and R. Tamassia, *Complexity models for incremental computation*, Theoretical Computer Science **130** (1994), no. 1, 203–236.

[MW88] D. Maier and D. S. Warren, *Computing with logic: Logic programming with prolog*, Benjamin Cummings, Menlo Park, CA, 1988.

[Pap86] C.H. Papadimitriou, *The theory of concurrency control*, Computer Science Press, 1986.

[Pap95] C.H. Papadimitriou, *Database metatheory: Asking the big queries*, Proc. ACM Symp. on Principles of Database Systems, 1995, pp. 1–10.

[PG88] J. Paredaens and D. Van Gucht, *Possibilities and limitations of using flat operators in nested algebra expressions*, Proc. ACM Symp. on Principles of Database Systems, 1988, pp. 29–38.

[PI94] S. Patnaik and N. Immerman, *Dyn-FO: A parallel, cynamic complexity class*, Proc. ACM Symp. on Principles of Database Systems, 1994.

[Pos47] E. L. Post, *Recursive unsolvability of a problem of Thue*, J. of Symbolic Logic **12** (1947), 1–11.

[Prz89] T. Przymusinski, *Every logic program has a natural stratification and an iterated least fixpoint model*, Proc. ACM Symp. on Principles of Database Systems, 1989, pp. 11–21.

[Prz90] T. Przymusinski, *Well-founded semantics coincides with three-valued stable semantics*, Fundamenta Informaticae **XIII** (1990), 445–463.

[PSV96] C.H. Papadimitriou, D. Suciu, and V. Vianu, *Topological queries in spatial databases*, Proc. ACM Symp. on Principles of Database Systems, 1996, To appear.

[Rau94] M. Rauch, *Improved data structures for fully dynamic biconnectivity*, Proc. ACM SIGACT Symp. on the Theory of Computing, 1994, pp. 686–695.

[RKS86] M.A. Roth, H.F. Korth, and A. Silberschatz, *Theory of non-first-normal-form relational databases*, Tech. Report TR-84-36, Univ. of Texas Austin, 1986.

[SP94] D. Suciu and J. Paredaens, *Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure*, Proc. ACM Symp. on Principles of Database Systems, 1994, pp. 171–179.

[ST95] A.P. Stolboushkin and M.A. Taitslin, *Finite queries do not have effective syntax*, Proc. ACM Symp. on Principles of Database Systems, May 1995, pp. 277–285.

[SY80] Y. Sagiv and M. Yannakakis, *Equivalence among expressions with the union and difference operators*, J. ACM **27(4)** (1980), 633–655.

[TF86] S.J. Thomas and P. C. Fischer, *Nested relational structures*, Advances in Computing Research, vol.3 (P. C. Kanellakis and F. Preparata, eds.), JAI Press, 1986, pp. 269–307.

[Tha91] B. Thalheim, *Dependencies in relational databases*, Teubner Verlagsgesellschaft, Stuttgart - Leipzig, 1991.

[Ull82] J.D. Ullman, *Principles of database systems (2nd edition)*, Computer Science Press, Potomac, Maryland, 1982.

[Ull88] J.D. Ullman, *Principles of database and knowledge base systems, volume i*, Computer Science Press, 1988.

[Ull89] J.D. Ullman, *Principles of database and knowledge base systems, volume ii: The new technologies*, Computer Science Press, 1989.

[UY90] J.D. Ullman and M. Yannakakis, *The input/output complexity of transitive closure*, Proc. ACM SIGMOD Symp. on the Management of Data, 1990, pp. 44–53.

[Var82] M. Y. Vardi, *The complexity of relational query languages*, Proc. ACM SIGACT Symp. on the Theory of Computing, 1982, pp. 137–146.

[Var87] M. Y. Vardi, *Fundamentals of dependency theory*, Trends in Theoretical Computer Science (E. Borger, ed.), Computer Science Press, 1987, pp. 171–224.

[Yan86] M. Yannakakis, *Querying weak instances*, Advances in Computing Research, vol. 3 (P. C. Kanellakis and F. Preparata, eds.), JAI Press, 1986, pp. 185–212.

[Yan95] M. Yannakakis, *Perspectives on database theory*, IEEE Conf. on Foundations of Computer Science, 1995, pp. 224–246.

Department of Computer Science and Engineering, CSE 0114, Univ. of California at San Diego, La Jolla, California 92093-0114, USA

E-mail address: vianu@cs.ucsd.edu