DEDUCTIVE DATABASES

The field of deductive databases is based on logic. The computational paradigm of deductive databases is to use rules that are provided with the database to derive new data from facts in the database. We describe a deductive database as well as a query and an answer to a query in a deductive database. Also discussed is how deductive databases extend relational databases (see the article, RELATIONAL DATABASES) and why it is a subset of logic programming (see AI LANGUAGES AND PROCESSING). Deductive databases are useful for expert and knowledge base systems needed in engineering applications. Then, the pre-history and the start of the field are described. The major historical developments in the field are discussed in subsequent sections. *Datalog* databases, negation, recursion in *Datalog,* semantic query optimization *SQO,* and user constraints *UC*s are described. *Datalog*$^{\neg}$ is discussed and alternative theories of negation and how they relate to knowledge base systems are explained (see KNOWLEDGE MANAGEMENT for details). Incomplete databases, denoted *Datalog*$^{\neg}_{disj}$, are described; such databases permit more expressive knowledge base systems.

BACKGROUND

A deductive database is an extension of a relational database. Formally, a deductive database (*DDB*) is a triple, ⟨*EDB, IDB, IC*⟩, where *EDB* is a set of facts, called the extensional database, *IDB* is a set of rules, called the intensional database, and *IC* is a set of integrity constraints. A *DDB* is based on first-order logic. An atomic formula is a k-place predicate symbol whose arguments are constants or variables. Atomic formulas evaluate to *true* or *false.* The *EDB* consists of ground atomic formulas or disjunctions of ground atomic formulas. An atomic formula is ground if it consists of a predicate with k arguments, where the arguments are constants. Examples of ground atomic formulas are *supplies*(*acme, shovels*), and *supplies*(*acme, screws*) whose intended meaning is "The Acme Corporation supplies shovels and screws." An example of a disjunction is: *supplierloc*(*acme, boston*) ∨ *supplierloc*(*acme, washington*) whose intended meaning is "The Acme Corporation is located either in Boston or in Washington, or in both locations." Corresponding to an atomic formula, there is a relation that consists of all tuples whose arguments are in an atomic formula with the same name. For the supplies predicate, there is a relation, the *SUPPLIES* relation that consists of a set of tuples, for example {⟨*acme, shovels*⟩, ⟨*acme, screws*⟩}, when the *SUPPLIES* relation consists of the above two facts. When facts in the *EDB* consist only of atoms, it is equivalent to a relational database. Throughout the article, predicate letters are written in lower case, and arguments of predicates that are constants are also written in lower case, while upper case letters denote variables.

The intensional database consists of a set of rules of the form:

$$L_1, \ldots, L_n \leftarrow M_1, \ldots, M_m, not\, M_{m+1}, \ldots, not\, M_{m+l}, \quad (1)$$

where the L_i and the M_j are atomic formulas, and *not* is a rule of default for negation (discussed below). Intensional rules are universally quantified and are an abbreviation of the formula:

$$\forall X_1 \ldots, X_n (A_1 \vee \ldots \vee A_n) \leftarrow B_1 \wedge \ldots \wedge B_m, not\, B_{m+1}, \ldots, not\, B_{m+l}), \quad (2)$$

where the $X_1, \ldots, X_n$ lists all free variables in Eq. (2).

*IC*s are rules written as in Eq. (2) and are used to describe properties that entries in a database should satisfy, so as to maintain database consistency during updates and for additional purposes as discussed below.

*DDB*s restrict arguments of atomic formulas to constants and variables, whereas in first-order logic, atomic formulas may also contain function symbols as arguments. This assures that answers to queries in *DDB*s return a finite set of answers. When there are function symbols, an infinite set of answers may be returned since an infinite number of terms may be generated from the finite number of constants and the function symbols, which is not possible in *DDB*s that contain a finite number of constants. Rules may be read either declaratively or procedurally. A declarative reading of Eq. 2 is: L_1 *or* L_2 *or . . . or* L_n is *true* if M_1 *and* M_2 *and . . . and* M_m *and not* M_{m+1} *and . . . and not* M_{m+l} are all *true.*

A procedural reading of Eq. 2 is: L_1 *or* L_2 *or . . . or* L_n are solved if M_1 *and* M_2 *and . . . and* M_m *and not* M_{m+1} *and . . . and not* M_{m+l} can be solved.

The left hand side of the implication, L_1 *or . . . or* L_n is called the head of the rule, while the right hand side, M_1 *and* M_2 *and . . . and* M_m *and not* M_{m+1} *and . . . and not* M_{m+l} is called the body of the rule.

Queries to a database, $Q(X_1, \ldots, X_r)$ are of the form $\exists X_1 \ldots \exists X_r(L_1 \wedge L_2 \ldots \wedge L_s)$ where $s \geq 1$, the L_i are literals, and the X_i, $1 \leq i \leq r$ are the free variables in Q. An answer to a query has the form $\langle a_{11}, \ldots, a_{1r}\rangle + \langle a_{21}, \ldots, a_{2r}\rangle + \cdots + \langle a_{k1}, \ldots, a_{kr}\rangle$ such that $Q(a_{11}, \ldots, a_{1r}) \vee Q(a_{21}, \ldots, a_{2r}) \vee \cdots \vee Q(a_{k1}, \ldots, a_{kr})$ is provable from the database. By provable, it is meant that an inference system is used to find answers to queries.

*DDB*s are closely related to logic programs when the facts are restricted to atomic formulas and the rules have only one atom in the left hand side of a rule. The main difference is that a logic program query searches for a single answer over a small set of facts, whereas a DDB query searches over a large set of facts to find all answers. In a logic program search proceeds top-down from the query to an answer. In *DDB*s, searches are bottom-up, starting from the facts, to find all answers. A logic program query might ask for an item supplied by a supplier, while in a deductive database, a query asks for all items supplied by a supplier. This seemingly slight difference actually has a dramatic impact on techniques required for Deductive Database query processing. Neither standard logic program query search proceeding purely top-down from the query to an answer, nor standard bottom-up search starting from the facts are adequate. An appropriate mix of both is actually required. *DDB*s restricted to atoms as facts and rules that consist of single atoms on the left hand side of a rule and atoms on the right hand side of a rule that do not contain the default rule for negation, *not,* are called *Datalog* databases, that is, rules in Eq. 2, where $n = 1$, $m \geq 0$, and $l = 0$. Rules in *Datalog* databases may be recursive. A rule is recursive if a literal with the same predicate symbol appears both in the left-hand and the right-hand side of Eq. (2). A relational database is a *DDB,* where the *EDB* consists of atoms, *IDB* rules are generally not recursive, and contains *IC*s. When all rules are nonrecursive in a relational database, they are called views.

There are several different concepts of the relationship of integrity constraints to the union of the *EDB* and the *IDB* in the *DDB.* Two such concepts are consistency and theoremhood. In the consistency approach, (proposed by Kowalski), an *IC* must be consistent with *EDB* $\cup$ *IDB.* In the theoremhood approach (proposed by Reiter and by Lloyd and Topor), an *IC* must be a theorem of *EDB* $\cup$ *IDB.*

To answer queries that consist of conjunctions of positive and default negated atoms in *Datalog* requires that a semantics be associated with negation since only positive atoms can be derived from *Datalog DDB*s. Default rules are used to find answers to negated questions. Several default rules are used in *Datalog DDB*s. Two are termed the closed world assumption (*CWA*), due to Reiter, and negation-as-finite-failure (*NFF*), due to Clark. In the *CWA,* failure to prove the positive atom implies that the negated atom is true. In the *NFF,* predicates in the *EDB* and the *IDB* are considered the *if* portion of the database and are closed by effectively reversing the implication to achieve the *only if* part of the database. The two approaches lead to slightly different results. Negation, as applied to disjunctive theories, is discussed later.

Example 1 Ancestor. Consider the following database that consists of parents and ancestors. The database consists of two predicates, whose schema are $p(X, Y)$ and is intended to mean that Y is the parent of X and $a(X, Y)$, which is intended to mean that Y is an ancester of X. The database consists of four EDB statements and two IDB rules:

r1. $p(mike, jack)$
r2. $p(sally, jack)$
r3. $p(katie, mike)$
r4. $p(beverly, mike)$
r5. $a(X, Y) \leftarrow p(X, Y)$
r6. $a(X, Y) \leftarrow a(X, Z), a(Z, Y)$

The answer to the question $p(mike, X)$ is *jack.* The answer to the question $a(mike, X)$ is *jack,* using rule r5. An answer to the query $a(katie, X)$ is *mike* using rule r5. Another answer to the query $a(katie, X)$ is found by using rule r6, and the fact that we have found that $a(katie, mike)$ and $a(mike, jack)$.

If we were to ask the query, $p(katie, jack)$, there is no response since there are only four facts, none of which specify $p(katie, jack)$, and there are no rules that can be used to find additional parents. Hence, the answer to the query by the *CWA* is *no, jack* is not the *parent of katie.*

More expressive power may be obtained in a *DDB* by allowing negated atoms on the right hand side of a rule. The semantics associated with such databases depends upon how the rule of negation is interpreted, as discussed later.

HISTORICAL BACKGROUND OF DEDUCTIVE DATABASES

The prehistory of *DDB*s is considered to be from 1957–1970. The efforts in this period used primarily ad-hoc or simple approaches to perform deduction. The period 1970–1980 were the formative years, which preceded the start of the field.

Prehistory of Deductive Databases

In 1957 a system, *ACSI-MATIC* was being developed to automate work in Army intelligence. An objective was to derive a new data based upon given information and general rules. Chains of related data were sought, and data contained reliability estimates. A prototype system was implemented to derive new data whose reliability values depended upon the reliability of the original data. The deduction used was modus ponens (i.e., from p and $p \rightarrow q$, one concludes q, where p and q are propositions).

Several *DDB*s were developed in the 1960s. Although in 1970, Codd founded the field of Relational Databases, relational systems were in use before then. In 1963, using a relational approach, Levien and Maron developed a system, Relational Data File (*RDF*), that had an inferential capability, implemented through a language termed *INFEREX.* An *INFEREX* program could be stored in the system (such as in current systems that store views) and re-executed, if necessary. A programmer specified reasoning rules via an *INFEREX* program. The system handled credibility ratings of sentences in forming deductions. Theoretical work by Kuhns on the *RDF* project recognized that there were classes of questions that were, in a sense, not reasonable. For example, let the database consist of the statement, "Reichenbach wrote *Elements of Symbolic Logic.*" Whereas the question, "What books has Reichenbach written?" is reasonable, the questions, "What books has Reichenbach not written?" or "Who did not write *Elements of Symbolic Logic*?" are not reasonable. It is one of the first times that the issue of negation in queries was explored. Kuhns related the imprecise notion of a reasonable question with a precisely defined notion of a definite formula.

The notion of definiteness is approximately as follows: Given a set of sentences S, a dictionary containing known terms D_S, a particular query Q, and an arbitrary name n, Q is said to be semidefinite *iff* for any name n, the answer to query Q is independent of whether or not D_S contains n. Q is said to be definite *iff* Q is semidefinite on every sentence set S. DiPaola proved there is no algorithm to determine whether or not a query is definite. This may be the first application of a theory of computing to databases.

Kuhns also considered the general problem of quantification in query systems. Related to work by Kuhns were papers in the late 1950s and early 1960s devoted to a general theory or formalization of questions by Aqvist in 1965, Belnap in 1963, Carnap in 1956, Harrah in 1963, Jespersen in 1965, and Kasher in 1967.

In 1966, Marill developed a system, Relational Structure System (*RSS*) that consisted of 12 rules that permitted such capabilities as chaining. He used a deduction procedure termed a breadth-first-followed-by-depth manner. Other work during that time was performed by Love, Rutman, and Savitt in 1970, on a system termed an Associative Store Processor (*ASP*).

In 1964, Raphael, for his Ph.D. thesis at M.I.T., developed a system, Semantic Information Retrieval (*SIR*), which had a limited capability with respect to deduction, using special rules. Green and Raphael subsequently designed and implemented several successors to *SIR:* $QA - 1$, a re-implementation of *SIR;* $QA - 2$, the first system to incorporate the Robinson Resolution Principle developed for automated theorem proving; $QA - 3$ that incorporated added heuristics; and $QA - 3.5$, which permitted alternative design strategies to be tested within the context of the resolution theorem prover. Green and Raphael were the first to recognize the importance and applicability of the work performed by Robinson in automated theorem proving. They developed the first *DDB* using formal techniques based on the Resolution Principle, which is a generalization of *modus ponens* to first-order predicate logic. The Robinson Resolution Principle is the standard method used to deduce new data in *DDB*s.

Deductive Databases: The Formative Years 1969–1978

The start of deductive databases is considered to be November, 1977, when a workshop, "Logic and Data Bases," was organized in Toulouse, France. The workshop included researchers who had performed work in deduction from 1969 to 1977 and used the Robinson Resolution Principle to perform deduction. The workshop, organized by Gallaire and Nicolas, in collaboration with Minker, led to the publication of papers from the workshop in the book, *Logic and Data Bases,* edited by Gallaire and Minker.

Many significant contributions were described in the book. Nicolas and Gallaire discussed the difference between model theory and proof theory. They demonstrated that the approach taken by the database community was model theoretic; that is, the database represents the truth of the theory; queries are answered by a bottom-up search. However, in logic programming, answers to a query used a proof theoretic approach, starting from the query, in a top-down search. Reiter contributed two papers. One dealt with compiling axioms. He noted that if the IDB contained no recursive axioms, then a theorem prover could be used to generate a new set of axioms where the head of each axiom was defined in terms of relations in a database. Hence, a theorem prover was no longer needed during query operations. His second paper discussed the closed world assumption (*CWA*), whereby in a theory, if one cannot prove an atomic formula is true, then the negation of the atomic formula is assumed to be true. Reiter's paper elucidated three major issues: the definition of a query, an answer to a query, and how one deals with negation. Clark presented an alternative theory of negation. He introduced the concept of *if-and-only-if* conditions that underly the meaning of negation, called negation-as-finite-failure. The Reiter and Clark papers are the first to formally define default negation in logic programs and deductive databases. Several implementations of deductive databases were reported. Chang developed a system termed *DEDUCE;* Kellog, Klahr, and Travis developed a system termed Deductively Augmented Data Management System (*DADM*); and Minker described a system termed Maryland Refutation Proof Procedure 3.0 (*MRPPS 3.0*). Kowalski discussed the use of logic for data description. Darvas, Futo, and Szeredi presented applications of *Prolog* to drug data and drug interactions. Nicolas and Yazdanian described the importance of integrity constraints in deductive databases. The book provided, for the first time, a comprehensive description of the interaction between logic and data bases.

References to work on the history of the development of the field of deductive databases may be found in Refs. 1 and 2. A brief description of the early systems is contained in Ref. 2. See Ref. 3 for papers cited in the book *Logic and Data Bases.*

DATALOG AND EXTENDED DATALOG DEDUCTIVE DATABASES

The first generalization of relational databases was to permit function-free recursive Horn rules in a database, that is, rules in which the head of a rule is an atom and the body of a rule is a conjunction of atoms (i.e., in Eq. 2, $n = 1$, $m \geq 0$ and $l = 0$). These databases are called deductive databases *DDB*s, or *Datalog* databases.

Datalog Databases

In 1976, van Emden and Kowalski formalized the semantics of logic programs that consists of Horn rules, where the rules are not necessarily function-free. They recognized that the semantics of Horn theories can be characterized in three distinct ways: by model, fixpoint, or proof theory. These three characterizations lead to the same semantics. When the logic program is function-free, their work provides the semantics for *Datalog* databases.

Model theory deals with a collection of models that captures the intended meaning of the database. Fixpoint theory deals with a fixpoint operator that constructs the collection of all atoms that can be inferred to be true from the database. Proof theory provides a procedure that finds answers to queries with respect to the database. van Emden and Kowalski showed that the intersection of all Herbrand models of a Horn *DDB* is a unique minimal model, is the same as all of the atoms in the fixpoint, and are the only atoms provable from the theory.

Example 2 Example of Semantics. Consider Example 1. The unique minimal model of the database is:

$$\begin{aligned}M = \{&p(mike, jack), p(sally, jack) p(katie, mike),\\ &p(beverly, mike), a(mike, jack), a(sally, jack),\\ &a(katie, mike), a(beverly, mike), a(katie, jack),\\ &a(beverly, jack)\}\end{aligned}$$

These atoms are all true, and when substituted into the rules in Example 1, they make all of the rules true. Hence, they form a model. If we were to add another fact to the model M, say $p(jack, sally)$, it would not contradict any of the rules, and would also be a model. This fact can be eliminated since the original set was a model and is contained in the expanded model. That is, minimal Herbrand models are preferred. It is also easy to see that the atoms in M are the only atoms that can be derived from the rules and the data. In Example 3, below, we show that these atoms are in the fixpoint of the database.

To find if the negation of a ground atom is true, one can substract, from the Herbrand base (the set of all atoms that can be constructed from the constants and the predicates in the database), the minimal Herbrand model. If the atom is contained in this set, then it is assumed false, and its negation is true. Alternatively, answering queries that consist of negated atoms that are ground may be achieved using negation-as-finite failure as described by Clark.

Initial approaches to answer queries in *DDB*s did not handle recursion and were primarily top-down (or backward reasoning). Answering queries in relational database systems was a bottom-up (or forward reasoning) approach to find all answers. In order to handle recursion, a variety of techniques were developed covering a range of different approaches. These techniques are usually separated into classes depending on whether they focus on top-down or bottom-up evaluation. Some are centered around an approach known as *Query SubQuery* (*QSQ*) introduced initially and developed further by Vielle (4,5); these are top-down. In this same class, the Extension Table method was defined by Dietrich and Warren (6) at about the same time. Others, centered around an approach called magic set rewriting, are based on an initial preprocessing of the datalog program before using a fairly direct bottom-up evaluation strategy. Magic sets were introduced initially by Bancilhon et al. (7) and developed further by Beeri and Ramakrishnan (8). In this same class, the Alexander method was defined by Rohmer, Lescoeur, and Kerisit at about the same time. The advantage of top-down techniques is that they naturally take advantage of constants in a query and thereby restrict the search space while enabling the use of optimized versions of relational algebra set-oriented operations where appropriate. Although there is no direct way to take advantage of the same information in bottom-up evaluation, there are bottom-up techniques that have essentially the same running time as top-down techniques. Indeed, Bry (9) has shown that the Alexander and magic set methods based on rewriting and methods based on resolution implement the same top-down evaluation of the original database rules by means of auxiliary rules processed bottom-up. In principle, handling recursion poses no additional problems. One can iterate search (referred to as the naive method) until a fixpoint is reached. This can be achieved in a finite set of steps since the database has a finite set of constants and is function free. However, it is unknown how many steps will be required to obtain the fixpoint. The Alexander and magic-set methods improve search time, when recursion exists, such as for transitive closure rules.

Example 3 Fixpoint. The fixpoint of a database is the set of all atoms that satisfy the EDB and the IDB. The fixpoint may be found in a naive manner by iterating until there are no more atoms that can be found. This is done as follows. We can consider that

Step (0) = ϕ. That is, nothing is in the fixpoint.

Step (1) = {*p*(*mike, jack*), *p*(*sally, jack*), *p*(*katie, mike*), *p*(*beverly, mike*)}. These are all facts and satisfy r1, r2, r3, and r4. The atoms in Step (0) ∪ Step (1) now constitute the partial fixpoint.

Step (2) = {*a*(*mike, jack*), *a*(*sally, jack*), *a*(*katie, mike*), *a*(*beverly, mike*)} are found by using the results of Step (0) ∪ Step (1) on rules r5 and r6. Only rule r5 provides additional atoms when applied. Step (0) ∪ Step (1) ∪ Step (2) become the revised partial fixpoint.

Step (3) = {*a*(*katie, jack*), *a*(*beverly, jack*)}. This results from the previous partial fixpoint. These were obtained from rule r6, which was the only rule that provided new atoms at this step. The new partial fixpoint is Step (0) ∪ Step (1) ∪ Step (2) ∪ Step (3).

Step (4) = ϕ. Hence, no additional atoms can be found that satisfy the EDB ∪ IDB. Hence, the fixpoint iteration may be terminated, and the fixpoint is Step (0) ∪ Step (1) ∪ Step (2) ∪ Step (3).

Notice that this is the same as the minimal model M in Example 1.

The naive fixpoint method, however, has several efficiency problems. One is that queries containing constants, such as ?$a(X, jack)$ can be computed more efficiently using a top-down approach. As previously discussed, the Alexander and magic set methods address this problem. A second source of inefficiency is that, at each iteration, the naive fixpoint recomputes old atoms along with new ones. This problem is solved by symbolic differentiation of the rules, yielding what is often called the seminaive fixpoint method. Therefore, most compilers for deductive database languages use a combination of methods, and rely on static analysis to choose the best method for the problem at hand. In general it is not known how many steps will be required to achieve the fixpoint.

Classes of recursive rules exist where it is known how many iterations will be required. These rules lead to what has been called bounded recursion, noted first by Minker and Nicolas and extended by Naughton and Sagiv. Example 4 illustrates bounded recursion.

Example 4 Bounded Recursion. If a rule is singular, then it is bound to terminate in a finite number of steps independent of the state of the database. A recursive rule is singular if it is of the form

$$R \leftarrow F \wedge R_1 \wedge \ldots \wedge R_n$$

where F is a conjunction of possibly empty base relations (i.e., empty EDB) and $R, R_1, R_2, \ldots, R_n$ are atoms that have the same relation name iff:

1. Each variable that occurs in an atom R_i and does not occur in R only occurs in R_i.
2. Each variable in R occurs in the same argument position in any atom R_i where it appears, except perhaps in at most one atom R_1 that contains all of the variables of R.

Thus, the rule

$$R(X, Y, Z) \leftarrow R(X, Y', Z), R(X, Y, Z')$$

is singular since (a) Y' and Z' appear respectively in the first and second atoms in the body of the rule (condition 1), and (b) the variables X, Y, Z always appear in the same argument position (condition 2).

The major use of *IC*s has been to update a database to assure it is consistent. Nicolas has shown how, using techniques from *DDB*s, to improve the speed of update. Reiter has shown that *Datalog* database can be queried with or without *IC*s, and the answer to the query is identical. However, this does not preclude the use of *IC*s in the query process. While *IC*s do not affect the result of a query, they may affect the efficiency to compute an answer. *IC*s provide semantic information about the data in the database. If a query requests a join (see RELATIONAL DATABASES) for which there will never be an answer because of the constraints, this can be used not to perform the query and to return an empty answer set. This avoids unnecessary joins on potentially large relational databases, or performing a long deduction in a *DDB*. The use of *IC*s to constrain a search is called semantic query optimization (*SQO*). McSkimin and Minker were the first to use *IC*s for *SQO* in *DDB*s. Hammer, Zdonik, and King first applied *SQO* to relational databases. Chakravarthy, Grant, and Minker formalized *SQO* and developed the partial subsumption algorithm and method of residues. These provide a general technique applicable to any relational or *DDB*. Godfrey, Gryz, and Minker apply the technique bottom-up. Gaasterland and Lobo extend *SQO* to include databases with negation in the body of rules, and Levy and Sagiv handle recursive *IDB* rules in *SQO*.

A topic related to *SQO* is that of cooperative answering systems. The objective is to inform a user as to why a particular query succeeded or failed. When a query fails, one generally, cannot tell why failure occurred. There may be several reasons: the database currently does not contain information to respond to the user, or there will never be an answer to the query. The distinction could be important to the user. User constraints (*UC*) are related to *IC*s. A user constraint is a formula that models a user's preferences. It may constrain providing answers to queries in which the user may have no interest (e.g., stating that in developing a route of travel, the user does not want to pass through a particular city) or provide other constraints to restrict search. When *UC*s are identical in form to *IC*s, they can be used for this purpose. While *IC*s provide the semantics of the entire database, *UC*s provide semantics of the user. *UC*s may be inconsistent with a database. Thus, a separation of these two semantics is essential. To maintain consistency of the database, only *IC*s are relevant. A query may be thought of as the conjunction of the query and the *UC*s. Hence, a query can be semantically optimized based both on *IC*s and *UC*s.

Other features may be built into a system, such as the ability to relax a query given that it fails, so that an answer to a related request may be found. This has been termed query relaxation.

The first article on magic sets may be found in Ref. 7 and further extensions in Ref. 8. A description of the magic set method to handle recursion in *DDB*s may be found in Refs. 10 and 11. The presentation of the *Extension Table* method is given in Ref. 10. The *QSQ* method was introduced initially in Ref. 4 and developed further in Ref. 5. The textbook by Abiteboul, Hull, and Vianu (12) presents an in-depth description of (Extended) Datalog syntax and semantics. It also provides comprehensive and detailed comparative analyses of Recursive Query Processing techniques. References to work in bounded recursion may be found in Ref. 2. For work on fixpoint theory of *Datalog,* and the work of van Emden and Kowalski, see the book by Lloyd (13). A comprehensive survey and references to work in cooperation answering systems is in Ref. 14. References to alternative definitions of *ICs,* semantic query optimization and the method of partial subsumption may be found in Ref. 2.

Extended Deductive Databases $Datalog^{\neg}_{ext}$ and Knowledge Bases

The ability to develop a semantics for databases, in which rules have a literal (i.e., an atomic formula or the negation of an atomic formula) in the head and literals with possibly negated-by-default literals in the body of a rule, has significantly expanded the ability to write and understand the semantics of complex applications. Such rules, called extended clauses, contain rules in Formula 2 where $n = 1$, $m \geq 0$, $l \geq 0$, and the *As* and *Bs* are literals. Such databases combine classical negation (represented by $\neg$) and default negation (represented by *not* immediately preceding a literal) and are called extended deductive databases. Combining classical and default negation provides users greater expressive power.

Logic programs that used default negation in the body of a clause first started in 1986. Apt, Blair and Walker, and Van Gelder introduced the concept of stratification to logic programs in which L_1 and the M_j in Formula (2) are atomic formulas, and there is no recursion through negation. They show there is a unique preferred minimal model computed from strata to strata. Przymusinski termed this minimal model the perfect model. When a theory is stratified, rules can be placed in different strata, where the definition of a predicate in the head of a rule is in a higher stratum than the definitions of predicates negated in the body of the rule. The definition of a predicate is the collection of rules containing the predicate in their head. Thus, one can compute positive predicates in a lower stratum, and a negated predicate's complement is true in the body of the clause if the positive atom has not been computed in the lower stratum. The same semantics is obtained regardless of how the database is stratified. When the theory contains no function symbols, the *DDB* is termed $Datalog^{\neg}$. If a database can be stratified, then there is no recursion through negation, and the database is called $Datalog^{\neg}_{strat}$.

Example 5 Stratified Program. The rules,

$$
\begin{array}{l}
r_1 : p \leftarrow q,\ not\ r \\
r_2 : q \leftarrow p \\
r_3 : q \leftarrow s \\
r_4 : s \\
\hline
r_5 : r \leftarrow t
\end{array}
$$

comprise a stratified theory in which there are two strata. The rule r_5 is in the lowest stratum, while the other rules are in a higher stratum. The predicate p is in a higher stratum than the stratum for r since it depends negatively on r. q is in the same stratum as p, as it depends upon p. s is also in the same stratum as q. The meaning of the stratified program is that $\{s, q, p\}$ are true, while $\{t, r\}$ are false. t is false since there is no defining rule for t. Since t is false, r is false, s is given as true, and, hence, q is true. Since q is true, and r is false, from rule r_1, p is true.

The theory of stratified databases was followed by permitting recursion through negation in Eq. (2), where the L_1 and M_j are atomic formulae, $n = 1$, $m \geq 0$, $l \geq 0$. In the context of *DDB*s, they are called normal deductive databases. Many semantics have been developed for these databases. The most prominent are the well-founded semantics of Van Gelder, Ross, and Schlipf and the stable semantics of Gelfond and Lifschitz. When the well-founded semantics is used, the database is called $Datalog^{\neg}_{norm,wfs}$, and when the stable semantics is used, the database is called $Datalog^{\neg}_{norm,stable}$. The well-founded semantics leads to a unique three-valued model, while the stable semantics leads to a (possibly empty) collection of models.

Example 6. Non-Stratifiable Database. Consider the database given by:

$$
\begin{aligned}
r_1 &: p(X) \leftarrow not\, q(X) \\
r_2 &: q(X) \leftarrow not\, p(X) \\
r_3 &: r(a) \leftarrow p(a) \\
r_4 &: r(a) \leftarrow q(a)
\end{aligned}
$$

Notice that r_1 and r_2 are recursive through negation. Hence, the database is not stratifiable. According to the well-founded semantics, $\{p(a), q(a), r(a)\}$ are assigned unknown. However, for the stable model semantics, there are two minimal models: $\{\{p(a), r(a)\}, \{q(a), r(a)\}\}$. Hence, one can conclude that $r(a)$ is true, while the disjunct, $p(a) \vee q(a)$, is true in the stable model semantics.

Relationships have been noted between the well-founded semantics and the stable semantics. When a database has a total well-founded model, that is, there are no unknown atoms, then this is also a stable model for the database and there are no other stable models. The stable model semantics can also be extended with three-valued logic, and then stable models generalize well-founded models.

Chen and Warren implemented a top-down approach to answer queries in the well-founded semantics, while Leone and Rullo developed a bottom-up method for $Datalog^{\neg}_{norm,wfs}$ databases. Several methods have been developed for computing answers to queries in stable model semantics. Fernández, Lobo, Minker, and Subrahmanian developed a bottom-up approach to compute answers to queries in stable model semantics based on the concept of model trees. Every branch of a model tree is a model of the database, where a node in a tree is an atom shared by each branch below that node. See Fig. 1 for an illustration of a model tree. Bell, Nerode, Ng, and Subrahmanian developed a method based on linear programming. Efficient implementations of stable model semantics are described by Niemela and Simons (15) and by Marek et al. (16).

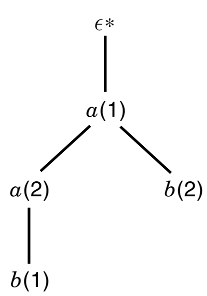

Figure 1. Model tree.

An extension of normal deductive databases, proposed by Gelfond and Lifschitz and by Pearce and Wagner, permits rules in Eq. (2), where L and M_j are literals. The semantics for normal deductive databases can be computed using a transformation that renames all classically negated atoms and adding an *IC* that states the new atom and the atom from which it arose cannot be true at the same time.

These notions of default negation have been used as separate ways to interpret and to deduce default information. That is, each application has chosen one notion of negation and has applied it to every piece of data in the domain of the application. Minker and Ruiz defined a more expressive *DDB* that allows several forms of default negation in the same database. Hence, different information in the domain may be treated appropriately. They introduce a new semantics called the well-founded stable semantics that characterizes the meaning of *DDB*s that combine well-founded and stable semantics.

A reason to extend databases to achieve more general non-monotonic semantics is the search for greater expressive power needed to implement knowledge base systems. Knowledge bases are important for artificial intelligence and expert system developments. A general way to represent knowledge bases is through logic. Work developed for extended *DDB*s concerning semantics and complexity apply directly to knowledge bases. For an example of a knowledge base, see Example 7. Extended *DDB*s, together with *IC*s, permit a wide range of knowledge bases (*KB*) to be implemented.

Since alternative extended *DDB*s have been implemented, the *KB* expert can focus on writing rules and *IC*s that characterize the problem, selecting the semantics that meets the needs of the problem and employing a *DDB* system that uses the required semantics.

Articles on stratified databases by Apt, Blair, and Walker, by Van Gelder, and by Przymusinski may be found in Ref. 17. See Refs. 10 and 11 for a description of computing answers to queries in stratified databases. For an article on the semantics of $Datalog^{\neg}_{wfs}$, see Ref. 18, see Ref. 19 for the stable model semantics, Ref. 2 for references to work on other semantics for normal extended deductive databases, and Schlipf (20) for a comprehensive survey article on complexity results for deductive databases. For results on negation in deductive databases, see the survey article by Shepherdson (21).

EXTENDED DISJUNCTIVE DEDUCTIVE DATABASE SEMANTICS $Datalog^{\neg}_{disj,ext}$

In the above databases, information is definite. However, many applications exist where knowledge of the world is incomplete. For example, when a null value appears as an argument of an attribute of a relation, the value of the attribute is unknown. Also, uncertainty in databases may be represented by probabilistic information. Another area of incompleteness arises when it is unknown which among several facts are true, but it is known one or more are true. It is, therefore, necessary to be able to represent and understand the semantics of theories that include incomplete data. The case where there is disjunctive information is discussed below. A natural extension is to permit disjunctions in the *EDB* and disjunctions in the heads of *IDB* rules. These rules are represented in Formula 2, where $n \geq 1$, $m \geq 0$, and $l \geq 0$, and are called extended disjunctive rules. Such databases are called extended disjunctive deductive databases (*EDDDB*), or $Datalog^{\neg}_{disj,ext}$.

Example 7 Knowledge Base (22) Consider the database, where $p(X, Y)$ denotes X is a professor in department Y, $a(X, Y)$ denotes individual X has an account on machine Y, $ab(W, Z)$ denotes it is abnormal in rule W to be individual Z.

We wish to represent the following information where *mike* and *john* are professors in the computer science department:

1. As a rule, professors in the computer science department have Vax accounts. This rule is not applicable to Mike. He may or may not have an account on that machine.
2. Every computer science professor has one of the Vax or IBM accounts, but not both.

These rules are reflected in the following extended disjunctive database.

1. $p(mike, cs) \leftarrow$
2. $p(john, cs) \leftarrow$
3. $\neg p(X, Y) \leftarrow not\ p(X, Y)$
4. $a(X, vax) \leftarrow p(X, cs),\ not\ ab(r4, X),\ not\ \neg a(X, vax)$
5. $ab(r4, mike) \leftarrow$
6. $a(X, vax) \vee a(X, ibm) \leftarrow p(X, cs),\ ab(r4, X)$
7. $\neg a(X, ibm) \leftarrow p(X, cs),\ a(X, vax)$
8. $\neg a(X, vax) \leftarrow p(X, cs),\ a(X, ibm)$
9. $a(X, ibm) \leftarrow \neg a(X, vax),\ p(X, cs)$

Rule 3 states that if by default, negation $p(X, Y)$ fails, then $p(X, Y)$ is logically false. The other rules encode the statements listed above. From this formalization one can deduce that *john* has a *vax* account, while *mike* has either a *vax* or an *ibm* account, but not both.

The semantics of *DDDB*s, is discussed first, where clauses are given by Formula (2), literals are restricted to atoms, and there is no default negation in the body of a clause. Next, the semantics of *EDDDB*s, where there are no restrictions on clauses in Eq. (2) is discussed.

Disjunctive Deductive Databases (DDDBs), $Datalog^{\neg}_{disj}$

The field of disjunctive deductive databases (*DDDB*s), referred to as $Datalog^{\neg}_{disj}$, started in 1982 by Minker who described how to answer both positive and negated queries in such databases. A major difference between the semantics of *DDB*s and *DDDB*s is that *DDB*s usually have a unique minimal model, whereas *DDDB*s generally have multiple minimal models.

To answer positive queries over *DDDB*s, it is sufficient to show the query is satisfied in every minimal model of the database. Thus, in the *DDDB,* $\{a \vee b\}$, there are two minimal models, $\{\{a\}, \{b\}\}$. The query, a? (that is, can a be derived from the database), is not satisfied in the model $\{b\}$, and hence, it cannot be concluded that a is true. However, the query, $(a \vee b)$ is satisfied in both minimal models, and hence, the answer to the query $\{a \vee b\}$ is yes. To answer negated queries, it is not sufficient to use Reiter's *CWA* since, as he noted, from $DB = \{a \vee b\}$, it is not possible to prove a, and it is not possible to prove b. Hence, by the *CWA,* not a and *not* b follow. But $\{a \vee b,\ not\ a,\ not\ b\}$ is not consistent. The Generalized Closed World Assumption (*GCWA*), developed by Minker, resolves this problem by specifying that a negated atom is true if the atom does not appear in any minimal model of the database. This provides a model theoretic definition of negation. An equivalent proof theoretic definition, also by Minker, is that an atom a is considered false if, whenever $a \vee C$ is proved true, then C can be proven true, where C is an arbitrary positive clause.

Answering queries in *DDDB*s has been studied by several individuals. Work by Fernández and Minker who developed the concept of a model tree is described. A model tree is a tree whose nodes consist of atoms. Every branch of the model tree is a model of the database. They show how one can incrementally compute sound and complete answers to queries in hierarchical *DDDB*s. An example of a model tree is shown in Fig. 1. A *DDDB* is hierarchical if it contains no recursion. One can develop a fixpoint operator over trees to capture the meaning of a *DDDB* that includes recursion. Fernández and Minker compute the model tree of the extensional *DDDB* once. To answer queries, intensional database rules may be invoked. However, the models of the extensional disjunctive part of the database do not have to be generated for each query. Their approach to compute answers generalizes both to stratified and normal *DDDB*s.

Example 8 Model Tree. Consider the following example given by the database: $\{a(1);\ a(2) \vee b(2);\ b(1) \vee b(2)\}$. There are two minimal models for this database $\{\{a(1), a(2), b(1),\}, \{a(1), b(2)\}\}$. These models may be written as a tree as shown in Fig. 1.

Loveland and his students have developed a top-down approach when the database is near Horn; that is, there are few disjunctive statements. They have developed a case-based reasoner that uses Prolog to perform the reasoning. They introduce a relevancy detection algorithm to be used with *SATCHMO,* developed by Manthey and Bry, for automated theorem proving. Their system, termed *SATCHMORE* (*SATCHMO* with relevancy), improves on *SATCHMO* by limiting uncontrolled use of forward chaining. There are currently several efforts devoted to implementing disjunctive deductive databases from a bottom-up approach.

Extended Disjunctive Deductive Databases, $Datalog^{\neg}_{disj,ext}$

Fernández and Minker developed a fixpoint characterization of the minimal models of disjunctive and stratified disjunctive deductive databases. They proved that the operator iteratively constructs the perfect models semantics (Przymusinski) of stratified *DDB*s. Given the equivalence between the perfect models semantics of stratified programs and prioritized circumscription as shown by Przymusinski, their characterization captures the meaning of the corresponding circumscribed theory. They present a bottom-up evaluation algorithm for *stratified DDDB*s. This algorithm uses the model-tree data structure to compute answers to queries. Fernández and Minker have developed the theory of *DDDB*s using the concept of model trees.

Alternative semantics were developed for non-stratifiable normal *DDDB*s by Ross (the strong well founded semantics); Baral, Lobo, and Minker (Generalized Disjunctive Well-Founded Semantics (*GDWFS*); Przymusinski (disjunctive stable model semantics); Przymusinski (stationary semantics); and Brass and Dix *(D-WFS* semantics). Przymusinski described a semantic framework for disjunctive logic programs and introduced the static expansions of disjunctive programs. The class of static expansions extends both the classes of stable, well-founded and stationary models of normal programs and the class of minimal models of disjunctive programs. Any static expansion of a program P provides the corresponding semantics for P consisting of the set of all sentences logically implied by the expansion. The *D-WFS* semantics permits a general approach to bottom-up computation in disjunctive programs.

There are a large number of different semantics, in addition to those listed here. A user who wishes to use such a system is faced with the problem of selecting the appropriate semantics for his needs. No guidelines have been developed. However, one way to assess the semantics desired is to consider the complexity of the semantics. Results have been obtained for these semantics by Schlipf and by Eiter and Gottlob.

Ben-Eliahu and Dechter showed that there is an interesting class of disjunctive databases that are tractable. In addition to work on tractable databases, consideration has been given to approximate reasoning where one may give up soundness or completeness of answers. Selman and Kautz developed lower and upper bounds for Horn (Datalog) databases, and Cadoli and del Val developed techniques for approximating and compiling databases.

A second way to determine the semantics to be used is through their properties. Dix proposed criteria that are useful to consider in determining the appropriate semantics to be used. Properties deemed to be useful are elimination of tautologies, where one wants the semantics to remain the same if a tautology is eliminated; generalized principle of partial evaluation, where if a rule is replaced by a one-step deduction, the semantics is unchanged; positive/negative reduction; elimination of non-minimal rules, where a subsumed rule is eliminated, and the semantics remains the same; consistency, where the semantics is not empty for all disjunctive databases; and independence, where if a literal l is true in a program P, and P' is a program whose language is independent of the language of P, then l remains true in the program consisting of the union of the two languages.

A semantics may have all the properties that one may desire and be computationally tractable and yet not provide answers that a user expected. If, for example, the user expected an answer $r(a)$ in response to a query $r(X)$, and the semantics were, for Example 6, the well-founded semantics, the user would receive the answer, $r(a)$ is unknown. However, if the stable model semantics had been used, the answer returned would be $r(a)$. Perhaps the best that can be expected is to provide users with complexity results and criteria by which they may decide as to which semantics meets the needs off their problems.

Understanding the semantics of disjunctive theories is related to nonmonotonic reasoning. The field of nonmonotonic reasoning has resulted in several alternative approaches to perform default reasoning. Hence, *DDDB*s may be used to compute answers to queries in such theories. Cadoli and Lenzerini developed complexity results concerning circumscription and closed world reasoning. Przymusinski, and Yuan and You describe relationships between autoepistemic circumscription and logic programming. Yuan and You use two different belief constraints to define two semantics, the stable cicumscriptive semantics and the well-founded circumscriptive semantics for autoepistemic theories.

References to work by Fernández and Minker and by Minker and Ruiz may be found in Ref. 2. Work on complexity results appears in Schlipf (23) and in Eiter and Gottlob (24,25). Relationships between $Datalog^{\neg}_{ext}$ and nonmonotonic theories may be found in Ref. 2. At the current time, there is no good source that lists prototype implementations of such databases.

IMPLEMENTATIONS OF DEDUCTIVE DATABASES

Although there have been many theoretical developments in the field of deductive databases, commercial systems have lagged behind. In the period pre-1970, several prototype systems were developed using ad hoc techniques to perform deduction. In the period 1970–1980, techniques based on the Robinson Resolution principle were developed.

During the period 1980 through the date of this article, a number of prototype systems were developed based upon the Robinson Resolution Principle and bottom-up techniques. Several efforts are described in the following paragraphs, followed by a brief description of commercial developments in progress. The commercial systems have benefited from these efforts and from the technical contributions described in this article.

The major efforts on prototype *DDB* systems since 1980 were developed at the European Computer Research Consortium (*ECRC*), at the University of Wisconsin, at Stanford University, and at the *MCC* Corporation. These efforts contributed both to the theory and implementation of *DDB*s.

Implementation efforts at *ECRC* were directed by Nicolas, started in 1984, and led to the study of algorithms and prototypes: deductive query evaluation methods (*QSQ*/*SLD* and others), integrity checking (Soundcheck) by Decker, consistency checking by Manthey and Bry (SATCHMO) (26), the deductive database system *EKS*($-$*V1*) by Vieille and his team, hypothetical reasoning and *IC*s checking, and aggregation through recursion. The EKS system used a top-down evaluation method and was released to *ECRC* shareholder companies in 1990.

Implementation efforts at *MCC,* directed by Tsur and Zaniolo, started in 1984 and emphasized bottom-up evaluation methods and query evaluation using such methods as seminaive evaluation, magic sets and counting, semantics for stratified negation and set-grouping, investigation of safety, the finiteness of answer sets, and join order optimization. The *LDL* system was implemented in 1988 and released in the period 1989–1991. It was among the first widely available *DDB*s and was distributed to universities and shareholder companies of MCC.

Implementation efforts at the University of Wisconsin, directed by Ramakrishnan, on the *Coral* DDBs started in the 1980s. Bottom-up and magic set methods were implemented. The system, written in C and C++, is extensible and provides aggregation and modularly stratified databases. *Coral* supports a declarative language, and an interface to C++ which allows for a combination of declarative and imperative programming. The declarative query language supports general Horn clauses augmented with complex terms, set-grouping, aggregation, negation, and relations with tuples that contain universally quantified variables. Coral supports many evaluation strategies and automatically chooses an efficient evaluation strategy. Users can guide query optimization by selecting from among alternative control choices. *Coral* provides imperative constructs such as update, insert, and delete rules. Disk-resident data is supported using the EXODUS storage manager, which also provides transaction management in a client-server environment.

Implementation at Stanford University, directed by Ullman, started in 1985 on *NAIL!* (Not Another Implementation of Logic!). The effort led to the first paper on recursion using the magic sets method. Other contributions were aggregation in logical rules and theoretical contributions to negation: stratified negation by Van Gelder, well-founded negation by Van Gelder, Ross, and Schlipf, and modularly stratified negation (27). A language called *GLUE* (28), developed for logical rules, has the power of *SQL* statements, as well as a conventional language for the construction of loops, procedures, and modules.

There has been considerable work on the above systems to develop efficient detection and implementation of classes of nonstratified programs. The concept of modularly-stratified programs has been implemented on *Coral,* the concept of *XY-stratification,* implemented in *LDL++,* and the related concept of *explicitly stratified programs* implemented in *Aditi.*

At the present time, two commercial *DDB* systems are under development, and some techniques from the *DDB* technology have been incorporated into relational technology. It is not surprising that after 20 years from the start of the field of *DDB*s, few commercial systems exist. It took approximately 12 years before relational systems were available commercially. As Ullman has stated on a number of occasions, deductive database theory is more subtle than relational database theory.

The two systems nearing completion as commercial products are *Aditi,* under development at the University of Melbourne, and *VALIDITY* whose development started at the Bull Corporation. According to a personal communication from Ramamohanarao, leader of the *Aditi* effort, the beta release of the system is scheduled for December 1997. *Aditi* handles stratified databases, recursion, and aggregation in stratified databases. It optimizes recursion with magic sets and seminaive evaluation. The system interfaces with *Prolog.*

At the Bull Corporation, Nicolas and Vieille headed an effort to develop the *VALIDITY DDB* system that integrates object-oriented features. *VALIDITY* was started in approximately 1992 and is an outgrowth of the work at *ECRC.* According to a personal communication from Nicolas, *VALIDITY* is now being further developed and marketed by Next Century Media, Inc., a California corporation in which Groupe Bull has some equity interests.

The *VALIDITY* software platform is currently used mainly to develop NCM's products in electronic media for interactive media applications. Two of these products enable marketers to target their advertising messages to household clusters, to individual households, and to specific consumers, based on the user's expressed and implied interests and preferences, and to convert the data coming from the user into a database of ongoing and useful information about these customers. A third product enables marketers to measure the effectiveness of their media plan and expenditures in a timely manner, based on a full census of the entire audience, rather than on samples which are fraught with inherent biases and errors. Other DDB applications can be found in the book edited by Ramakrishnan (29).

Many techniques introduced within *DDB*s are finding their way into relational technology. The new SQL standards for relational databases are beginning to adopt many of the powerful features of *DDB*s. In the SQL-2 standards (also known as SQL-92), a general class of *ICS,* called *asserts,* allow for arbitrary relationships between tables and views to be declared. These constraints exist as separate statements in the database and are not attached to a particular table or view. This extension is powerful enough to express the types of *IC*s generally associated with *DDB*s. However, only the full SQL-2 standard includes assert specifications. The intermediate SQL-2 standard, the basis for most current commercial implementations, does not include asserts. The relational language for the next generation *SQL*, *SQL3*, currently provides an operation called recursive union that supports recursive processing of tables. The use of the recursive union operator allows both linear (single-parent or tree) recursion and nonlinear (multiparent, or general directed graph) recursion.

Linear recursion is currently a part of the client server of IBM's *DB2* system. They are using the magic sets method to perform linear recursion. Indications are that the *ORACLE* database system will support some form of recursion.

A further development is that semantic query optimization is being incorporated into relational databases. In *DB2,* cases are recognized when only one answer is to be found, and the search is terminated. In other systems, equalities and other arithmetic constraints are being added to optimize search. One can envision the use of join elimination in *SQO* to be introduced to relational technology. One can now estimate when it will be useful to eliminate a join. The tools and techniques already exist, and it is merely a matter of time before users and system implementers have them as part of their database systems.

Detailed descriptions of contributions made by these systems and others may be found in Ref. 30. A description of some implementation techniques in *Datalog* and recursion in *SQL3* may be found in Refs. 31–33.

SUMMARY AND REFERENCES

The article describes the prehistory of deductive databases starting from 1957 to approximately 1970. The use of a general rule of inference, based upon the Robinson Resolution Principle, developed by J. A. Robinson (34), started in 1968 with the work of Green and Raphael (35,36), led to a number of systems, and culminated in the start of the field in November, 1977, with a Workshop held in Toulouse, France that resulted in the appearance of a book edited by Gallaire and Minker (3). The publication of books based on subsequent Toulouse workshops (37,38) and, in 1984, of the survey paper by Gallaire, Minker, and Nicolas (39) were other landmark events in the history of the field.

The field has progressed rapidly and has led to an understanding of negation and has provided a theoretical framework so that it is well-understood what is meant by a query and an answer to a query. The field of relational databases is encompassed by the work in *DDB*s. Complex knowledge based systems can be implemented using the technology. There are, however, many different kinds of *DDB*s as described in this article. Theoretical results concerning fixpoint theory for *DDB*s may be found in Lloyd (13), while fixpoint theory and theories of negation for disjunctive deductive databases may be found in Lobo, Minker, and Rajasekar (40). Complexity results have not been summarized in this paper. A summary of complexity results is presented in Ref. 2. The least complex *DDB*s are, in order, *Datalog*, $Datalog^{\neg}_{str}$, $Datalog^{\neg}_{wfs}$, and $Datalog^{\neg}_{stab}$. The first three databases result in unique minimal models. Other databases are more complex and, in addition, there are no current semantics that are uniformly agreed upon for $Datalog_{disj}$. As noted earlier, a combination of properties of DDBs, developed by Dix and discussed in Ref. 41, and the complexity of these systems, as described in Refs. 23–25, could be used once such systems are developed.

As of the early 1990s, various efforts have been made to enrich Deductive Database Systems with object-oriented features. The main results on this topic can be found in the Proceedings of the DOOD Conference series on Deductive and Object-Oriented Databases (42–46) co-established in 1989 by Shojiro Nishio, Serge Abiteboul, Jack Minker, and Jean-Marie Nicolas.

There are many topics in deductive databases that have not been covered in this article. For references trends to topics such as uncertainty, time, active databases, see Ref. 2.

The book (12) illustrates the current shift from the Relational Model to the Deductive/Logic Database Model as the reference model for investigating theoretical issues. The implementation of tools developed from the Deductive/Logic Database Model, such as the ability to handle recursion and semantic query optimization in the Relational Model provides further evidence of this trend.

BIBLIOGRAPHY

1. J. Minker, Perspectives in deductive databases, *J. Logic Program.,* **5**: 33–60, 1988.
2. J. Minker, Logic and databases: A 20 year retrospective, In D. Pedreschi and C. Zaniolo (eds.), *Logic in Databases,* New York: Springer, 1996, pp. 3–57.
3. H. Gallaire and J. Minker (eds.), *Logic and Databases,* New York: Plenum, 1978.
4. L. Vieille, Recursive axioms in deductive databases: The Query/SubQuery approach, *Proc. 1st Int. Conf. Expert Database Syst.,* 1986, pp. 253–267.
5. L. Vieille, Recursive query processing: The power of logic, *Theor. Comput. Sci.,* **69**: 1989.
6. S. W. Dietrich and D. S. Warren, Extension tables: Memo relations in logic programming, *Proc. Symp. Logic Program.,* San Francisco, CA, 1987, pp. 264–273.
7. F. Bancilhon et al., Magic sets and other strange ways to implement logic programs, *Proc. ACM Symp. Prin. Database Sys.,* 1986.
8. C. Beeri and R. Ramakrishnan, On the power of magic, *J. Logic Program.,* **10** (3/4): 255–300, 1991.
9. F. Bry, Query evaluation in recursive databases: Bottom-up and top-down reconciled, *Data Knowl. Eng.,* pp. 289–312, 1990.
10. J. D. Ullman, *Principles of Database and Knowledge-Base Systems I,* Rockville, MD: Computer Science Press, 1988.
11. J. D. Ullman, *Principles of Database and Knowledge-Base Systems II,* Rockville, MD: Computer Science Press, 1988.
12. S. Abiteboul, Y. Sagiv, and V. Vianu, *Foundations of Databases,* Reading, MA: Addison-Wesley, 1995.
13. J. W. Lloyd, *Foundations of Logic Programming,* 2nd ed., Berlin: Springer-Verlag, 1987.
14. T. Gaasterland, P. Godfrey, and J. Minker, An overview of cooperative answering, *J. Intell. Inf. Syst.,* **1** (2): 123–157, 1992 (invited paper).
15. I. Niemela and P. Simons, Smodels—an implementation of the stable model and well-founded semantics for normal lp. Submitted for publication, 1997.
16. W. Marek, A. Nerode, and J. Remmel, A theory of nonmonotonic rule systems II. *Ann. Math. Artif. Intell.* **5**: 229–263, 1992.
17. J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming.* San Mateo, CA: Morgan Kaufmann, 1988.
18. A. Van Gelder, K. Ross, and J. S. Schlipf, Unfounded sets and well-founded semantics for general logic programs, *Proc. 7th Symp. Prin. Database Syst.* 1988, pp. 221–230.
19. M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, *Proc. 5th Int. Conf. Symp. Logic Program.,* Seattle, WA, 1988, pp. 1070–1080.
20. J. S. Schlipf, Complexity and undecideability results for logic programming, *Ann. Math. Artif. Intell.,* **15** (3-4): 257–288, 1995.
21. J. C. Shepherdson, Negation in logic programming. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming,* San Mateo, CA: Morgan Kaufmann, 1988, pp. 19–88.
22. C. Baral and M. Gelfond, Logic programming and knowledge representation, *J. Logic Program.,* **19/20**: 73–148, 1994.
23. J. S. Schlipf, A survey of complexity and undecidability results in logic programming, in H. Blair et al. (eds.), *Informal Proceedings of the Workshop on Structural Complexity and Recursion-theoretic Methods in Logic Programming,* Washington, DC: 1992, pp. 143–164.
24. T. Eiter and G. Gottlob, Complexity aspects of various semantics for disjunctive databases, *Proc. 12th ACM SIGART-SIGMOD-SIGART Symp. Princ. Database Syst. (PODS'93),* 1993, pp. 158–167.
25. T. Eiter and G. Gottlob, Complexity results for disjunctive logic programming and application to nonmonotonic logics, *Proc. Int. Logic Program. Symp. (ILPS'93),* Vancouver, BC, Canada, 1993, pp. 266–278.
26. R. Manthey and F. Bry, Satchmo: A theorem prover implemented in Prolog, *Proc. 9th Int. Conf. Autom. Deduct. (CADE),* 1988.
27. K. A. Ross, Modular stratification and magic sets for datalog programs with negation, *Proc. ACM Symp. Princ. Database Syst.,* 1990.
28. S. Morishita, M. Derr, and G. Phipps, Design and implementation of the Glue-Nail database system, *Proc. ACM-SIGMOD'93 Conf.,* 1993, pp. 147–167.
29. R. Ramakrishnan, *Applications of Logic Databases,* Boston: Kluwer Academic Publishers, 1995.
30. R. Ramakrishnan and J. D. Ullman, A survey of research on deductive database systems, *J. Logic Program.,* **23** (2): 125–149, 1995.
31. R. Ramakrishnan, *Database Management Systems,* New York: McGraw-Hill, 1997.

32. C. Zaniolo, *Advanced Database Systems,* San Mateo, CA: Morgan Kaufmann, 1997.
33. J. D. Ullman and J. Widom (eds.), *A First Course in Database Systems.* Upper Saddle River, NJ: Prentice-Hall, 1997.
34. J. A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM,* **12** (1): 23–41, 1965.
35. C. C. Green and B. Raphael, Research in intelligent question answering systems, *Proc. ACM 23rd Natl. Conf.,* 1968, pp. 169–181.
36. C. C. Green and B. Raphael, The use of theorem-proving techniques in question-answering systems, *Proc. 23rd Natl. Conf. ACM,* 1968.
37. H. Gallaire, J. Minker, and J-M. Nicolas (eds.), *Advances in Database Theory,* Vol. 1, New York: Plenum, 1981.
38. H. Gallaire, J. Minker, and J-M. Nicolas (eds.), *Advances in Database Theory,* Vol. 2, New York: Plenum, 1984.
39. H. Gallaire, J. Minker, and J-M. Nicolas, *Logic and Databases: A Deductive Approach,* Vol. 16 (2), New York: ACM Computing Surveys, 1984.
40. J. Lobo, J. Minker, and A. Rajasekar, *Foundations of Disjunctive Logic Programming,* Cambridge, MA: MIT Press, 1992.
41. G. Brewka, J. Dix, and K. Konolige, *Nonmonotonic Reasoning: An Overview,* Stanford, CA: Center for the Study of Language and Information, 1997.
42. W. Kim, J.-M. Nicolas, and S. Nishio (eds.), *Proc. 1st Int. Conf. Deduct. Object-Oriented Databases (DOOD'89),* 1990.
43. C. Delobel, M. Kifer, and Y. Masunaga (eds.), *Proc. 2nd Int. Conf. Deduct. Object-Oriented Databases (DOOD'91),* 1991.
44. S. Ceri, K. Tanaka, and S. Tsur (eds.), *Proc. 3rd Int. Conf. Deduct. Object-Oriented Databases (DOOD'93),* December, 1993.
45. T-W. Ling, A. Mendelzon, and L. Vieille (eds.), *Proc. 4th Int. Conf. Deduct. Object-Oriented Databases (DOOD'95),* 1995, LNCS 1013.
46. F. Bry, R. Ramakrishnan, and K. Ramamohanarao (eds.), *Proc. 5th Int. Conf. Deduct. Object-Oriented Databases (DOOD'97),* 1997.

JACK MINKER
University of Maryland