What You Always Wanted to Know About Datalog (And Never Dared to Ask)

STEFANO CERI, GEORG GOTTLOB, AND LETIZIA TANCA

Abstract—**Datalog is a database query language based on the logic programming paradigm; it has been designed and intensively studied over the last five years. We present the syntax and semantics of Datalog and its use for querying a relational database. Then, we classify optimization methods for achieving efficient evaluations of Datalog queries, and present the most relevant methods. Finally, we discuss various exhancements of Datalog, currently under study, and indicate what is still needed in order to extend Datalog's applicability to the solution of real-life problems. The aim of this paper is to provide a survey of research performed on Datalog, also addressed to those members of the database community who are not too familiar with logic programming concepts.**

Index Terms—**Deductive databases, logic programming, recursive queries, relational databases, query optimization.**

I. Introduction

Recent years have seen substantial efforts in the direction of merging artificial intelligence and database technologies for the development of large and persistent knowledge bases. An important contribution towards this goal comes from the integration of logic programming and databases. The focus has been mostly concentrated by the database theory community on well-formalized issues, like the definition of a new rule-based language, called Datalog, specifically designed for interacting with large databases; and the definition of optimization methods for various types of Datalog rules, together with the study of their efficiency [120], [15], [16]. In parallel, various experimental projects have shown the feasibility of Datalog programming environments [119], [24], [88].[1]

Present efforts in the integration of artificial intelligence and databases take a much more basic and pragmatic approach; in particular, several attempts fall in the category of "loose coupling," where existing AI and DB environments are interconnected through ad-hoc interfaces. In other cases, AI systems have solved persistency issues by developing internal databases for their tools; but these internal databases typically do not allow data sharing and recovery, thus do not properly belong to current database technology [26]. The spread and success of such enhanced AI systems, however, indicate that there is a great need for them.

Loose coupling has been attempted in the area of Logic Programming and databases as well, by interconnecting Prolog systems to relational databases [42], [25], [46], [64], [33], [97].

Although interesting results have been achieved, most studies indicate that simple interfaces are too inefficient; an enhancement in efficiency is achieved by intelligent interfaces [64], [33]. This indicates that loose coupling might in fact solve today's problems, but on the long range, strong integration is required. More generally, we expect that knowledge base management systems will provide direct access to data and will support rule-based interaction as one of the programming paradigms. Datalog is a first step in this direction.

The reaction of the database community to Datalog has often been marked by skepticism; in particular, the immediate or even future practical use of research on sophisticated rule-based interactions has been questioned [94]. Nevertheless, we do expect that Datalog's experience, properly filtered, will teach important lessons to researchers involved in the development of knowledge base systems. The purpose of this paper is to give a self-contained survey of the research that has been performed recently on Datalog. More exhaustive treatments can be found in the books [34], [122], and [92]. Other interesting survey papers, approaching the subject from different perspectives, are [53] and [100].

This paper is organized as follows. In Section II we present the foundations of Datalog: the syntactic structure and the semantics of Datalog programs. In Section III we explain how Datalog is used as a query language over relational databases; in particular, we indicate how Datalog can be immediately translated to equations of relational algebra. In Section IV we present a taxonomy of the various optimization methods, emphasizing the distinction between program transformation and evaluation methods. In Section V, we present a survey of evaluation methods and program transformation techniques, selecting the most representative ones within the classes outlined in Section IV. In Section VI, we present several formal extensions given to the Datalog language to enhance its expressiveness, and we survey some of the current research projects

Manuscript received March 1, 1989. This paper is based on Parts II and III of the book by S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases* (New York: Springer-Verlag, to be published).

S. Ceri is with the Dipartimento di Matematica, Universita di Modena, Modena, Italy.

G. Gottlob is with the Institut fur Angenwandte Informatik, TU, Wien, Austria.

L. Tanca is with the Dipartimento di Elettronica, Politecnico di Milano, Milano, Italy.

IEEE Log Number 8928155.

[1]The term "Datalog" was also used by Maier and Warren in [82] to denote a subset of Prolog.

1041-4347/89/0300-0146$01.00 © 1989 IEEE

in this area. Finally, in Section VII we attempt an evaluation of what will be required in Datalog in order to become more attractive and usable.

II. Datalog: Semantics and Evaluation Paradigms

In this section we define the syntax of the Datalog query language and explain its logical semantics. We give a model-theoretic characterization of Datalog programs and show how they can be evaluated in a bottom-up fashion. This evaluation corresponds to computing a least fixpoint. Finally, we briefly mention another evaluation paradigm called "top-down evaluation," and compare Datalog to the well-known logic programming language *Prolog*.

A. The Syntax of Datalog Programs

Datalog is in many respects a simplified version of general *Logic Programming* [78]. A logic program consists of a finite set of *facts* and *rules*. Facts are assertions about a relevant piece of the world, such as: "John is the father of Harry". Rules are sentences which allow us to deduce facts from other facts. An example of a rule is: "If X is a parent of Y and if Y is a parent of Z, then X is a grandparent of Z". Note that rules, in order to be general, usually contain *variables* (in our case, X, Y, and Z). Both facts and rules are particular forms of *knowledge*.

In the formalism of Datalog both facts and rules are represented as *Horn clauses* of the general shape

$$L_0 :- L_1, \ldots, L_n$$

where each L_i is a *literal* of the form $p_i(t_1, \ldots, t_{k_i})$ such that p_i is a *predicate symbol* and the t_j are *terms*. A term is either a *constant* or a *variable*.[2] The left-hand side (LHS) of a Datalog clause is called its *head* and the right-hand side (RHS) is called its *body*. The body of a clause may be empty. Clauses with an empty body represent facts; clauses with at least one literal in the body represent rules.

The fact "John is the father of Bob", for example, can be represented as *father*(*bob*, *john*). The rule "If X is a parent of Y and if Y is a parent of Z, then X is a grandparent of Z" can be represented as

$$grandpar(Z, X) :- par(Y, X), par(Z, Y).$$

Here the symbols *par* and *grandpar* are predicate symbols, the symbols *john* and *bob* are constants, and the symbols X, Y, and Z are variables. We will use the following notational convention: constants and predicate symbols are strings beginning with a lower-case character; variables are strings beginning with an upper-case character. Note that for a given Datalog program it is always clear from the context whether a particular nonvariable symbol is a constant or a predicate symbol. We require that all literals with the same predicate symbol are of the same *arity*, i.e., that they have the same number of arguments. A literal, fact, rule, or clause which does not contain any variables is called *ground*.

Any Datalog program P must satisfy the following *safety conditions*.

- Each fact of P is ground.
- Each variable which occurs in the head of a rule of P must also occur in the body of the same rule.

These conditions guarantee that the set of all facts that can be derived from a Datalog program is finite.

B. Datalog and Relational Databases

In the context of general Logic Programming it is usually assumed that all the knowledge (facts and rules) relevant to a particular application is contained within a single logic program P. Datalog, on the other hand, has been developed for applications which use a large number of facts stored in a relational database. Therefore, we will always consider two sets of clauses: a set of ground facts, called the *Extensional Database* (*EDB*), physically stored in a relational database, and a Datalog program P called the *Intensional Database* (*IDB*).[3] The predicates occurring in the EDB and in P are divided into two disjoint sets: the *EDB-predicates*, which are all those occurring in the extensional database and the *IDB-predicates*, which occur in P but not in the EDB. We require that the head predicate of each clause in P be an IDB-predicate. EDB-predicates may occur in P, but only in clause bodies.

Ground facts are stored in a relational database; we assume that each EDB-predicate r corresponds to exactly one relation R of our database such that each fact $r(c_1, \ldots, c_k)$ of the EDB is stored as a tuple $< c_1, \ldots, c_n >$ of R.

Also the IDB-predicates of P can be identified with relations, called *IDB-relations*, or also *derived relations*, defined through the program P and the EDB. IDB relations are not stored explicitly, and correspond to relational *views*. The materialization of these views, i.e., their effective (and efficient) computation, is the main task of a Datalog compiler or interpreter.

As an example of a relational EDB, consider a database E_1 consisting of two relations with respective schemes *PERSON*(*NAME*) and *PAR*(*CHILD*, *PARENT*). The first contains the names of persons and the second expresses a parent relationship between persons. Let the actual instances of these relations have the following values:

$$\begin{aligned} PERSON = \{&< ann >, < bertrand >, < charles >, \\ &< dorothy >, < evelyn >, < fred >, \\ &< george >, < hilary >\} \end{aligned}$$

$$\begin{aligned} PAR = \{&< dorothy, george >, < evelyn, george >, \\ &< bertrand, dorothy >, < ann, dorothy >, \\ &< ann, hilary >, < charles, evelyn >\}. \end{aligned}$$

[2]Note that in general Logic Programming, a term can also be a complex nested structure made of function symbols, constants, and variables. Extensions of Datalog to cover such complex terms are outlined in Section VI.

[3]In the literature an IDB is usually defined as a set of *rules*. Here the Datalog program P may also contain *facts*. This terminological extension is harmless.

These relations express the set of ground facts $E = \{person(ann), person(bertrand), \ldots, par(dorothy, george), \ldots, par(charles, evelyn)\}$.

Let P_1 be a Datalog program consisting of the following clauses:

r_1: $sgc(X, X) :- person(X)$.

r_2: $sgc(X, Y) :- par(X, X1), sgc(X1, Y1), par(Y, Y1)$.

Due to rule r_1, the derived relation *SGC* will contain a tuple $< p, p >$ for each person p. Rule r_2 is recursive and states that two persons are same generation cousins whenever they have parents which are in turn same generation cousins. Further examples of tuples belonging to *SGC* are: $< dorothy, evelyn >$, $< charles, ann >$, and $< ann, charles >$.

Note that program P_1 can be considered as a *query* against the EDB E_1, producing as answer the relation *SGC*. In this setting, the distinction between the two sets of clauses, E and P, makes yet more sense. Usually a database (in our case the EDB) is considered as a time-varying collection of information. A query (in our case, a program P), on the other hand, is a time-invariant mapping which associates a result to each possible database state. For this reason we will formally define the semantics of a Datalog program P as a mapping from database states to result states. The database states are collections of EDB-facts and the result states are IDB-facts.

Usually Datalog programs define large IDB-relations. It often happens that a user is interested in a subset of these relations. For instance, he or she might want to know the same generation cousins of Ann only rather than all same generation cousins of all persons in the database. To express such an additional constraint, one can specify a *goal* to a Datalog program. A goal is a single literal preceded by a question mark and a dash, for example, in our case, $?\text{-}sgc(ann, X)$. Goals usually serve to formulate ad hoc queries against a view defined by a Datalog program.

C. The Logical Semantics of Datalog

Each Datalog fact F can be identified with an atomic formula F^* of First-Order Logic. Each Datalog rule R of the form $L_0 :- L_1, \ldots, L_n$ represents a first-order formula R^* of the form $\forall X_1, \ldots \forall X_m(L_1 \wedge \ldots \wedge L_n \Rightarrow L_0)$, where $X_1, \ldots, X_m$ are all the variables occurring in R. A set S of Datalog clauses corresponds to the conjunction S^* of all formulas C^* such that $C \in S$.

The *Herbrand Base HB* is the set of all facts that we can express in the language of Datalog, i.e., all literals of the form $P(c_1, \ldots, c_k)$ such that all c_i are constants. Furthermore, let *EHB* denote the extensional part of the Herbrand base, i.e., all literals of *HB* whose predicate is an EDB-predicate and, accordingly, let *IHB* denote the set of all literals of *HB* whose predicate is an IDB-predicate. If S is a finite set of Datalog clauses, we denote by $cons(S)$ the set of all facts that are logical consequences of S^*.

The semantics of a Datalog program P can now be described as a mapping $\mathfrak{M}_P$ from *EHB* to *IHB* which to each possible extensional database $E \subseteq EHB$ associates the set $\mathfrak{M}_P(E)$ of intensional "result facts" defined by $\mathfrak{M}_P(E) = cons(P \cup E) \cap IHB$.

Let K and L be two literals (not necessarily ground). We say that K *subsumes* L, denoted by $K \rhd L$, if there exists a substitution θ of variables such that $K\theta = L$, i.e., if θ applied to K yields L. If $K \rhd L$ we also say that L is an *instance* of K. For example, $q(a, b, b)$ and $q(c, c, c)$ are both instances of $q(X, Y, Y)$, but $q(b, b, a)$ is not.

When a goal "?- G" is given, then the semantics of the program P with respect to this goal is a mapping $\mathfrak{M}_{P,G}$ from *EHB* to *IHB* defined as follows

$$\forall E \subseteq EHB\ \mathfrak{M}_{P,G}(E) = \{H \mid H \in \mathfrak{M}_P(E) \wedge G > H\}.$$

D. The Model Theory of Datalog

Model theory is a branch of mathematical logic which defines the semantics of formal systems, by considering their possible interpretations, i.e., the different intended meanings that the used symbols and formulas may assume. Early developments of general model theory date back to works of Löwenheim [79], Skolem [113], and Gödel [56]. A comprehensive exposition is given in [39]. The model theoretic characterization of general logic theories often requires the use of quite sophisticated tools of modern algebra [83]. The model theory of Horn clause systems and logic programs, however, is less difficult. It has been developed in [125], [11] and is also explained in [78]. Datalog, being a simplified version of Logic Programming, can be described very easily in terms of model theory.

An *interpretation* (in the context of Datalog) consists of an assignment of a concrete meaning to constant and predicate symbols. A Datalog clause can be interpreted in several different ways. A clause may be true under a certain interpretation and false under another one. If a clause C is true under a given interpretation, we say that this interpretation *satisfies* C.

The concept of logical consequence, in the context of Datalog, can be defined as follows: a fact F follows logically from a set S of clauses, iff each interpretation satisfying every clause of S also satisfies F. If F follows from S, we write $S \models F$.

Note that this definition captures quite well our intuitive understanding of logical consequence. However, since general interpretations are quite unhandy objects, we will limit ourselves to consider interpretations of a particular type, called *Herbrand Interpretations*. The recognition that we may forget about all other interpretations is due to the famous logicians Löwenheim, Skolem, and Herbrand.[4]

A Herbrand interpretation assigns to each constant symbol "itself", i.e., a lexicographic entity. Predicate symbols are assigned predicates ranging over constant symbols. Thus, two nonidentical Herbrand interpretations differ only in the respective interpretations of the predicate symbols. For instance, one Herbrand interpretation may satisfy the fact $l(t, q)$ and another one may not sat-

[4]Actually, Herbrand interpretations can be defined in the much more general context of Clausal Logic [39], [80] and Logic Programming [125], [78].

isfy this fact. For this reason, any Herbrand interpretation can be identified with a subset $\mathcal{I}$ of the Herbrand base *HB*. This subset contains all the ground facts which are true under the interpretation. Thus, a ground fact $p(c_1, \ldots, c_n)$ is true under the interpretation $\mathcal{I}$ iff $p(c_1, \ldots, c_n) \in \mathcal{I}$. A *Datalog* rule of the form L_0: $-L_1, \ldots, L_n$ is true under $\mathcal{I}$ iff for each substitution θ which replaces variables by constants, whenever $L_1\theta \in \mathcal{I} \wedge \ldots \wedge L_n\theta \in \mathcal{I}$, then it also holds that $L_0\theta \in \mathcal{I}$.

A Herbrand interpretation which satisfies a clause C or a set of clauses S is called a *Herbrand model* for C or, respectively, for S.

Consider, for example, the Herbrand interpretations $\mathcal{I}_1 = \{person(john), person(jack), person(jim), sgc(john, john), par(jim, john), par(jack, john), sgc(john, john), sgc(jack, jack), sgc(jim, jim)\}$. Obviously, $\mathcal{I}_1$ is not a Herbrand model of the program P_1 given in Section II-B. Let $\mathcal{I}_2 = \mathcal{I}_1 \cup \{sgc(jim, jack), sgc(jack, jim)\}$. It is easy to see that $\mathcal{I}_2$ is a Herbrand model of P_1.

The set $cons(S)$ of all consequence facts of a set S of Datalog clauses can thus be characterized as follows: $cons(S)$ is the set of all ground facts which are satisfied by each Herbrand model of S. Since a ground fact F is satisfied by a Herbrand interpretation $\mathcal{I}$ iff $F \in \mathcal{I}$, $cons(S)$ is equal to the intersection of all Herbrand models of S. Summarized

$$cons(S) = \{F \in HB \mid S \models F\} = \bigcap \{\mathcal{I} \mid \mathcal{I} \text{ is a Herbrand model of } S\}.$$

Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two Herbrand models of S. It is easy to see that their intersection $\mathcal{I}_1 \cap \mathcal{I}_2$ is also a Herbrand model of S. More generally, it can be shown that the intersection of an arbitrary (possibly infinite) number of Herbrand models of S always yields a Herbrand model of S. This property is called the *model intersection property*. Note that the model intersection property not only holds for Datalog clauses, but for a more general type of clause called *definite Horn clauses* [125], [78].

From the model intersection property it follows, in particular, that for each set S of Datalog clauses, $cons(S)$ is a Herbrand model of S. Since $cons(S)$ is a subset of any other Herbrand model of S, we call $cons(S)$ the *least Herbrand model of S*.

Up to here nothing has been said about how $cons(S)$ can be *computed*. The following subsections as well as Sections III–V deal with this problem. In the rest of this section we will remain at a quite abstract level and we will not care about implementation and storage issues. In particular, we will ignore that a set S of Datalog clauses may consist of program clauses and of EDB-clauses. The problem of retrieving facts from an EDB is deferred to Sections III–V.

E. The Proof Theory of Datalog

Proof theory is concerned with the analysis of logical inference. As a branch of mathematical logic, this discipline was founded by Hilbert [60]. While the investigation of relevant extensions of first-order logic (e.g., number theory) is faced with considerable problems such as consistency and incompleteness issues [55], [57], the proof theoretic analysis of subformalisms of first-order logic, such as Logic Programming or Datalog, is easier and much less ambitious, and is more oriented towards algorithmic issues.

In this section we show how Datalog rules can be used to produce new facts from given facts. We define the notion of "fact inference" and introduce a proof-theoretic framework which allows one to infer all ground facts which are consequences of a finite set of Datalog clauses.

Consider a Datalog rule R of the form $L_0 :- L_1, \ldots, L_n$ and a list of ground facts $F_1, \ldots, F_n$. If a substitution θ exists such that for each $1 \leq i \leq n$ $L_i\theta = F_i$ then, from rule R and from the facts $F_1, \ldots, F_n$, we can *infer* in one step the fact $L_0\theta$. The inferred fact may be either a new fact or it may be already known.

What we have just described is a general inference rule, which produces new Datalog facts from given Datalog rules and facts. We refer to this rule as the *Elementary Production Principle* (*EPP*). In some sense, EPP can be considered as being a *meta-rule*, since it is independent of any particular Datalog rules, and treats them just as syntactic entities.

For example, consider the Datalog rule r_1 of program P_1:

$$r_1: sgc(X, X) :- person(X).$$

From this rule and from the fact $person(george)$ we can infer in one step $sgc(george, george)$. The substitution used here was $\theta = \{X \leftarrow george\}$. Now recall the second rule of P_1

$$r_2: sgc(X, Y) :- par(X, X1), sgc(X1, Y1), par(Y, Y1)$$

and consider the facts $par(dorothy, george)$, $sgc(george, george)$, and $par(evelyn, george)$. By applying EPP and using the substitution $\theta = \{X \leftarrow dorothy, Y \leftarrow evelyn, X_1 \leftarrow george, Y_1 \leftarrow george\}$, we infer in one step $sgc(dorothy, evelyn)$.

Let S be a set of Datalog clauses. Informally, a ground fact F can be *inferred from S*, denoted by $S \vdash F$ iff either $F \in S$ or F can be obtained by applying the inference rule EPP a finite number of times. The relationship "$\vdash$" is more precisely defined by the following recursive rules:

- $S \vdash F$ if $F \in S$.
- $S \vdash F$ if a rule $R \in S$ and ground facts $F_1, \ldots, F_n$ exist such that $\forall 1 \leq i \leq n$ $S \vdash F_i$ and F can be inferred in one step by the application of EPP to R and $F_1, \ldots, F_n$.

The sequence of applications of EPP which is used to infer a ground fact F from S is called a *proof* of F from S. Any proof can be represented as a *proof tree* with different levels and with the derived fact F at the top.

Let S_1 denote the set of all clauses present in our example-program P_1 and in our example-database E_1, i.e., $S_1 = P_1 \cup E_1$. It is easy to see that $S_1 \vdash sgc(ann, charles)$. The corresponding proof-tree is depicted in Fig. 1.

Now that we have a proof-theoretic framework which allows us to infer new ground facts from an original set

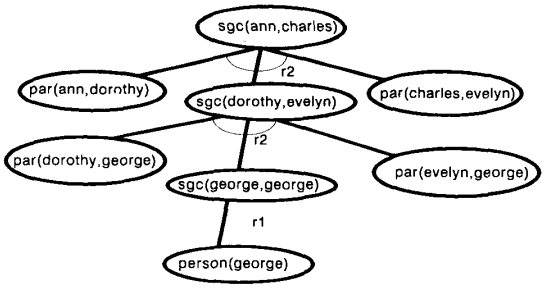


Fig. 1. Proof tree of *sgc* (*ann*, *charles*).

of Datalog clauses S, let us compare this approach to the model-theoretic approach presented in the last subsection. The following important theorem holds.

Soundness and Completeness Theorem: Let S be a set of Datalog clauses and let F be a ground fact. $S \vdash F$ iff $S \models F$.

A proof of this theorem can be found in [34].

In order to check whether EPP applies to a rule R of the form $L_0 :- L_1, \ldots, L_n$ and to a (ordered) list of ground facts $F_1, \ldots, F_n$, one has to find an appropriate substitution θ such that for each $1 \leq i \leq n$ $L_i\theta = F_i$. Such a substitution can be found by *matching* each L_i in the rule body against the corresponding fact F_i. Such a matching either fails or provides a "local" substitution θ_i for all variables occurring in L_i. If at least one matching fails or if the local substitutions are not mutually compatible because they assign different constants to the same variable, then EPP does not apply. If all matchings are successful and if the substitutions are all compatible, then we obtain the global substitution θ by forming the union of all local substitutions $\theta = \theta_1 \cup \cdots \cup \theta_n$. (Note that matching is a particular form of *unification* [34].)

There exists a very simple method of computing $cons(S)$ for each finite set of Datalog *clauses*:

FUNCTION INFER(S)
INPUT: a finite set S of Datalog clauses
OUTPUT: $cons(S)$
BEGIN
$W := S$;
WHILE EPP applies to some rule and facts of W
producing a new ground fact $F \notin W$
DO $W := W \cup \{F\}$;
RETURN($W \cap HB$) /* all facts of W, but not the rules */
END.

The INFER algorithm always terminates and produces as output a *finite* set of facts $cons(S)$. If we assume that the arities of all predicates that may occur in a Datalog program are bound by a constant, then the output of the INFER algorithm and its runtime are both polynomial in the size of its input S.

The order in which INFER generates new facts corresponds to the bottom-up order of proof trees. For this reason, the principle underlying INFER is called *bottom-up evaluation*. In the terminology of artificial intelligence this principle is also referred to as *forward chaining*, because the Datalog rules are processed *forward*, i.e., in the sense of the logical implication sign, from premises to conclusions.

F. Fixpoint Characterization of Cons (S)

Let us show that the set $cons(S)$ can be characterized as the least fixpoint of a mapping T_S from $\mathcal{P}(HB)$ to $\mathcal{P}(HB)$, where $\mathcal{P}(HB)$ denotes the powerset of HB.

If S is a set of Datalog clauses, then let $infer1(S)$ denote the set of all ground facts which can be inferred in one step from the rules and facts of S through EPP. Furthermore, let $FACTS(S)$ denote the set of all facts of S and let $RULES(S)$ denote the set of all rules of S. Obviously, we have $S = FACTS(S) \cup RULES(S)$.

The *transformation* T_S *associated to* S is a mapping from $\mathcal{P}(HB)$ to $\mathcal{P}(HB)$ defined as follows:

$$\forall W \subseteq HB: T_S(W) = W \cup FACTS(S) \cup infer1(RULES(S) \cup W).$$

It is easy to see that any Herbrand interpretation $\mathfrak{I} \subseteq HB$ is a Herbrand model of S iff $\mathfrak{I}$ is a *fixpoint* of T_S, i.e., $T_S(\mathfrak{I}) = \mathfrak{I}$. In particular, the least Herbrand model $cons(S)$ is the *least fixpoint* of T_S.

$cons(S)$ can be computed by *least fixpoint iteration*, i.e., by computing in order, $T_S(\varnothing)$, $T_S(T_S(\varnothing))$, $T_S(T_S(T_S(\varnothing)))$, $\cdots$, until one term is equal to its predecessor. This final term is $cons(S)$, the least fixpoint of S. This computation is quite similar to the INFER algorithm, however, instead of adding *one* new fact at each iteration step, a *set of new facts* are added: all those new facts which can be inferred in one step from the old ones.

There exists even a transformation T'_S which is somewhat less complicated than T_S and has the same least fixpoint

$$\forall W \subseteq HB: T'_S(W) = FACTS(S) \cup infer1(RULES(S) \cup W).$$

Thus, T'_S differs from T_S by the omission of the term W in the union.

Although both T_S and T'_S have the same *least* fixpoint $cons(S)$, they do not have the same set of fixpoints. In particular, not all Herbrand models of S are fixpoints of T'_S.

G. Top-Down Evaluation of Datalog Goals

The *top-down* method is a radically different way of evaluating Datalog programs. Proof trees are constructed from the top to the bottom. This method is particularly appropriate when a goal is specified together with a Datalog program.

Consider the program P_1 and the EDB E_1 of our "same-generation" example. Assume that the goal ?-$sgc(ann, X)$ is specified. One possibility of finding the required answers is to compute first the entire set $cons(P_1 \cup E_1)$ by bottom-up evaluation and then throw away all facts which are not subsumed by our goal. This would be a

noticeable waste of energy, since we would derive much more facts than necessary.

The other possibility is to start with the goal and construct proof trees from the top to the bottom by applying EPP "backwards". In the context of artificial intelligence such methods are also referred to as *backward chaining*. The general principle of backward chaining is described in [114] and in [34].

In Section V of this paper we will present a top-down method for evaluating Datalog programs against an EDB. This method, called the *Query-Subquery approach* (*QSQ*), implicitly constructs all proof trees for a given goal in a recursive fashion.

Also the well-known programming language Prolog [45] is based on the principle of backward chaining. However, as we outline in the next subsection, the semantics of Prolog differs from that of Datalog.

H. Datalog and Prolog

From the syntactical point of view, Datalog is a subset of Prolog; hence each set of Datalog clauses could be parsed and executed by a Prolog interpreter. However, Datalog and Prolog differ in their semantics.

While Datalog, as a simplified version of general Logic Programming [78], has a purely declarative semantics, the meaning of Prolog programs is defined by an operational semantics, i.e., by the specification of how Prolog programs must be executed. A Prolog program is processed according to a resolution strategy which uses a depth-first search method with backtracking for constructing proof trees and respects the order of the clauses and literals as they appear in the program [45]. This strategy does not guarantee termination. The termination of a recursive Prolog program depends strongly on the order of the rules in the program, and on the order of the literals in the rules.

Consider, for example, the program P_1' consisting of the following clauses:

r_1': $sgc(X, Y) :- sgc(X1, Y1), par(X, X1), par(Y, Y1).$

r_2': $sgc(X, X) :- person(X).$

Note that this program differs from program P_1 of Section II-B only by the order of the rules, and of literals in the rule bodies. From a Datalog viewpoint, the order of clauses and literals is totally irrelevant, hence P_1', as a Datalog program, is equivalent to P_1. On the other hand, if we submit P_1' and the EDB E_1 to a Prolog interpreter and activate it, say, with the goal "?-$sgc(ann, X)$", then we would run into infinite recursion without getting any result.

Prolog has several system predicates, such as the *cut*, which render the language even more procedural. It is, however, a very rich and flexible programming language which has gained enormous popularity over the last years.

It is possible to couple Prolog to an external database. A Prolog interpreter can then distinguish between IDB and EDB-predicates. When an EDB goal is encountered during the execution of a Prolog program, the interpreter tries to retrieve a matching tuple from mass memory. Due to the procedural semantics of Prolog, which prescribes a particular order of visiting goals and subgoals, the required interaction between the interpreter and the external database is of the type *one-tuple-at-a-time*. This method of accessing mass storage data is quite inefficient compared to the *set-oriented* methods used by high-level query languages. Several enhanced coupling mechanisms have been proposed [64], [33], but no one takes full advantage of set-oriented techniques. This is probably not possible without compromising Prolog's semantics.

It is the aim of the next section to show that Datalog is well suited for set-oriented techniques.

III. Datalog Is Really a Database Language

Although expressing queries and views in Datalog is quite intuitive and fascinating from the user's viewpoint, we should not forget that the aim of database query languages like Datalog is providing access to large quantities of data stored in mass memory. Thus, in order to enable an easy integration between Datalog and database management systems, we need to relate the logic programming formalism to the most common database languages. We have chosen relational algebra as such a data retrieval language. This section provides an informal description of the translation of Datalog programs and goals into relational algebra.

A. Translation of Datalog Queries into Relational Algebra

Each clause of a Datalog program is translated, by a syntax-directed translation algorithm, into an inclusion relationship of relational algebra. The set of inclusion relationships that refer to the same predicate is then interpreted as an *equation* of relational algebra. Thus, we say that a Datalog program gives rise to a system of *algebraic equations*. Each IDB-predicate of the Datalog program corresponds to a *variable relation*; each EDB-predicate of the Datalog program corresponds to a *constant relation*. Determining a *solution of the system* corresponds to determining the value of the variable relations which satisfy the system of equations [61], [63]. The translation from Datalog to relational algebra is described in [28], [34], and [122].

Let us consider a Datalog clause $C: p(\alpha_1, \ldots, \alpha_n) :- q_1(\beta_1, \ldots, \beta_k), \ldots q_m(\beta_s, \ldots, \beta_h)$. The translation associates to C an inclusion relationship $Expr(Q_1, \ldots, Q_m) \subseteq P$, among the relations $P, Q_1, \ldots, Q_m$ that correspond to predicates $p, q_1, \ldots, q_m$,[5] with the convention that relation attributes are named by the number of the corresponding argument in the related predicate. For example, the Datalog rules of program P_1 from Section II:

r_1: $sgc(X, X) :- person(X).$

r_2: $sgc(X, Y) :- par(X, X1), sgc(X1, Y1), par(Y, Y1)$

[5]Note that some of the q_i might be p itself, yielding a recursive rule.

are translated into the inclusion relationships:

$$\Pi_{1,5}\left(\left(PAR \underset{2=1}{\bowtie} SGC\right) \underset{4=2}{\bowtie} PAR\right) \subseteq SGC$$

$$\Pi_{1,1}\ PERSON \subseteq SGC$$

The rationale of the translation is that literals with common variables give rise to *joins*, while the head literal determines the *projection*. Details of translation rules can be found in [28]. Note that, in order to obtain a two-column relation *SGC* in the second inclusion relationship, we have performed a double projection of the unique column of relation *PERSON*.

For each IDB predicate p, we now collect all the inclusion relationships of the type $Expr_i(Q_1, \ldots, Q_m) \subseteq P$, and generate an *algebraic equation* having P as LHS, and the *union* of all the left-hand sides of the inclusion relationships as RHS:

$$P = Expr_1(Q_1, \ldots, Q_m) \cup Expr_2(Q_1, \ldots, Q_m) \ldots \cup Expr_{m_p}(Q_1, \ldots, Q_m).$$

For instance, from the above inclusion relationships we obtain the following equation:

$$\Pi_{1,5}\left(\left(PAR \underset{2=1}{\bowtie} SGC\right) \underset{4=2}{\bowtie} PAR\right) \cup \Pi_{1,1}\ PERSON = SGC$$

Note that the transformation of several disequations into one equation really captures the minimality requirement contained in the least Herbrand model semantics of a Datalog program. In fact, it expresses the fact that we are only interested in those ground facts that are consequences of our program.

We also translate *logic goals* into *algebraic queries*. Input Datalog goals are translated into projections and selections over one variable relation of the system of algebraic equations. For example, the logic goal "?-$p(X)$." is equivalent to the algebraic query "P", and "?-$q(a, X)$". is equivalent to "$\sigma_{1=a}Q$".

B. The Expressive Power of Datalog

The system produced by the above translation includes all the classical relational operations, with the exception of difference; we say that it is written in *positive relational algebra*, RA^+. It can be easily shown that each defining expression of RA^+ can also be translated into a Datalog program [34]. This means that Datalog is at least as expressive as RA^+; in fact, Datalog is strictly more expressive than RA^+ because in Datalog it is possible to express recursive queries, which are not expressible in RA^+.

However, there are expressions in full relational algebra that cannot be expressed by Datalog programs. These are the queries that make use of the *difference* operator. For example, given two binary relations R and S, there is no Datalog rule defining $R - S$. Fig. 2 graphically represents the situation. We will see in Section VI that these expressions can be captured by enriching pure Datalog with the use of logical negation ($\neg$).

Note also that, even though Datalog is syntactically a subset of first-order logic, strictly speaking they are not comparable. Indeed, the semantics of Datalog is based on the choice of a specific model (the least Herbrand model), while first-order logic does not *a priori* require a particular choice of the model. Interesting tractations of the problem of the expressive power of relational query languages can be found in [10] and [2].

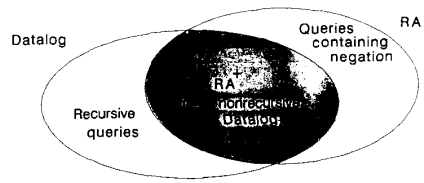


Fig. 2. The expressive power of Datalog.

IV. The Optimization Problem: An Overview

In this section we provide a classification of efficient methods for evaluating Datalog goals. A systematic overview of methods is required, since optimization can be achieved using a variety of techniques, and understanding their relationships is not obvious.

We classify optimization methods according to the *formalism*, to the *search strategy*, to the *objective of the optimization*, and to the *type of considered information*.

A. Logic and Algebraic Formalism

It comes out from Section III that programs in Datalog can equivalently be expressed as systems of equations of RA^+. So, we consider two alternative formalisms, that we regard, respectively, as *logic* and *algebraic*, and we discuss optimization methods that belong to both worlds.

We emphasize the convenience of mapping to the algebraic formalism, as this allows reusing classical results, such as conjunctive query optimization, common subexpression analysis, and the quantitative determination of costs associated to each operation of relational algebra [69], [84], [122].

B. Search Strategy

We recall from Section II that the evaluation of a Datalog goal can be performed in two different ways: *bottom-up*, starting from the existing facts and inferring new facts, or rather *top-down*, trying to verify the premises which are needed in order for the conclusion to hold.

In fact, these two evaluation strategies represent different interpretations of a rule. *Bottom-up* evaluations consider rules as *productions*, that apply the initial program to the EDB, and produce all the possible consequences of the program, until no new fact can be deduced. Bottom-up methods can naturally be applied in a set-oriented fashion, i.e., taking as input the entire relations of the EDB. This is a desirable feature in the Datalog context, where large quantities of data must be retrieved from mass memory. On the other hand, bottom-up methods do not take immediate advantage of the selectivity due to the existence of arguments bound to constants in the goal predicate.

In top-down evaluation, instead, rules are seen as *problem generators*. Each goal is considered as a problem that

must be solved. The initial goal is matched with the left-hand side of some rule, and *generates* other problems corresponding to the right-hand side predicates of that rule; this process is continued until no new problems are generated. In this case, if the goal contains some bound argument, then only facts that are somehow related to the goal constants are involved in the computation. Thus, this evaluation mode already performs a relevant optimization because the computation automatically disregards many of the facts which are not useful for producing the result. On the other hand, in top-down methods it is more natural to produce the answer *one-tuple-at-a-time*, and this is an undesirable feature in Datalog.

If we restrict our attention to the top-down approach, we can further distinguish two search methods: *breadth-first* or *depth-first*. With the depth-first approach, we face the disadvantage that the order of literals in rule bodies strongly affects the performance of methods. This happens in Prolog, where not only efficiency, but even *termination* of programs is affected by the left-to-right order of subgoals in the rule bodies. Instead, Datalog goals seem more naturally executed through breadth-first techniques, as the result of the computation is neither affected by the order of predicates within the right-hand sides (RHS) of rules, nor by the order of rules within the program.

C. Objectives of Optimization Methods

Our third classification criterion is based on the different *objectives* of optimization methods: some methods perform *program transformation*, namely, transforming a program into another program which is written in the same formalism, but yields a more efficient computation when one applies an evaluation method to it; we refer to these as *rewriting methods*. Given a goal G and a program P, the rewritten program P' is *equivalent* to P *with respect to* G, as it produces the same result. Formally, recalling the definition of $\mathfrak{M}_{P,G}$ of Section II, two programs P and P' are equivalent with respect to a goal G iff $\mathfrak{M}_{P,G} = \mathfrak{M}_{P',G}$.

These methods contrast with the *pure evaluation methods*, which propose effective evaluation strategies, where the optimization is performed during the evaluation itself.

D. Type of Considered Information

Optimization methods differ in the *type of information used in the optimization*.

Syntactic optimization is the most widely used; it deals with those transformations to a program which descend from the program's syntactic features. In particular, we distinguish two kinds of structural properties. One is the analysis of the *program structure*, and in particular the type of rules which constitute the program. For example, some methods exploit the *linearity* (see Section V) of the rules to produce optimized forms of evaluation. The second one is the *structure of the goal*, and in particular the selectivity that comes from goal constants. These two approaches are not mutually exclusive: it is possible to build syntactic methods which combine both cases.

Semantic optimization, instead, concerns the use of additional semantic knowledge about the database in order to produce an efficient answer to a query; the combination of the query with additional semantic information is performed automatically. Semantic methods are often based on integrity constraints, which express properties of valid databases. For instance, a constraint might state that *"all intercontinental flights directed to Milan land in Malpensa airport"*. This constraint can be used to produce the answer of a goal asking for *"the arrival airport of the intercontinental flight AZ747 from New York to Milan"* without accessing the EDB. Although we think that semantic optimization has the potential for significant improvements of query processing strategies, we do not further consider semantic optimization methods in this paper; various approaches to semantic optimization are proposed in the literature; among others; see [70], [36].

CRITERION	ALTERNATIVES
Formalism	logic vs. relational algebra
Search technique	bottom-up vs. top-down
Traversal order	depth-first vs. breadth-first
Objective	rewriting vs. pure evaluation
Approach	syntactic vs. semantic
Structure	rule structure vs. goal structure

(a)

	BOTTOM-UP	TOP-DOWN
EVALUATION METHODS	Naive (Jacobi, Gauss-Seidel) [28], [29] [61], [15] Semi-naive [15] Henschen-Naqvi [59]	Query-subquery [128], [130]
	LOGIC	ALGEBRAIC
REWRITING METHODS	Magic sets [14], [17] Counting [14] Magic Counting [106] Static filtering [68]	Variables reduction and Constant reduction [29]

(b)

Fig. 3. Classification of evaluation and optimization methods.

E. Classification of Evaluation and Optimization Methods

Fig. 3(a) summarizes classification criteria for Datalog optimization methods; they concern the *search strategy* (bottom-up or top-down), the *objective* (rewriting or pure evaluation), and the *formalism* (logic or algebraic). By combining approaches and excluding some alternatives that do not correspond to relevant classes of methods, we obtain four classes, each including rather homogeneous methods:

1) top-down evaluation methods
2) bottom-up evaluation methods
3) logic rewriting methods
4) algebraic rewriting methods.

Fig. 3(b) shows some of the most well-known evaluation and optimization methods present in the literature. This list is not exhaustive for obvious brevity reasons; we apologize to authors whose methods are omitted. In the next section we survey informally some of these methods.

Finally, we expect optimization methods to satisfy three important properties.

- Methods must be *sound*: they should not include in the result tuples which do not belong to it.
- Methods must be *complete*: they must produce all the tuples of the result.
- Methods must *terminate*: the computation should be performed in finite time.

Although we omit to present formal proofs, all the methods surveyed in the next section satisfy all these properties.

V. Survey of Execution Methods and Optimization Techniques

In this section we present methods for *optimizing* a Datalog program, i.e., for generating efficiently the actual set of tuples which satisfy a given goal for a given set of Datalog rules. In order to do this, we first present a bottom-up *naive* evaluation method (called the *Gauss–Seidel* method), and discuss briefly its sources of inefficiency. Then, we examine how to reduce these sources of inefficiency: we introduce the *Semi-Naive* approach which improves a bottom-up computation of linear rules, and we present a top-down efficient strategy, called *Query-Subquery*. Then, we present two of the most known rewriting methods, *Magic Sets* and *Counting*, which are used to optimize the behavior of bottom-up computations. Finally, we briefly overview other optimization methods.

A. Bottom-Up Evaluation

The Gauss–Seidel method is an algebraic version of the *naive evaluation* paradigm, which is present in many different forms in the literature [13], [85], [41], [86], [87]. The method is also well known in numerical analysis, where it is used for determining the iterative solution (*fixpoint*) of systems of equations. Assume the following system Σ of relational equations:

$$R_i = E_i(R_1, \ldots, R_n), \qquad (i = 1, \ldots, n).$$

The Gauss–Seidel method proceeds as follows: initially, the variable relations R_i are set equal to the empty set. Then, the computation $R_i := E_i(R_1, \ldots, R_n)$, $(i = 1, \ldots, n)$ is iterated until all the R_i do not change between two consecutive iterations (namely, until the R_i have reached a *fixpoint*). At the end of the computation, the value assumed by the variable relations R_i is the solution of the system Σ.

GAUSS–SEIDEL METHOD
INPUT: A system of algebraic equations Σ, and an Extensional Database EDB.
OUTPUT: The values of the variable relations $R_1, \ldots, R_n$.
METHOD:
FOR $i := 1$ TO n DO $R_i := \varnothing$;
REPEAT
  cond := *true*;
  For $i := 1$ TO n DO
    BEGIN
    $S := R_i$;
    $R_i := E_i(R_1, \ldots, R_n)$;
    IF $R_i \neq S$ THEN *cond* := *false*;
    END;
UNTIL *cond*;
FOR $i := 1$ TO n DO OUTPUT(R_i).
ENDMETHOD

Note that step "$R_i := E_i(R_1, \ldots, R_n)$" of the Gauss–Seidel algorithm has the same effect as the application of rule EPP of Section II. However, instead of acting on single tuples, here we apply algebraic operations *simultaneously to entire relations*.

Variants of the Gauss–Seidel method are the *Jacobi* method and the *Chaotic* method [28]. The latter is obtained by computing the various algebraic expressions not in a strict sequential order. Different versions of the chaotic method yield the so-called *lazy* and *data flow* evaluations, that correspond to starting the evaluation of computable relations, respectively, at the latest or at the earliest convenience.

The Gauss–Seidel method has two sources of inefficiency.

a) Several tuples are computed multiple times during the iteration process. In particular, during the iterative evaluation of a relation R, tuples belonging to relations $R^{(i)}$ will also belong to all subsequent relations $R^{(j)}$, $j \geq i$, until the fixpoint is reached.

b) The method produces the *entire* result relations. If the goal contains constant arguments, they are selected only at the end. In this way, several tuples are computed without being really required, and eliminated by the final selection.

Inefficiencies due to observation a) above are partially eliminated through the *semi-naive* methods discussed in the remainder of this subsection; inefficiencies due to observation b) above are dealt with by evaluation methods (like *query–subquery*) and rewriting methods (like *magic sets*, *counting*, etc.), discussed in the next subsections.

Semi-naive evaluation is a bottom-up technique designed for eliminating redundancy in the evaluation of tuples at different iterations. Several versions of this method can also be found in the literature [13], [28], [20].

Consider the Gauss–Seidel algorithm, applied to solve a single equation on variable R. Let $R^{(k)}$ be the temporary value of relation R at iteration step k. The *differential* of R at step k of the iteration is defined as

$$D^{(k)} = R^{(k)} - R^{(k-1)}.$$

The differential term, expressing the new tuples of $R^{(k)}$ at each iteration k, is exactly what we would like to use at each iteration, instead of the entire relation $R^{(k)}$; this is legal with *linear equations*. An equation of relational algebra

$$R = E(R)$$

is *linear with respect to* R if it verifies the following property:

$$E(R' \cup R'') = E(R') \cup E(R'')$$

for any two relations R' and R'' having the same arity as R. Note that linearity is ensured if only one occurrence of R appears in the expression $E(R)$. The differential of R at step k of the iteration is simply $E(D(R))$:

$$E(D^{(k)}) = E(R^{(k)} - R^{(k-1)}) = E(R^{(k)}) - E(R^{(k-1)})$$

so that one can compute $R^{(k+1)} = E(R^{(k)})$ as the union

$$E(R^{(k-1)}) \cup E(D^{(k)}).$$

The advantage of this formula is that, at each iteration k, we need to compute $E(D^{(k)})$ rather than $E(R^{(k)})$. This can be generalized to systems of equations (see [29], [63]).

Extensions of this method, the *general semi-naive*, and the *pseudo-rewriting semi-naive* (discussed, among others, in [34], [135], and [63]) enable a less efficient application of a similar approach to nonlinear equations.

B. Top-Down Evaluation

The *query–subquery* (*QSQ*) algorithm [129] is an efficient top-down evaluation algorithm, optimizing the behavior of backward-chaining methods as described in Section II.

The objective of the QSQ method is to access the minimum number of facts needed in order to determine the answer. In order to do this, the fundamental notion of *subquery* is introduced. A goal, together with a program, determines a *query*. Literals in the body of any one of the rules defining the goal predicate are subgoals of the given goal. Thus, a subgoal, together with the program, yields a *subquery*; this definition applies recursively to subgoals of rules which are subsequently activated. In order to answer the query, each goal is expanded in a list of subgoals, which are recursively expanded in their turn.

The method maintains two sets: a set P of *answer tuples*, containing answers to the main goal and answers to intermediate subqueries, which is represented by a set of *temporary relations* (one relation for each IDB-predicate); and a set Q of *current subqueries* (or *subquery instances*), which contains all the subgoals that are currently under consideration.

Thus, the function of the QSQ algorithm is twofold: *generating new answers* and *generating new subqueries that must be answered*. There are two versions of the QSQ algorithm, an iterative one (QSQI) and a recursive one (QSQR). The difference between the two concerns which of these two functions has priority over the other: QSQI privileges the production of answers, thus, when a new subquery is encountered, it is suspended until the end of the production of all the possible answers that do not require using the new subquery. QSQR behaves in the opposite way: whenever a new subquery is found, it is recursively expanded and the answering to the current subquery is postponed to when the new subquery has been completely solved.

At the end of the computation, P includes the answer to the goal; hence, as in the Gauss–Seidel method, it is required to perform (una tantum) the final selection. However, the reader should notice that the QSQ method uses the information about constants in the goal, hence, the size of P and of all the relations involved in the computations is comparatively much smaller than the size of all the relations involved in the Gauss–Seidel computation.

This algorithm can be compared to the Prolog inferential machine, which is also top-down. The comparison is purely indicative, based on the idea of using a Prolog inferential machine to execute a Datalog program. We note that Prolog acts one-tuple-at-a-time, while QSQ is *set-oriented*, as it processes whole relations. In this sense, QSQ is appropriate for database processing. Also, QSQ is breadth-first and always terminates, while Prolog is depth-first and may instead not terminate in some cases.

The query–subquery algorithm was introduced by Vieille in [128]. The version introduced in that paper was found to be incomplete. The author [129], [130] and others [93], [100] have subsequently provided corrections or complete versions of the algorithm. A detailed description of the method can be found in [34].

C. Magic Sets

The method of the magic sets is a logical rewriting method which transforms a program into a larger one, containing some more rules that define new IDB predicates. These IDB predicates serve as constraints, which force the program variables to satisfy some additional conditions. Thus, during bottom-up computation, the variables of the modified rules may assume only *some* of the values that were instead allowed for variables of the original rules. In most cases, this makes the new program more efficient.

Details about this algorithm can be found in [14], [17], [22], [34]. We show here a significant example, and make some comments on the rewritten program. Consider again the program P_1 with the goal $?\text{-}sgc(ann, X)$. After the magic sets transformation, the program becomes:

r_1': $sgc(X, X) :- person(X).$

r_2': $sgc(X, Y) :- magic(X1), par(X, X1),$
$\quad sgc(X1, Y1), par(Y, Y1).$

r_3': $magic(ann).$

r_4': $magic(X1) :- magic(X), par(X, X1).$

Let us first notice what has happened to the initial program: two new rules have been added, and one rule has been modified. Let us consider this rule (r_2'). A new literal has been added to its body: $magic(X1)$. The presence of this new literal forces the argument $X1$ of the EDB relation *PAR* to assume only some specific values, i.e., values which also belong to the IDB relation *MAGIC*.

Let us now see how this new relation is defined. The first rule (r_3') simply says that the constant *ann* belongs to it. Rule r_4' says that, if $X1$ is in *MAGIC*, and X is a parent of $X1$, then also X is in *MAGIC*. The result of the

computation of relation *MAGIC* is

$$MAGIC = \{ < ann >, < dorothy >, < hilary >, < george > \}.$$

Thus, the IDB relation *MAGIC* contains *all the ancestors of ann*. The tuples of the relation *MAGIC* defined by the magic rules form the *magic set*. It is often called the **cone** of *ann*, referring to the fact that, starting from *ann*, the ancestors grow "fanning out" like a cone. By imposing that, in r_2', $X1$ must belong to relation *MAGIC*, we are imposing that, in the computation of Ann's cousins at the same generation, we only have to consider those other pairs of same generation cousins whose first element is an ancestor of Ann.

This example can be used to introduce the idea of *sideways information passing* (*SIP*) [22]. Intuitively, given a certain rule and a literal in the rule body with some bound argument(s), one can use the knowledge about the relation corresponding to this literal to obtain bindings for uninstantiated variables in other argument positions. This process can be iterated for each literal in the rule body, and recursively on other IDB-predicates. Thus, known information (bindings to constants) is *passed sideways* within the rule body. As we have seen in Section V-B, this is the normal behavior of top-down evaluation methods, for instance QSQ.

After the magic sets transformation, the resulting program can be evaluated by a simple algorithm like Gauss–Seidel or semi-naive, still taking advantage of the binding passing strategy. However, the application of a bottom-up computation method to the rewritten program may produce more tuples than exactly those of the goal answer, as it includes all the same generation pairs of all people belonging to the cone of *ann*. Thus, as in QSQ, the resulting relation must be finally selected to obtain the answer.

The magic sets method has been extended by Sacca' and Zaniolo to a class of queries to logic programs that contain function symbols [105]. A more sophisticated technique called "Supplementary Magic Sets", has been introduced by Beeri and others [22], whose virtue is to eliminate some repeated computations. Another improvement of the magic set method is proposed in [99], where top-down evaluation is completely mimicked by bottom-up, using a semi-naive evaluation of rewritten rules; the rule rewriting uses initial goal bindings in a very sophisticated way; for instance, goals like $p(X, X)$, give rise to the binding of the two occurrences of variables X to each other. This was not covered by the original magic sets method.

D. Counting

The Counting method is a rewriting method based on the knowledge of the goal bindings; the method includes the computation of the magic set, but each element of the magic set is complemented by additional information expressing its "distance" from the goal constant. The counting method was also first presented in [14]; an improved version was introduced in [22].

Consider again the program P_1 used as an example of the magic set method in the previous subsection. The magic set method restricts the computation to the ancestors of *ann*; for each of these elements, the counting method maintains the information whether it is one of *ann*'s parents (distance 1), *ann*'s grandparents (distance 2), *ann*'s grand-grandparents (distance 3), etc. The rewritten program contains the computation of these distances. At this point, computation may be restricted, respectively, to the children of *ann*'s parents, to the grandchildren of *ann*'s grandparents, to the grand-grandchildren of *ann*'s grand-grandparents, etc.

The following is the result of applying the counting transformation to the output of the magic sets method:

$$sgc'(X, X, I) :- person(X), integer(I).$$
$$sgc'(X, Y, I) :- counting(X1, I), par(X, X1), sgc'(X1, Y1, J), par(Y, Y1), I = J - 1.$$
$$counting(ann, 0).$$
$$counting(X1, I) :- counting(X, J), par(X, X1), I = J + 1.$$

With the goal:

$$? - sgc'(ann, Y, 0).$$

With some liberality, we use built-in predicates for addition and subtraction, that will be discussed in Section VI-A. The reader can observe that the counting predicate increments generation levels from *ann* upwards, marking the level of each element of the magic set, whereas the *sgc'* predicate decrements generation levels. The goal selects only the *sgc* pairs of level 0, i.e., those at the same level as *ann*. At each step, among the tuples of the magic set, the program only uses the tuples that have the appropriate distance from *ann*.

The application of the counting method has the potential for improving the efficiency of the computation, but clearly the method does not terminate when the database is cyclic (as the increment of the counting variable is not arrested). Thus, the counting method applies to acyclic databases only. A further hypothesis is that the program be linear, with at most one recursive rule for each predicate.

In order to improve the applicability of the counting method whenever it is not known *a priori* whether the database is cyclic, Sacca' and Zaniolo have introduced [106] the "magic counting method", which constantly monitors the counting computation in order to determine whether the underlying database is cyclic; if so, it switches to the magic set computation.

E. The Method of Henschen and Naqvi

One of the earliest pure evaluation methods proposed in the literature was developed by Henschen and Naqvi

[59]; the method applies to linear Datalog programs with goals that contain bound arguments.

This method produces an iterative program that evaluates the goal through several steps. Each step produces some of the answer tuples, and, at the same time, computes symbolically a new expression that has to be evaluated at the following step. The method is based on a "functional" interpretation of predicates: we can view any predicate p having at least two arguments as a set function from some of its arguments to the remaining ones, associating to each set S of values of its first arguments a set S' of values of its second arguments; for instance, if p is a binary predicate, we denote by f_p the functional mapping

$$S' = f_p(S) = \{ y \mid x \in S \quad \text{and} \quad p(x, y) \}.$$

With this notation, one can also perform the composition of predicate functions, in the usual mathematical sense. This can be applied to the solution of linear Datalog programs, by considering the bound arguments as the function's domains. We exemplify the method of Henschen–Naqvi on program P_1, with the goal: ?-$sgc(a, X)$.

$$f_{person}(a) \cup f_{rap}(f_{person}(f_{par}(a))) \cup f_{rap}\left(f_{rap}\left(f_{person}(f_{par}(f_{par}(a)))\right)\right) \cup \ldots$$

where f_{person} denotes a unary function returning all persons, f_{par} is the set function from the first to the second argument of the relation PAR, and f_{rap} is the set function from the second to the first argument of the relation PAR.

The most interesting feature of the method of Henschen and Naqvi is that it integrates two kinds of computation: at a certain step, some tuples of the answer are computed, but also some symbolic manipulation is performed. The first kind of computation is typical of pure evaluation methods, the second is characteristic of rewriting methods.

Substantially the same functional interpretation of predicates is given in a more general form by Gardarin and De Maidreville in [52], who propose a method for evaluating queries as function series, where the functions involved correspond to the functional interpretation of the predicates. Follow-up work can be found in [54].

F. Other Efficient Evaluation Methods

Several other methods can be found in the literature; most of them share some characteristics with the methods presented above.

Static Filtering is a rewriting method introduced by Kifer and Lozinskii in [68]. In this method, a bottom-up evaluation is viewed as a *flow* of tuples through a graph derived from the program, called *relation-axiom graph*, with two types of nodes: *relation nodes*, associated to predicates, and *axiom nodes*, associated to rules. Computation is ideally preformed inside axiom nodes.

When the goal has some bound arguments, those tuples produced during the graph traversal which do not satisfy the bindings can be eliminated at the end of the computation; the idea of the method is that of "cutting" off useless tuples from the computation at an earlier stage of their flow towards the goal node. This is achieved by imposing conditions on predicate arguments, called *filters*, to the output edges of each relation node; the expressions of filters are derived from the goal bindings, and propagated along the graph by a *push* operation.

A similar approach was presented by Devanbu and Agrawal in [47], but restricting the application to linear rules with only one occurrence of the recursive predicate in the RHS. Another interesting (pure evaluation) algorithm (the Apex method) was previously introduced by Lozinskii in [81]. The first method for pushing selections into recursive expressions was proposed by Aho and Ullman in a seminal paper [10]; the static filtering technique can be considered as a generalization of that concept.

Another generalization of [10] is presented by Ceri and Tanca in [29], by introducing the pair of methods *Variable Reduction* and *Constant Reduction* which apply to generic systems of algebraic equations. Both methods push the initial selections so as to use their "filtering" ability as soon as possible; the two methods act either by rewriting equations into equivalent ones with different (smaller) variables or by reducing the size of involved relations prior to the equations' evaluation. These methods are part of a structured approach to the optimization of systems of algebraic equations derived from logic programs, presented in [29].

An idea similar to magic sets is exhibited in the *Alexander Method*, published in [101]; the method consists of rewriting the rules for each recursive predicate so that they represent, respectively, problems and solutions for the predicate; all new rules obtained are evaluated bottom-up, but the evaluation of the problem rules simulates in fact top-down evaluation and allows binding passing among subgoals.

There are also other possible approaches to optimization. One of them is the method for redundancy elimination of Sagiv: in [107] an algorithm is presented for minimizing the *size* (in terms of number of rules in the program and of number of atoms in a rule) of a Datalog program under a decidable condition called *uniform equivalence*. Another kind of optimization is achieved by Ramakrishnan and others, in [98], where the objective of optimization is *pushing projections*, rather than selections, in the body of a Datalog rule. This means deleting some argument positions in the literals of the rule body, which also has, sometimes, the effect of making some rules redundant for the computation.

A comparative evaluation of the performance of methods presented prior to 1986 is presented by Bancilhon and Ramakrishnan [15]. The study uses a benchmark which includes only few, conventional programs; it excludes cyclic databases, which are quite common. However, this paper remains one of the few quantitative approaches to the comparison of optimization methods; we think that this topic deserves a much more thorough treatment, al-

though achieving a comparative evaluation of methods' performance is extremely difficult due to the variety and inherent complexity of methods and to the continuous evolution of the field.

G. Computing Transitive Closures Efficiently

Computing transitive closures efficiently has been recognized as a significant subproblem of the efficient computation of general recursive queries [62], [7]–[9], [123]. In fact, many proposals of other extensions to the relational query languages in "nontraditional" directions include this operation [8], [102], [58], [136]. Thus, an independent area of research has been developed which studies the efficient implementation of transitive closures improving the naive and semi-naive methods. The efficient methods are mainly concerned with transitive closures of binary relations representing graphs and trees, very often making use of sparse matrices to represent them.

In [62], a logarithmic technique is applied in order to build more efficient iterations. The paper provides an interesting study of performance on tree-shaped databases, showing that logarithmic techniques perform much better than the traditional ones (as it could be expected). In [7], *direct* algorithms are proposed, which are based on studies performed in different contexts [108], [109], [131], [132]. The name "direct" algorithm descends from the fact that the length of the computation does not depend on the length of paths in the underlying graph. Several research efforts have also been directed towards the parallel computation of transitive closures. In these proposals, the computation of transitive closures is performed on several processors at the same time; the significant problem is avoiding maintaining too many duplicates, and to perform too many duplicate computations on different processors. Two such algorithms were proposed in [124]. Other significant papers are [9] and [67].

VI. Extensions of Pure Datalog

The Datalog syntax we have been considering so far corresponds to a very restricted subset of first-order logic and is often referred to as *pure Datalog*. Several extensions of pure Datalog have been proposed in the literature or are currently under investigation. The most important of these extensions are *built-in predicates, negation*, and *complex objects*.

A. Built-In Predicates

Built-in predicates (or "built-ins") are expressed by special predicate symbols such as $>$, $<$, $\geq$, $\leq$, $=$, $\neq$ with a predefined meaning. These symbols can occur in the right-hand side of a Datalog rule; they are usually written in infix notation.

Consider, for example, the following program P_2 consisting of a single rule where *par* is an EDB predicate:

$$P_2\text{: } sibling(X, Y) :- par(Z, X), par(Z, Y), X \neq Y.$$

The meaning of this program is obvious. By use of the inequality built-in predicate we avoid that a person is considered as his own sibling.

From a formal point of view, built-ins can be considered as EDB-predicates with a different physical realization than ordinary EDB-predicates: they are not explicitly stored in the EDB but are implemented as procedures which are evaluated during the execution of a Datalog program. However, built-ins correspond in most cases to *infinite* relations, and this may endanger the *safety* of Datalog programs.

Safety means that a Datalog program should always have a finite output, i.e., the intensional relations defined by a Datalog program must be finite. It is easy to see that safety can be guaranteed by requiring that each variable occurring as argument of a built-in predicate in a rule body must also occur in an ordinary predicate of the same rule body, or must be bound by an equality (or a sequence of equalities) to a variable of such an ordinary predicate or to a constant. Here, by "ordinary predicate", we mean a nonbuilt-in predicate.

During the evaluation of a Datalog rule with built-in predicates, the following principle has to be observed: defer the evaluation of a built-in predicate until all arguments of this predicate are bound to constants. An exception to this principle can (sometimes should) be made for the equality predicate. An equality should be evaluated as soon as *one* of its two arguments is a constant or is bound to a constant.

In a similar way, *arithmetical built-in predicates* can be used. For instance, a predicate $plus(X, Y, Z)$ may be used for expressing $X + Y = Z$, where the variables X, Y, and Z are supposed to range over a numeric domain. During the evaluation of a rule body, such a predicate can be evaluated as soon as bindings for its "input variables" (here X and Y) are provided.

Finally, let us remark that when Datalog rules are transformed into algebraic equations, then many built-in predicates can be expressed through join conditions. The above defined program P_2, for instance, is translated into the following equation of relational algebra:

$$SIB = \Pi_{2,4}(PAR \bowtie_{1=1 \wedge 2 \neq 2} PAR)$$

where *SIB* and *PAR* denote the relations corresponding to the predicates *sibling* and *parent*, respectively.

B. Incorporating Negation into Datalog: The Problem

In pure Datalog, the negation sign "$\neg$" is not allowed to appear. However, by adopting the Closed World Assumption (CWA), we may infer negative facts from a set of pure Datalog clauses.

Note that the CWA is not a universally valid logical rule, but just a principle that one may or may not adopt, depending on the semantics given to a language. In the context of Datalog, the CWA can be formulated as follows:

CWA: If a fact does not logically follow from a set of

Datalog clauses, then we conclude that the negation of this fact is true.

Negative Datalog facts are positive ground literals preceded by the negation sign, for instance, $\neg$ *sgc*(*bertrand*, *hilary*). Note that this negative fact follows by the CWA from $P_1 \cup E_1$, since *sgc*(*bertrand*, *hilary*) does not follow from $P_1 \cup E_1$. If F denotes a negative ground fact, then $|F|$ denotes its positive counterpart. For example: $|\neg sgc(bertrand, hilary)| = sgc(bertrand, hilary)$.

The CWA applied to pure Datalog allows us to deduce negative facts from a set S of Datalog clauses. It does not, however, allow us to use these negative facts in order to deduce some further facts. In real life, it is often necessary to express rules whose premises contain negative information, for instance: "*if X* is a student *and X* is *not* a graduate student, *then X* is an undergraduate student". In pure Datalog, there is no way to represent such a rule.

Note that in relational algebra an expression corresponding to the above rule can be formulated with ease by use of the set-difference operator "$-$". Assume that a one-column relation *STUD* contains the names of all students and another one-column relation *GRAD* contains the names of all graduate students. Then we obtain the relation *UND* of all undergraduate students by simply subtracting *GRAD* from *STUD*, thus

$$UND = STUD - GRAD.$$

Our intention is now to extend pure Datalog by allowing negated literals in rule bodies. Assume that the unary predicate symbols *stud*, *und*, and *grad* represent the properties of being a student, an undergraduate, and a graduate, respectively. Our rule could then be formulated as follows:

$$und(X) :- stud(X), \neg grad(X).$$

More formally, let us define Datalog$^\neg$ as the language whose syntax is that of Datalog except that negated literals are allowed in rule bodies. Accordingly, a Datalog$^\neg$ clause is either a positive (ground) fact or a rule where negative literals are allowed to appear in the body. For safety reasons we also require that each variable occurring in a negative literal of a rule body also occurs in a positive literal of the same rule body.

In order to discuss the semantics of Datalog$^\neg$ programs, we first generalize the notion of the *Herbrand Model* (see Section II-D) to cover negation in rule bodies.

Let $\mathfrak{I}$ be a Herbrand interpretation, i.e., a subset of the Herbrand base *HB*. Let F denote a positive or negative Datalog fact.

$$F \text{ is satisfied in } \mathfrak{I} \text{ iff } \begin{cases} F \text{ is a positive fact and } F \in \mathfrak{I}, \text{ or} \\ F \text{ is a negative fact and } |F| \notin \mathfrak{I}. \end{cases}$$

Now, let R be a Datalog$^\neg$ rule of the form $L_0 :- L_1, \ldots, L_n$ and let $\mathfrak{I}$ be a Herbrand interpretation. R is satisfied in $\mathfrak{I}$ iff for each ground substitution θ for R, whenever it holds that for all $1 \leq i \leq n$, $L_i\theta$ is satisfied in $\mathfrak{I}$, then it also holds that $L_0\theta$ is satisfied in $\mathfrak{I}$. (Note that $L_0\theta$ is satisfied in $\mathfrak{I}$ iff $L_0\theta \in \mathfrak{I}$, since $L_0\theta$ is positive.)

Let S be a set of Datalog$^\neg$ clauses. A Herbrand interpretation $\mathfrak{I}$ is a Herbrand model of S iff all facts and rules of S are satisfied in $\mathfrak{I}$.

In analogy to pure Datalog, we require that the set of all positive facts derivable from a set S of Datalog$^\neg$ clauses be a minimal model of S. However, a set S of Datalog$^\neg$ clauses may have more than one minimal Herbrand model. For instance, if $S_c = \{boring(chess) :- \neg interesting(chess)\}$, then S has two minimal Herbrand models: $H_a = \{interesting(chess)\}$ and $H_b = \{boring(chess)\}$.

The existence of several minimal Herbrand models for a set of Datalog$^\neg$ clauses entails difficulties in defining the semantics of Datalog$^\neg$ programs: which of the different minimal Herbrand models should be chosen? Note also that the model minimality requirement is inconsistent with the CWA. By the CWA both facts $\neg$ *boring*(*chess*) and $\neg$ *interesting*(*chess*) can be deduced from the above set S_c. Thus, neither of the models H_a and H_b are consistent with the CWA.

In the following we describe a policy which is commonly referred to as *stratified evaluation of Datalog$^\neg$ programs*, or simply as *stratified Datalog$^\neg$*. This policy permits us to select a distinguished minimal Herbrand model in a very natural and intuitive way by approximating the CWA. However, as we will see later, this method does not apply to all Datalog$^\neg$ programs, but only to particular subclass; the so-called *stratified* programs [37], [12]. Note also that this technique can be used in the more general context of Logic Programming with negation [12] as well.

C. Stratified Datalog$^\neg$

This policy of choosing a particular Herbrand model, and thus of determining the semantics of a Datalog$^\neg$ program is guided by the following intuition: when evaluating a rule with one or more negative literals in the body, first evaluate the predicates corresponding to these negative literals. Then the CWA is "locally" applied to these predicates.

For instance, the clause set S_c, as defined above, would be evaluated as follows: before trying to evaluate the predicate *boring*, we evaluate the predicate *interesting* which occurs negatively in the rule body. Since there are no rules and facts in S_c allowing us to deduce any fact of the form *interesting*(α), the set of positive answers to this predicate is empty. In particular, *interesting*(*chess*) cannot be derived. Hence, by applying the CWA "locally" to the *interesting* predicate, we derive $\neg$ *interesting*(*chess*). Now we evaluate the unique rule of S_c and get *boring*(*chess*). Thus, the computed Herbrand model is $H_b = \{boring(chess)\}$.

When several rules occur in a Datalog$^\neg$ program, then the evaluation of a rule body may engender the evaluation of subsequent rules. These rules may contain in turn neg-

ative literals in their bodies and so on. Thus, it is required that before evaluating a predicate in a rule head, it is always possible to completely evaluate all the predicates which occur negatively in the rule body or in the bodies of some subsequent rules and all those predicates which are needed in order to evaluate these negative predicates.

If a program fulfills this condition it is called *stratified*. Any stratified program P can be partitioned into disjoint sets of clauses $P = P^1 \cup \ldots \cup P^i \cup \ldots \cup P^n$ called *strata*, such that each IDB-predicate of P has its defining clauses within one stratum and P^1 contains only clauses with either no negative literals or with negative literals corresponding to EDB-predicates and each stratum P^i contains only clauses whose negative literals correspond to predicates defined in lower strata. The partition of P into $P^1 \ldots P^n$ is called a *stratification* of P.

Assume a stratified program P with given stratification $P^1 \ldots P^n$ has to be evaluated against an EDB E. The evaluation is done stratum-by-stratum as follows. First, P^1 is evaluated by applying the CWA locally to the EDB, i.e., by assuming $\neg p(c_1, \ldots, c_k)$ for each k-ary EDB-predicate p and constants $c_1, \ldots, c_k$ where $p(c_1, \ldots, c_k) \notin E$. Then the other strata are evaluated in ascending order. During the evaluation of each stratum P^i, the result of the previous computations is used and the CWA is made "locally" for all EDB-predicates and for all predicates defined by lower strata.

Consider, for example, the following program P_s, where d is the only EDB-predicate:

$$r_1\colon\ p(X, Y) :- \neg q(X, Y), s(X, Y).$$

$$r_2\colon\ q(X, Y) :- q(X, Z), q(Z, Y).$$

$$r_3\colon\ q(X, Y) :- d(X, Y), \neg r(X, Y).$$

$$r_4\colon\ r(X, Y) :- d(Y, X).$$

$$r_5\colon\ s(X, Y) :- q(X, Z), q(Y, T), X \neq Y$$

A stratification of P_s is: $P_s^1 = \{r_4\}$, $P_s^2 = \{r_2, r_3, r_5\}$, $P_s^3 = \{r_1\}$. Assume P_s is evaluated over the EDB $E_s = \{d(a, b), d(b, c), d(e, e)\}$. The evaluation of the first stratum produces the new facts: $r(b, a)$, $r(c, b)$, $r(e, e)$. The computation of the second stratum yields the following additional facts: $q(a, b)$, $q(b, c)$, $q(a, c)$, $s(a, b)$, $s(b, a)$. Finally, by evaluating the third stratum we get $p(b, a)$.

Note that a stratified program has, in general, several different stratifications. The program P_s, for example, has the following alternative stratification: $P_s^1 = \{r_4\}$, $P_s^2 = \{r_2, r_3\}$, $P_s^3 = \{r_1, r_5\}$. However, it can be shown [12] that all stratifications are equivalent, i.e., the result of the evaluation of a stratified Datalog$^\neg$ program P is independent of the stratification used.

It is easy to decide whether a given Datalog$^\neg$ program is stratified by analyzing the *Extended Dependency Graph EDG*(P) of P. The nodes of EDG(P) consist of the IDB-predicate symbols occurring in P. There is a (directed) edge $< p, q >$ in EDG(P) iff the predicate symbol q occurs positively or negatively in a body of a rule whose head predicate is p. Furthermore, the edge $< p, q >$ is marked with a "$\neg$" sign iff there exists at least one rule R with head predicate p such that q occurs negatively in the body of R. The extended dependency graph EDG(P_s) of the program P_s is depicted in Fig. 4.

A Datalog$^\neg$ program P is stratified iff EDG(P) does not contain any cycle involving an edge labeled with "$\neg$". If P is stratified, then it is quite easy to construct a particular stratification for P from EDG(P) [34]. Another method for constructing a stratification is given in [122].

It can be shown that the strata-by-strata evaluation of a stratified program P on the base of an underlying EDB E always produces a minimal Herbrand model of $P \cup$ EDB. This model is also called the *Perfect Model* and can be characterized in a purely nonprocedural way [12], [95]. *Local stratification*, a refinement of stratification, is proposed in [95].

D. Inflationary Evaluation and Expressive Power of Datalog$^\neg$ Programs

Another evaluation paradigm for Datalog$^\neg$ programs has been proposed recently in [4], [72]. This method, called *inflationary evaluation*, has the advantage of applying to all Datalog$^\neg$ programs and not just stratified Datalog$^\neg$ programs.

Let P be a Datalog$^\neg$ program and E an EDB. The inflationary evaluation of P on E is performed iteratively so that all rules of P are processed in parallel at each step. From the EDB and the facts already derived, new facts are derived by applying the rules of P. These new facts are added to the result at the end of each step. At each step, the CWA is made temporarily during the evaluation of the rule bodies: it is assumed that the negation of all facts not yet derived is valid. The procedure terminates when no more additional facts can be derived.

Consider, for example, the following nonstratified program P_i, where d is the only EDB-predicate:

$$r_1\colon\ s(X) :- p(X), q(X), \neg r(X).$$

$$r_2\colon\ p(X) :- d(X), \neg q(X).$$

$$r_3\colon\ q(X) :- d(X), \neg p(X).$$

$$r_4\colon\ r(X) :- d(X), d(b).$$

Assume that this program is evaluated inflationarily against an EDB $E_i = \{d(a)\}$. During the first iteration step, rule r_2 produces the new fact $p(a)$, and rule r_3 produces the new fact $q(a)$. During the second iteration step, rule r_1 produces the new fact $s(a)$. Since no further facts are derivable, the procedure stops with the result $\{p(a), q(a), s(a)\}$.

It is easy to see the inflationary evaluation of a Datalog$^\neg$ program P corresponds to the computation of a least fixpoint [72], [34]. Furthermore, the result united to E is a Herbrand model of $P \cup E$. However, in general, it is not a *minimal* Herbrand model. For instance, in the above example the computed Herbrand model is $\{d(a), p(a), q(a), s(a)\}$, but this model is not minimal since

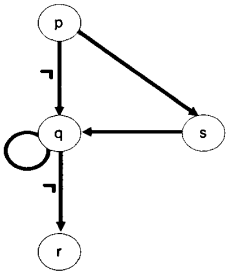


Fig. 4. Extended dependency graph EDG(P_S).

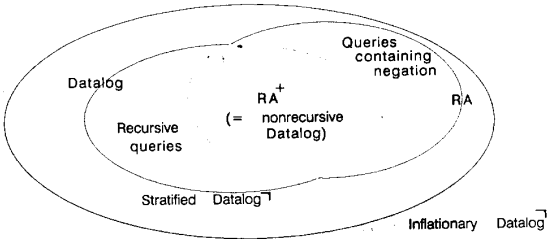


Fig. 5. Hierarchy of expressiveness of different versions of Datalog.

$\{d(a), q(a)\}$ and $\{d(a), p(a)\}$ are smaller Herbrand models of $P_i \cup E_i$.

Just as for pure Datalog programs, one can specify a goal together with a Datalog$^\neg$ program. In that case, again, the output consists of all those derived IDB-facts which are subsumed by the given goal. It can be shown [4], [71], [72], that for each stratified Datalog$^\neg$ program P with goal G there exists a Datalog$^\neg$ program P' such that the output of the stratified computation of P on any EDB E w.r.t. G is equal to the output of the inflationary computation of P' on E w.r.t. G. This means that inflationary Datalog$^\neg$ is at least as expressive as stratified Datalog$^\neg$. Moreover, there exist programs whose inflationary evaluation (w.r.t. a given goal) cannot be simulated by any strata-by-strata computation of a stratified Datalog$^\neg$ program. Thus, inflationary Datalog$^\neg$ is computationally strictly more expressive than stratified Datalog$^\neg$. Furthermore, it has been shown that inflationary Datalog$^\neg$ has the same expressive power as *Fixpoint Logic on Finite Structures*, a well-known formalism obtained by extending first-order logic with a least fixpoint operator for positive first-order formulas [6].

It is also easy to see that stratified Datalog$^\neg$ is in turn strictly more expressive than pure Datalog [4]. Fig. 5 shows the hierarchy of expressiveness of the different languages and formalisms presented in this paper. More information on this subject can be found in [38].

We finish our discussion on incorporating negation into Datalog by giving a brief survey on other relevant work on this topic.

A different approach for defining the meaning of logic programs with negation is based on a proposal of Clark [44]. He essentially considers the *completion* of a logic program by viewing the definitions of derived predicates as logical equivalences rather than logical implications. The semantics of a logic program with negation can then be defined as a minimal model of its completion. This approach was discussed by Sheperdson [110], [111] and Lloyd [78]. Fitting [49] and Kunen [74] refined this approach by using three-valued logic: in their setting, a fact can be true, false, or undefined. In the context of Datalog these approaches are not completely satisfactory.

A recently introduced and very promising approach, also based on three-valued logic, is the *well-founded semantics* by Van Gelder, Ross, and Schlipf [126]. Their method nicely extends the stratified approach to arbitrary logic programs with negation. In particular, every stratified program is semantically characterized by a *total* model, i.e., a model such that each fact of the Herbrand base has either truth value "*true*" or "*false*". This model coincides with the perfect model mentioned above. Nonstratified programs, on the other hand, can be characterized by *partial* models, where single facts may assume truth value "*undefined.*" A fixpoint method for computing the well-founded partial model is given in [127], while resolution-based procedural semantics for well-founded negation is provided in [103]. Further important papers related to well-founded semantics are [27], where the relationship to logical constructivism is investigated, and [96].

E. Complex Objects

The "objects" handled by pure Datalog programs correspond to the tuples of relations which in turn are made of attribute values. Each attribute value is atomic, i.e., not composed of sub-objects; thus the underlying data model consists of relations in first normal form. This model has the advantage of being both mathematically simple and easy to implement. On the other hand, several new application areas (such as computer-aided design, office automation, and knowledge representation) require the storage and manipulation of (deeply nested) structured objects of high complexity. Such complex objects cannot be represented as atomic entities in the normalized relational model but are broken into several autonomous objects. This implies a number of severe problems of conceptual and technical nature.

For this reason, the relational model has been extended in several ways to allow the compact representation of complex objects. Datalog can be extended accordingly. The main features that are added to Datalog in order to represent and manipulate complex objects are *function symbols* as a glue for composing objects from sub-objects and *set constructors* for being able to build objects which are collections of other objects. Function symbols are "uniterpreted", i.e., they do not have any predefined meaning. Usually one also adds a number of *predefined* functions for manipulating sets and elements of sets to the standard vocabulary of Datalog.

There exist several different approaches for incorporating the concept of a complex structured object into the

formalism of Datalog [133], [119], [75]. One of the most well-known approaches has been developed within the LDL (Logic Data Language) Project, carried out at MCC, Austin, TX. The notation for representing sets and related issues used in this subsection is the one of LDL.

Examples of complex facts involving function symbols and sets are:

$$person(name(joe, berger), birthdate(1956, june, 30), children(\{max, sarah, jim\})).$$

$$person(name(joe, coker), birthdate(1956, june, 30), children(\{bill, sarah\})).$$

$$person(name(bebe, suong), birthdate(1958, may, 5), children(\{jim, max, sarah\})).$$

Here *name*, *birthdate*, and *children* are function symbols.

Variables may represent atomic objects (i.e., constants) or compound objects. The following rule relates the last names of all persons having the same birthdate and the same first name:

$$similar(X, Y) :- person(name(Z, X), B, C), person(name(Z, Y), B, D).$$

By this rule, we can derive, for instance, the new fact $similar(berger, coker)$.

Two sets are considered equal iff they contain the same elements, independently of the order in which these elements appear. The following rule defines a predicate $eqchilds(X, Y)$ stating that X and Y are the names of persons whose children have exactly the same first names:

$$eqchilds(X, Y) :- person(X, B, C), person(Y, D, C).$$

By this rule we can derive, for instance, $eqchilds(name(joe, berger), name(bebe, suong))$.

LDL offers several built-in predicates and functions for handling sets. The most important are:

- $member(t, S)$, a built-in predicate for expressing that t is an element of the set S. Notice that t can be a complex term and may contain sets as components.
- $union(S, A, B)$, a built-in predicate for expressing the $S = A \cup B$.

Sets can be introduced not only by enumeration but also by *grouping*. Grouping allows us to define a set in a rule head by indicating the properties of its elements in the corresponding rule body. The following rule, for instance, defines the set of all persons (identified by their last name) who have a child called Sarah.

$$sarahpar(<X>) :- person(name(A, X), B, C), member(sarah, C).$$

This rule generates the new fact $sarahpar(\{berger, coker, suong\})$.

Using complex objects in Datalog is not as easy as it might appear. Several problems have to be taken into consideration. First of all, the use of function symbols may endanger the safety of programs. It is undecidable whether a Datalog program with function symbols has a finite or an infinite result. The simplest solution is to leave the responsibility to the programmer. A similar problem is the finiteness of sets. Furthermore, not all Datalog programs with sets have a well-defined semantics. In particular, one should avoid self-referential set definitions such as $p(<X>) :- p(X)$. Such definitions come close to Russell's paradox. A large class of programs free of self-reference, called *admissible programs*, is defined in [21]. Note also that the test whether two terms (or literals) involving sets match is a computationally hard problem. This is a particular case of *theory unification* [116].

Another interesting problem is the consistency problem for monovalued data functions. A monovalued data function f is an evaluable function symbol, interpreted as a mathematical function. Such functions can be defined by the rules of a logic program. However, the unicity of the function value must be ensured. References [3], [76], and [77] deal with this problem.

A Logic Programming language for data manipulation such as LDL should be conceived in accordance with an appropriate data model which formalizes the storage and retrieval principles and the manipulation primitives that a DBMS offers for the objects referenced by the language. Pure Datalog, for instance, can be based on the relational model in first normal form (nowadays often called the *flat* relational model) because the concept of a *literal* nicely matches the one of a *tuple* in a relation and because the single evaluation steps of a Datalog program can be translated into appropriate sequences of relational operations. On the other hand, Logic Programming languages dealing with structured objects such as LDL require more complex data models.

Quite a number of extensions of the relational model have been developed in the last years in order to allow the storage and manipulation of complex objects. The most famous ones are the NF^2 model by Jaeschke and Schek [66], the model of nested relations by Fisher and Thomas [48], the model of Abiteboul and Beeri [1], [2] (which is more general than the former two models), the "Franco–Armenian Data model" FAD by Bancilhon, Briggs, Khoshafian, and Valduriez [19] (based on a calculus for complex objects by Bancilhon and Khoshafian [18]). ALGRES, a quite powerful data-model for complex objects, supports an extended relational algebra augmented with a fixpoint operator, and thus is an ideal base for implementing interpreters or compilers for logic data languages with complex objects [30], [31].

An excellent overview and comparison of data models for complex objects is given in [1].

F. An Overview of Research Prototypes

We mention here some of the research prototypes currently under development based on the language Datalog (or its variations).

The LDL project is under development at Microelectronics and Computer Technology Corporation (MCC),

Austin, TX. The project's goal is to implement an integrated system for processing queries in Logic Data Language (LDL), a language which extends Datalog. We gave some examples of the most significant constructs of LDL in Section VI. A general overview of the LDL project can be found in [43]. Features for dealing with complex terms in LDL are presented by Zaniolo [133]; a language overview is given by Tsur and Zaniolo [119]; the treatment of sets and negation is formally presented by Beeri *et al.* [21]; other papers on the subject are [19] and [73]. An excellent description of the LDL language and of related features is given in [92].

The NAIL! project (Not Another Implementation of Logic!) is under development at Stanford University with the support of NSF and IBM. NAIL! processes queries in Datalog, but interfaces a conventional SQL database system (running on IBM PC/RT). An overview of the NAIL! project is presented by Morris, Ullman, and Van Gelder in [88]; an update can be found in [89]. Various kinds of algorithms applied in the prototype are discussed in [90] and [121].

The KIWI Esprit project, sponsored by the EEC, is a joint effort for the development of knowledge bases, programmed through an object-oriented language (OOPS), and interfaced to an existing relational database. The Advanced Database Interface (ADE) of KIWI is developed jointly by the University of Calabria, CRAI, and ENIDATA (Italy). A general overview of KIWI and ADE can be found in [106a]. The mini-magic variation to the magic set approach, used in KIWI, is described by Sacca' and Zaniolo [104].

The ALGRES Project, under development in the frame of the METEOR Esprit project, is also sponsored by the EEC. The ALGRES project extends the relational model to support nonnormalized relations and a fixpoint operator, and supports Datalog as programming language. ALGRES is a joint effort of the Politecnico di Milano and of TXT-Techint (Italy). A general overview of the ALGRES Project can be found in a paper by Ceri, Crespi-Reghizzi *et al.* [32]. Other papers are [30] and [31].

Other relevant research projects which deal with rule-based computations are POSTGRES, under development at Berkeley University [117], and the 5th GENERATION Project [50], [91], under development at the Institute for New Generation Computer Technology (ICOT), Tokio, Japan. More details about the above-mentioned projects can be found in [34]. Other recent overviews of the projects on *databases* and *logic* were presented by Zaniolo [134] on a dedicated issue of IEEE-Data Engineering and by Gardarin and Simon [53] on TSI.

VII. Conclusions

There are no doubts that Datalog theory has been nicely developed in the last five years, through the flourishing of many elegant contributions. The main attraction of Datalog is the possibility of dealing, within a unique formalism, with nonrecursive expressions (or *views*) as well as with recursive ones. Although this area is still very active, we feel that some basic understanding has been established, thus allowing for systematic treatment.

One of the major challenges that Datalog research has still to meet is to convince the knowledge base community of the practical merits of this theory. The weaknesses of Datalog work have been indicated as follows.

a) Very few applications have been shown which can take full advantage of Datalog's expressive power. In particular, no useful applications have been reported so far for nonlinear or mutually recursive rules.

b) Datalog is not considered as a programming language, but rather as a "pure" computational paradigm. For instance, Datalog does not provide support for writing user's interfaces, and does not support quite useful programming tools, such as modularization and structured types.

c) Datalog does not compromise its clean declarative style in any way; while sometimes it is required that the programmer may take control on inference processing, by stating the order and method of execution of rules. This is typical, for instance, of many expert systems shells.

d) Datalog systems have been considered, until now, as closed worlds, that do not talk to other systems; while the current trend is towards supporting heterogeneous systems.

Some of the above criticisms are in fact well founded, and provide an indication of the directions in which we expect Datalog to move in order to become fully applicable. Datalog research will have to consider with great care the advances in other research areas; in particular, we have indicated in Section VI that Datalog can be extended to support complex terms; this is a first step towards the development of new language paradigms which use some of the concepts from object-oriented databases. The foundation of such an evolution have already been placed by Beeri [23] and Abiteboul and Kanellakis [5]; this work has shown that rule-based and object-oriented approaches are not in opposition, but rather they are capable of providing useful programming concepts to each other.

In summary, we expect that supporting rule computation will be one of the ingredients of future knowledge base systems; Datalog research has provided exact methods and a fairly good understanding for approaching this issue.

References

[1] S. Abiteboul and S. Grumbach, "Bases de donnees et objets complexes," *Tech. Sci. Inform.*, vol. 6, no. 5, 1987.

[2] S. Abiteboul and C. Beeri, "On the power of languages for the manipulation of complex objects," in *Proc. Int. Workshop Theory Appl. Nested Relations Complex Objects*, Darmstadt, West Germany, abstract, 1987.

[3] S. Abiteboul and R. Hull, "Data functions, datalog and negation," in *Proc. ACM-SIGMOD Conf.*, 1988.

[4] S. Abiteboul and V. Vianu, "Procedural and declarative database update languages," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1988, pp. 240–250.

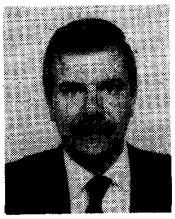
[5] S. Abiteboul and P. C. Kanellakis, "Object identity as a query language primitive," manuscript, 1989.

[6] P. Aczel, "An introduction to inductive definitions," in *The Hand-*

book of Mathematical Logic, J. Barwise, Ed. Amsterdam, The Netherlands: North-Holland, 1977, pp. 739-782.
[7] R. Agrawal and H. V. Jagadish, "Direct algorithms for computing the transitive closure of database relations," in *Proc. 13th Int. Conf. Very Large Databases*, Brighton, U.K., 1987.
[8] ——, "Alpha: An extension of relational algebra to express a class of recursive queries," in *Proc. IEEE 3rd Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1987.
[9] ——, "Multiprocessor transitive closure algorithms," *Data Eng.* (Special Issue on Databases for Parallel and Distributed Systems), vol. 12. Mar. 1989.
[10] A. V. Aho and J. D. Ullman, "University of data retrieval languages," presented at the 6th ACM Symp. Principles Programming Languages, San Antonio, TX, Jan. 1979.
[11] K. R. Apt and M. H. VanEmden, "Contributions to the theory of logic programming," *J. ACM*, vol. 29, no. 3, 1982.
[12] C. Apt, H. Blair, and A. Walker, "Towards a theory of declarative knowledge," *IBM Res. Rep.* RC 11681, Apr. 1986.
[13] F. Bancilhon, "Naive evaluation of recursively defined relations," in *On Knowledge Based Management Systems—Integrating Database and AI Systems*, Brodie and Mylopoulos, Eds. New York: Springer-Verlag, 1985.
[14] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs," in *Proc. ACM SIGMOD-SIGAT Symp. Principles Database Syst.*, Cambridge, MA, Mar. 1986.
[15] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing," in *Proc. ACM-SIGMOD Conf.*, May 1986.
[16] ——, "Performance evaluation of data intensive logic programs," in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Washington, DC, 1986.
[17] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets: Algorithms and examples," manuscript, 1986.
[18] F. Bancilhon and S. Khoshafian, "A calculus for complex objects," in *Proc. SIGMOD 86*, 1986.
[19] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, A powerful and simple database language," in *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, U.K., 1987.
[20] R. Bayer, "Query evaluation and recursion in deductive database systems," manuscript, 1985.
[21] C. Beeri, *et al.*, "Sets and negation in a logical database language (LDL1)," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, San Diego, CA, Mar. 1987.
[22] C. Beeri and R. Ramakrishnan, "On the power of magic," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, San Diego, CA, Mar. 1987.
[23] C. Beeri, "Data models and languages for databases," in *Proc. 2nd Int. Conf. Database Theory*, Bruges, Belgium, 1988; and in *LNCS 326*. New York: Springer-Verlag, 1988.
[24] J. Bocca, H. Decker, J.-M. Nicolas, L. Vielle, and M. Wallace, "Some steps toward a DBMS-based KBMS," in *Proc. IFIP World Conf.*, Dublin, 1986.
[25] J. Bocca, "On the evaluation strategy of EDUCE," in *Proc. ACM-SIGMOD Conf.*, Washington, DC, May 1986.
[26] M. L. Brodie, "Future intelligent information systems: AI and database technologies working together," in *Readings in Artificial Intelligence and Databases*. San Mateo, CA: Morgan Kaufman, 1988.
[27] F. Bry, "Logic programming as constructivism: A formalization and its application to databases," in *8th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1989, pp. 34-50.
[28] S. Ceri, G. Gottlob, and L. Lavazza, "Translation and optimization of logic queries: the algebraic approach," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Aug. 1986.
[29] S. Ceri and L. Tanca, "Optimization of systems of algebraic equations for evaluating Datalog queries," in *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, U.K., Sept. 1987.
[30] S. Ceri, S. Crespi Reghizzi, G. Gottlob, F. Lamperti, L. Lavazza, L. Tanca, and R. Zicari, "The ALGRES project," in *Proc. Int. Conf. Extending Database Technol. (EDBT88)*, Venice, 1988.
[31] S. Ceri, S. Crespi-Reghizzi, L. Lavazza, and R. Zicari, "ALGRES: A system for the specification and prototyping of complex databases," Dip. Elettronica, Politecnico di Milano, Int. Rep. 87-018, 1987.
[32] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L. Lavazza, and R. Zicari, "ALGRES: An advanced database system for complex applications," *IEEE Software*, to be published.
[33] S. Ceri, G. Gottlob, and G. Wiederhold, "Efficient database access through Prolog," *IEEE Trans. Software Eng.*, Feb. 1989.
[34] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*. New York: Springer-Verlag, to be published.
[35] U. S. Chakravarthy, J. Minker, and J. Grant, "Semantic query optimization: Additional constraints and control strategies," in *Proc. 1st Int. Conf. Expert Database Syst.*, L. Kerschberg, Ed., Charleston, 1986; and in *Expert Database Systems*. Menlo Park, CA: Benjamin-Cummings, 1987.
[36] U. S. Chakravarthy, J. Grant, and J. Minker, "Foundations of semantic query optimization for deductive databases," in *Proc. Int. Workshop Foundations Deductive Databases Logic Programming*, J. Minker, Ed., Aug. 1986.
[37] A. Chandra and D. Harel, "Horn clause queries and generalizations," *J. Logic Programming*, vol. 1, pp. 1-15, 1985.
[38] A. Chandra, "A theory of database queries," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, Mar. 1988.
[39] C. L. Chang and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic, 1973.
[40] C. C. Chang and H. J. Keisler, *Model Theory*. Amsterdam, The Netherlands. 1977.
[41] C. Chang, "On the evaluation of queries containing derived relations in relational databases," in *Advances in Database Theory*, Vol. 1, H. Gallaire, J. Minker, and J. M. Nicholas, Eds. New York: Plenum, 1981.
[42] C. L. Chang and A. Walker, "PROSQL: A Prolog programming interface with SQL/DS," in *Proc. First Workshop Expert Database Syst.*, Kiawah Island, SC, Oct. 1984; and in *Expert Database Systems*, L. Kerschberg, Ed. Menlo Park, CA: Benjamin-Cummings, 1986.
[43] D. Chimenti, T. O'Hare, R. Krishnamurthy, S. Naqvi, S. Tsur, C. West, and C. Zaniolo, "An overview of the LDL system," Special Issue on Databases and Logic, *IEEE Data Engineering*, vol. 10, Dec. 1987.
[44] K. L. Clark, "Negation as failure," in *Logic and Databases*. New York: Plenum, 1978.
[45] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. New York: Springer-Verlag, 1981.
[46] F. Cuppens and R. Demolombe, "A PROLOG-relational DBMS interface using delayed evaluation," presented at the Workshop on Integration of Logic Programming and Databases, Venice, Dec. 1986.
[47] P. Devanbu and R. Agrawal, "Moving selections into fixpoint queries," manuscript, Bell Labs, Murray Hill, NJ, 1986.
[48] P. Fischer and S. Thomas, "Operators for non-first-normal-form relations," in *Proc. 7th Int. Comput. Software Appl. Conf.*, Chicago, IL, 1983.
[49] M. Fitting, "A Kripke-Kleene semantics for logic programs," *J. Logic Programming*, vol. 2, pp. 295-312, 1985.
[50] K. Fuchi, "Revisiting original philosophy of fifth generation computer project," presented at the Int. Conf. Fifth Generation Comput. Syst., 1984.
[51] H. Gallaire and J. Minker, Eds, *Logic and Databases*. New York: Plenum, 1978.
[52] G. Gardarin and C. De Maindreville, "Evaluation of database recursive logic programs as recurrent function series," in *Proc. ACM-SIGMOD Conf.*, Washington, DC, May 1986.
[53] G. Gardarin and E. Simon, "Les systemes de gestion de bases se donnees deductives," *Technique et Science Informatiques*, vol. 6, 1987.
[54] G. Gardarin, "Magic functions: A technique to optimize extended datalog recursive programs," in *Proc. 13th Conf. Very Large Databases*, Brighton, U.K., 1987.
[55] G. Gentzen, "Die Wiederspruchsfreiheit der reinen Zahlentheorie," *Math. Annalen*, vol. 112, pp. 493-565, 1936.
[56] K. Gödel, "Die Vollständigkeit der Axiome des logischen Funktionenkalküls," *Monatshefte für Mathematik und Physik*, vol. 37, pp. 349-360, 1930.
[57] ——, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik und Physik*, vol. 38, pp. 173-198, 1931.
[58] A. Guttman, "New features for relational database systems to support CAD applications," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Berkeley, June 1984.
[59] L. J. Henschen and S. A. Naqvi, "On compiling queries in recursive first order databases," *J. ACM*, vol. 31, no. 1, 1984.
[60] D. Hilbert, "Axiomatisches Denken," *Mathematische Annalen 78*, pp. 405-415, 1918.

[61] Y. E. Ioannidis and E. Wong, "An algebraic approach to recursive inference," Electron. Res. Lab., Univ. California, Berkeley, Int. Rep. UCB/ERL M85/92, 1985.
[62] Y. E. Ioannidis, "On the computation of the transitive closure of relational operators," in *Proc. 12th Int. Conf. Very Large Databases*, Kyoto, Japan, 1986.
[63] Y. E. Ioannidis and E. Wong, "Transforming non-linear recursion into linear recursion," manuscript, 1987.
[64] Y. E. Ioannidis, J. Chen, M. A. Friedman, and M. M. Tsangaris, "BERMUDA—An architectural perspective on interfacing Prolog to a database machine," Dep. Comput. Sci., Univ. Wisconsin, Tech. Rep. 723, Oct. 1987.
[65] H. Itoh, "Research and development on knowledge base systems at ICOT," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986.
[66] B. Jaeschke and H. J. Schek, "Remarks on the algebra of non first normal form relations," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, Los Angeles, CA, 1982, pp. 124–138.
[67] J. F. Jenq and S. Sahni, "All pairs shortest paths on a hypercube multiprocessor," in *Proc. IEEE Int. Conf. Parallel Processing*, Aug. 1987.
[68] M. Kifer and E. L. Lozinskii, "Filtering data flow in deductive databases," in *Proc. 1st Int. Conf. Database Theory*, Roma, Sept. 1986.
[69] W. Kim, D. S. Reiner, and D. S. Batory, *Query Processing in Database Systems*. New York: Springer-Verlag, 1985.
[70] J. King, "Quist: A system for semantic query optimization in relational databases," in *Proc. 7th Int. Conf. Very Large Data Bases*, Cannes, 1981.
[71] Ph.G. Kolaitis, "On the expressive power of stratified datalog programs," Stanford Univ., Stanford, CA, preprint, Nov. 1987.
[72] Ph.G. Kolaitis and Ch.H. Papadimitriou, "Why not negation by fixpoint?," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1988, pp. 231–239.
[73] R. Krishnamurthy and C. Zaniolo, "Optimization in a logic based language for knowledge and data intensive applications," in *Proc. Int. Conf. Extending Database Technol. (EDBT88)*, Venice, 1988; and *LNCS*, No. 303. New York: Springer-Verlag, 1988.
[74] K. Kunen, "Negation in logic programming," *J. Logic Programming*, vol. 4, pp. 289–308, 1987.
[75] G. M. Kuper, "Logic programming with sets," in *Proc. ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1987, pp. 11–20.
[76] E. Lambrichts, P. Nees, J. Paredaens, P. Peelman, and L. Tanca, "MilAnt: An extension of datalog with complex objects, functions and negation," Dep. Comput. Sci., Univ. Antwerp, Int. Rep., 1988.
[77] E. Lambrichts, P. Nees, J. Paredaens, P. Peelman, and L. Tanca, "Integration of functions in the fixpoint semantics of rule based systems," in *Proc. 2nd Symp. Math. Fundamentals Database Theory*, Visegrad, Hungary, June 1989; and *LNCS*. New York: Springer-Verlag, 1989.
[78] J. W. Lloyd, *Foundations of Logic Programming*, 2nd extended ed. New York: Springer-Verlag, 1987.
[79] L. Löwenheim, "Über Möglichkeiten im Relativkalkül," *Mathematische Annalen*, vol. 76, pp. 447–470, 1915.
[80] D. W. Loveland, *Automated Theorem Proving: A Logical Basis*. Amsterdam, The Netherlands: North-Holland, 1978.
[81] E. Lozinskii, "Evaluating queries in deductive databases by generating," in *Proc. Int. Joint Conf. Artificial Intell.*, 1985.
[82] D. Maier and D. S. Warren, *Computing with Logic*. Menlo Park, CA: Benjamin-Cummings, 1988.
[83] A. I. Malcev, *Algebraic Systems*. New York: Springer-Verlag, 1973.
[84] V. Mannino, P. Chu, and T. Sager, "Statistical profile estimation in database systems," *ACM Comput. Surveys*, vol. 20, Sept. 1988.
[85] D. McKay and S. Shapiro, "Using active connection graphs for reasoning with recursive rules," in *Proc. 7th Int. Joint Conf. Artificial Intell.*, 1981.
[86] G. Marque-Pucheu, "Algebraic structure of answers in a recursive logic data-base," *Acta Inform.*, 1983.
[87] G. Marque-Pucheu, J. M. Gallausiaux, and G. Jomier; "Interfacing Prolog and relational database management systems," *New Applications of Databases*, Gardarin and Gelenbe, Eds. New York: Academic, 1984.
[88] K. Morris, J. D. Ullman, and A. Van Gelder, "Design overview of the Nail! system," in *Proc. Int. Conf. Logic Programming*. New York: Academic, 1986.
[89] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder, "YAWN! (Yet another window on NAIL!)," Special Issue on Databases and Logic, *IEEE Data Eng.*, vol. 10, Dec. 1987.
[90] K. A. Morris, "An algorithm for ordering subgoals in Nail!," in *Proc. ACM SIGNOD-SIGACT Symp. Principles Database Syst.*, Austin, TX, 1988.
[91] K. Murakami, T. Kakuta, N. Miyazaki, S. Shibayama, and H. Yokota, "A relational database machine, first step to knowledge base machine," in *Proc. 10th Symp. Comput. Architecture*, June 1983.
[92] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases*. New York: Computer Science Press, 1989.
[93] W. Nejdl, "Recursive strategies for answering recursive queries—the RQA/FQI strategy," in *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, U.K., Sept. 1987.
[94] E. Neuhold and M. Stonebraker, "Future directions in DBMS research," Int. Comput. Sci. Inst., Berkeley, CA, TR 88-01, May 1988.
[95] T. Przymusinski, "On the semantics of stratified deductive databases," in *Proc. Workshop Foundations Deductive Databases Logic Programming*, Washington, DC, 1986, pp. 433–443.
[96] ——, "Every logic program has a natural stratification and an iterated least fixed point model," in *8th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1989, pp. 11–21.
[97] Quintus Computer Systems Inc., Mountain View, CA, *Quintus Prolog Data Base Interface Manual*, version 1, June 29, 1987.
[98] R. Ramakrishnan, C. Beeri, and R. Krishnamurty, "Optimizing existential Datalog queries," in *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Syst.*, Austin, TX, Mar. 1988.
[99] ——, "Magic templates, A spellbinding approach to logic evaluation," in *Proc. Logic Programming Conf.*, Aug. 1988.
[100] D. Roelants, "Recursive rules in logic databases," Philips Res. Lab., Bruxelles, Rep. R513, Mar. 1987, submitted for publication.
[101] J. Rohmer, R. Lescoeur, and J. M. Kerisit, "The Alexander method: technique for the processing of recursive axioms in deductive databases," in *New Generation Computing*, vol. 4. New York: Springer-Verlag, 1986.
[102] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal recursion: A practical approach to supporting recursive applications," in *Proc. ACM SIGMOD 1986 Int. Conf. Management of Data*, Washington, DC, May 1986.
[103] K. A. Ross, "A procedural semantics for well founded negation in logic programs," in *8th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1989, pp. 22–32.
[104] D. Sacca' and C. Zaniolo, "On the implementation of a simple class of logic queries for databases," in *Proc. ACM 1986 SIGMOD-SIGACT Symp. Principles Database Syst.*, Cambridge, MA, Mar. 1986.
[105] ——, "Implementing recursive logic queries with function symbols," MCC Tech. Rep. DB-401-86, Dec. 1986.
[106] ——, "Magic counting methods," in *Proc. ACM-SIGMOD Conf.*, San Francisco, CA, May 1987.
[106a] D. Sacca, M. Dispinziezi, A. Mecchia, C. Pizzuti, C. Del Gracco, and P. Naggar, "The advanced database environment of the KIWI system," Special Issue on Databases and Logic, *IEEE Data Engineering*, vol. 10, no. 4, Dec. 1987.
[107] Y. Sagiv, "Optimizing Datalog programs," in *Proc. ACM 1987 SIGMOD-SIGACT Symp. Principles Database Syst.*, San Diego, CA, Mar. 1987.
[108] L. Schmitz, "An improved transitive closure algorithm," *Comput.*, vol. 30, 1983.
[109] C. P. Schnorr, "An algorithm for transitive closure with linear expected time," *SIAM J. Comput.*, vol. 7, May 1978.
[110] J. C. Shepherdson, "Negation as Failure II," *J. Logic Programming*, vol. 2, no. 3, pp. 185–202, 1985.
[111] ——, "Negation in logic programming," in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Los Altos, CA, 1988, pp. 19–88.
[112] O. Shmueli, and Sh. Naqvi, "Set grouping and layering in Horn clause programs," in *Proc. Int. Conf. Logic Programming*, 1987, pp. 152–177.
[113] T. Skolem, "Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen," *Skrifter utgit av Videnskapsselskapet i Kristiana, I, Mat-Nat. Klasse*, no. 4, 1920.

[114] D. E. Smith, M. R. Genesereth, and M. L. Ginsberg, "Controlling recursive inference," *Artifical Intell.*, vol. 30, no. 3, 1986.
[115] L. Sterling and E. Shapiro, *The Art of Prolog*. Cambridge, MA: M.I.T. Press, 1986.
[116] M. E. Stickel, "A unification algorithm for associative commutative functions," *JACM*, vol. 28, July 1981.
[117] M. Stonebraker and L. A. Rowe, Eds., "The Postgres papers," Univ. Calif, Berkeley, Mem. UCB/ERL M86/86, June 1987 (revised version).
[118] L. Tanca, "Optimization of recursive logic queries to relational databases" (in Italian) Ph.D. dissertation, *Politecnico di Milano and Universita' di Napoli*, 1988.
[119] S. Tsur and C. Zaniolo, "LDL: A logic-based query language," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, 1986.
[120] J. D. Ullman, "Implementation of logic query languages for databases," *ACM Trans. Database Syst.*, vol. 10, no. 3, 1985.
[121] J. D. Ullman and A. Van Gelder, "Testing applicability of top-down capture rules," Stanford Univ., Stanford, CA, Int. Rep. STAN-CS-85-1046; and *ACM-J.*, to be published.
[122] J. D. Ullman, *Principles of Databases and Knowledge-Base Systems*, Volume I. Potomac, MD: Computer Science, 1988.
[123] P. Valduriez and H. Boral, "Evaluation of recursive queries using join indices," in *Proc. 1st Int. Conf. Expert Database Syst.*, Charleston, SC, 1986.
[124] P. Valduriez and S. Khoshafian, "Parallel evaluation of the transitive closure of a database relation," *Int. J. Parallel Programming*, vol. 17, Feb. 1988.
[125] M. Van Emden and R. Kowalski, "The semantics of predicate logic as a programming language," *J. ACM*, vol. 4, Oct. 1976.
[126] A. Van Gelder, A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," in *7th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1988, pp. 221–230.
[127] A. Van Gelder, "The alternating fixpoint of logic programs with negation," in *8th ACM Symp. Principles Database Syst. (PODS)*, Mar. 1989, pp. 1-10.
[128] L. Vieille, "Recursive axioms in deductive databases: The Query-Subquery approach," in *Proc. Int. Conf. Expert Database Syst.*, L. Kerschberg, Ed., Charleston, 1986.
[129] ——, "A database complete proof procedure based on SLD resolution," ECRC, Munich, West Germany, Int. Rep. IR-KB-40, Nov. 1986.
[130] ——, "From QSQ to QoSaQ: Global optimization of recursive queries," in *Proc. 2nd Int. Conf. Expert Database Syst.*, L. Kerschberg, Ed., Tyson Corner, 1988.
[131] H. S. Warren, "A modification of Warshall's algorithm for the transitive closure of binary relations," *Commun. ACM*, vol. 18, Apr. 1975.
[132] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, June 1962.
[133] C. Zaniolo, "The representation and deductive retrieval of complex objects," in *Proc. 11th Int. Conf. Very Large Data Bases*, Aug. 1985.
[134] ——, Special Issue on Databases and Logic, *IEEE Data Eng.*, vol. 10, Dec. 1987.
[135] C. Zaniolo and D. Sacca', "Rule rewriting methods for efficient evaluation of Horn logic," MCC Tech. Rep. DB-084-87, 1987.
[136] M. M. Zloof, "Query-by-example: Operations on the transitive closure," IBM, Yorktown Heights, NY, RC 5526, 1975.



Stefano Ceri is a Professor of Computer Science at the Dipartimento di Matematica, University of Modena, Modena, Italy, and Visiting Professor at Stanford University, Stanford, CA, during the summer terms. Until 1986 he was with the Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy. His research interests include distributed databases, deductive and object-oriented databases, database design, medical databases, and the use of databases in software engineering. He is the author of numerous articles in these areas and coauthor of the book, *Distributed Databases, Principles and Systems* (New York: McGraw-Hill). He has been an active participant of several joint projects between the university and industry, sponsored by the National Research Council of Italy, the NSF, and Esprit (EEC).

Dr. Ceri is a member of ACM, Vice-Chairman of the IEEE Technical Committee on Data Engineering, and Associate Editor of the journals, *ACM Transactions on Database Systems* and *Distributed Computing*.

Georg Gottlob received the Dipl.-Ing. degree and the Doctorate in computer science from the Technical University of Vienna, Vienna, Austria.

From 1982 to 1984 he served as a Research Associate at the Politecnico di Milano, Milan, Italy, and from 1985 to 1988 he was a Staff Scientist at the Institute for Applied Mathematics, Italian National Research Council (C.N.R.), Genoa. During the Summers of 1985 and 1987, he lectured and performed research at Stanford University, Stanford, CA. Presently he is a Professor of Computer Science at the Technical University of Vienna, where he directs the Database and Expert-System Subdivision. His research interests are in the fields of databases, expert-systems, and applied mathematical logic.

Dr. Gottlob is a member of ACM, the IEEE Computer Society, and the Kurt Goedel Society.

Letizia Tanca received the Ph.D. degree in applied mathematics and computer science from the Politecnico di Milano, Milan, Italy, in 1988.

Prior to working towards the Ph.D. degree, she worked in industry as a Software Designer. At present she is Postdoctorate Fellow at the Politecnico di Milano. Her main research interests are the treatment of negative and functional information in relational and deductive databases and extensions to the relational data model (in Milan) within the project Algres (an extended relational environment for manipulating complex objects). She is also collaborating with Prof. Paredaens of the University of Antwerp (UIA), on the project of a rule-based language for manipulating complex objects and functions. She has written a book on the integration of *Relational Databases and Logic Programming*, coauthored with S. Ceri and G. Gottlob (New York: Springer-Verlag, to be published).