DATALOG

Grigoris Karvounarakis
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
gkarvoun@cis.upenn.edu

SYNONYMS

deductive databases

DEFINITION

An important limitation of relational calculus/algebra is that it cannot express queries involving "paths" through an instance, such as taking the transitive closure over a binary relation. *Datalog* extends conjunctive queries with recursion to support such queries. A Datalog program consists of a set of rules, each of which is a conjunctive query. Recursion is introduced by allowing the same relational symbols in both the heads and the bodies of the rules. A surprising and elegant property of Datalog is that there are three very different but equivalent approaches to define its semantics, namely the *model-theoretic*, *proof-theoretic* and *fixpoint* approaches. Datalog inherits these properties from logic programming and its standard language Prolog. The main restriction that distinguishes Datalog from Prolog is that function symbols are not allowed.

Several techniques have been proposed for the efficient evaluation of Datalog programs. They are usually separated into two classes depending on whether they focus on *top-down* and *bottom-up* evaluation. The ones that have had the most impact are centered around *magic sets rewriting*, which involves an initial preprocessing of the Datalog program before following a bottom-up evaluation strategy. The addition of negation to Datalog rules yields highly expressive languages, but the semantics above do not extend naturally to them. For Datalog$\neg$, i.e., Datalog with negated atoms in the body of the rules, *stratified* semantics, which impose syntactic restrictions on the use of negation and recursion, is natural and relatively easy to understand. The present account is based primarily on the material in [1]. Each of [1, 9, 2] has an excellent introduction to Datalog.[1] An informal survey can be found in [8]. The individual research contributions to Datalog are cited in the union of the bibliographies of these textbooks.

HISTORICAL BACKGROUND

Datalog is a restriction of the paradigm of *logic programming (LP)* and its standard programming language, Prolog, to the field of databases. What makes logic programming attractive is its *declarative* nature, as opposed to the more operational flavor of other programming paradigms, be they imperative, object-oriented, or functional. This led in the late 70's and in the 80's to much LP-related activity in Artificial Intelligence and even supercomputing (The Fifth Generation Project) which has later subsided dramatically. In databases this remains a useful paradigm, since the relational calculus is also a declarative language and LP provides a mechanism for extending its expressiveness with so-called *recursive queries*.

The name "Datalog" was coined by David Maier [1]. Research on recursive queries in databases picked up in

[1] We follow [8, 2] (rather than [1, 9]) in capitalizing the name.

the 80s and eventually led to several prototype *deductive database systems* [9, 8] whose data is organized into relations, but whose queries are based on Datalog.

Datalog has not quite made it as a practical query language due to the preeminence of SQL. When the need for recursive queries was recognized by RDBMS vendors, they preferred to extend SQL with some limited forms of recursion [8]. Nonetheless, more recent research on data integration has found Datalog to be a useful conceptual specification tool.

SCIENTIFIC FUNDAMENTALS

Datalog syntax

The syntax of Datalog follows that of the logic programming language Prolog with the proviso that only constants and relational symbols are allowed (no function symbols).

Definition 1. *Fix a relational schema. A* Datalog rule *has the form:*[2]

$$T(\mathbf{x}) \text{ :- } q(\mathbf{x}, \mathbf{y})$$

where $\mathbf{x} = x_1, \ldots, x_n$ *is a tuple of* distinguished *variables,* $\mathbf{y} = y_1, \ldots, y_m$ *is a tuple of "existentially quantified" variables,* T *is a relation and* q *is a conjunction of relational atoms. The left-hand side is called the* head *of the rule and corresponds to the output/result of the query and the right-hand side is called the* body *of the rule. Note that all distinguished variables in the head need to appear in at least one atom in the body, i.e., the rules are* range restricted. *A Datalog rule is identical to a conjunctive query in rule-based syntax, except that in the latter we don't always have a name for the head relation symbol. A Datalog program is a finite set of Datalog rules over the same schema. Relation symbols (a.k.a. predicates) that appear only in the body of the program's rules are called* edb *(extensional database) predicates, while those that appear in the head of some rule are called* idb *(intensional database) predicates. A Datalog program defines a Datalog query when one of the idb predicates is specified as the* output.

For example, if G is a relation representing edges of a graph, the following Datalog program P_{TC} computes its transitive closure in the output predicate T:

$$T(x, y) \text{ :- } G(x, y)$$
$$T(x, y) \text{ :- } G(x, z), T(z, y)$$

Semantics

Three different but equivalent definitions can be given for the semantics of Datalog programs, namely the *model-theoretic*, *proof-theoretic* and *fixpoint* semantics.

As in the entry RELATIONAL CALCULUS, we fix a countably infinite set $\mathbb{D}$ of constants as the sole universe for structures/instances. Since there are no function symbols, any relational instance over $\mathbb{D}$ is an *Herbrand interpretation* in the sense used in logic programming.

In the model-theoretic semantics of Datalog, we associate each rule with a first-order sentence as follows. First recall that as a conjunctive query, $T(\mathbf{x})$:- $q(\mathbf{x}, \mathbf{y})$ corresponds to the first-order query $T \equiv \{\mathbf{x} \mid \exists \mathbf{y} q(\mathbf{x}, \mathbf{y})\}$. To this we associate the sentence $\forall \mathbf{x}\ (\exists \mathbf{y}\ q(\mathbf{x}, \mathbf{y}) \rightarrow T(\mathbf{x}))$ which is clearly satisfied in a structure in which T is interpreted as the answer to the query. Note that this sentence is a *definite Horn clause*. More generally, given a Datalog program P, let Σ_P be the set of Horn clauses associated to the rules of P.

Let I be an input database instance, in this case an instance of the schema consisting only of edb predicates. A *model* of P is an instance of the entire schema (both edb and idb relation symbols) which coincides with I on

[2]The use of the symbol :- has its roots in Prolog, but some texts, e.g., [1], use the symbol $\leftarrow$instead, to convey the fact that each rule is closely related to a logical implication, as explained in the discussion of model-theoretic semantics.

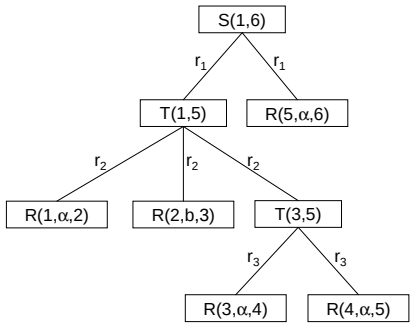

Figure 1: Proof tree

the edb predicates and which satisfies Σ_P. However, there can be infinitely many instances that satisfy a given program and instance of the edb relations. Thus, logic programming, and consequently Datalog use a *minimal* model, i.e., one such that no subset of it is also a model. This is usually understood as a manifestation of the *closed world assumption*: don't assume more than you need! It can be shown that for Datalog, there is exactly one minimal model, which is also the *minimum* model.

In the *proof-theoretic* approach of defining the semantics of Datalog, we first note that a tuple of constants in a relation can be seen as the head of a rule with empty body. Such rules are called *facts*. As we saw before, Datalog rules can be associated with first-order sentences. Facts correspond to just variable-free relational atoms. Now, the main idea of the proof-theoretic semantics is that the answer of a Datalog program consists of the set of facts that can be proven from the edb facts using the rules of the program as *proof rules*. More precisely, a *proof tree* of a fact A is a labeled tree where (1) each vertex of the tree is labeled by a fact; (2) each leaf is labeled by a fact in the base data; (3) the root is labeled by A; and (4) for each internal vertex, there exists an instantiation $A_1 \text{ :- } A_2, \ldots, A_n$ of a rule r such that the vertex is labeled A_1 and its children are respectively labeled $A_2, \ldots, A_n$ and the edges are labeled r.

Example 1. *Consider the program:*

$$\begin{array}{ll} (r_1) & S(x_1, x_3) \text{ :- } T(x_1, x_2), R(x_2, a, x_3) \\ (r_2) & T(x_1, x_4) \text{ :- } R(x_1, a, x_2), R(x_2, b, x_3), T(x_3, x_4) \\ (r_3) & T(x_1, x_4) \text{ :- } R(x_1, a, x_2), R(x_2, a, x_3) \end{array}$$

and the instance

$$\{R(1, a, 2), R(2, b, 3), R(3, a, 4), R(4, a, 5), R(5, a, 6)\}$$

A proof tree of $S(1, 6)$ is shown in Figure 1.

Because rule instantiation and application correspond to standard first-order inference rules (substitution and modus ponens), the proof trees are actually rearrangements of first-order proofs. This connects Datalog, through logic programming, to automated theorem-proving. One technique for constructing proofs such as the one above in a *top-down* fashion (i.e., starting from the fact to be proven) is *SLD resolution* [1]. Alternatively, one can start from base data and apply rules on them (and subsequently on facts derived this way) to create proof trees for new facts.

The third approach is an operational semantics for Datalog programs stemming from *fixpoint* theory. The main idea is to use the rules of the Datalog program to define the *immediate consequence* operator, which maps idb instances to idb instances. Interestingly, the immediate consequence operator can be expressed in relational algebra, in fact, in the SPCU (no difference) fragment of the relational algebra, enriched with edb relation names.

For example, the immediate consequence operator $\mathcal{F}$ for the transitive closure above is:

$$\mathcal{F}(T) \;=\; G \bowtie T \;\cup\; G$$

One way to think about this operator is that it applies rules on existing facts to get new facts according to the head of those rules. In general, for a recursive Datalog program, the same operator can be repeatedly applied on facts produced by previous applications of it. It is easy to see that the immediate consequence operator is *monotone*. Another crucial observation is that it will not introduce any constants beyond those in the edb instance or in the heads of the rules. This means that any idb instance constructed by iteration of the immediate consequence operator is over the *active domain* of the program and the edb instance. This active domain is finite, so there are only finitely many possible idb instances. They are easily seen to form a finite poset ordered by inclusion. At this point one of several technical variants of fixpoint theory can be put to work. The bottom line is that the immediate consequence operator has a *least fixpoint* which is an idb instance and which is the semantics of the program. It can be shown that this idb instance is the same as the one in the minimal model semantics and the one in the proof tree semantics. It can also be shown that this least fixpoint can be reached after finitely many iterations of the immediate consequence operator which gives a Datalog evaluation procedure called *bottom-up*.

Evaluation and Optimization of Datalog

The simplest bottom-up evaluation strategy, also called *naive* evaluation, is based directly on fixpoint Datalog semantics. The main idea is to repeatedly apply the immediate consequence operator on results of all previous steps (starting from the base data in the first step) until some step doesn't yield any new data. It is clear that naive evaluation involves a lot of redundant computation, since every step recomputes all facts already computed in previous steps. *Seminaive* evaluation tries to overcome this deficiency, by producing at every step only facts that can be derived using at least one of the new facts produced in the last step (as opposed to all previous steps).

In some cases, bottom-up evaluation can produce a lot of "intermediate" tuples that are not used in derivations of any facts in the output relation of the query. The top-down approach avoids this problem, by using heuristic techniques to focus attention on relevant facts, i.e., ones that appear in some proof tree of a query answer, especially for Datalog programs with constants appearing in some atoms. The most common approach in this direction is called the query-subquery (QSQ) framework. QSQ generalizes the SLD resolution technique, on which the proof-theoretic semantics are based, by applying it in sets, as opposed to individual tuples, as well as using constants to select only relevant tuples as early as possible. In particular, if an atom of an idb relation appears in the body of a rule with a constant for some attribute, this constant can be pushed to rules producing this idb. Similarly, "sideways information passing" is used to pass constant binding information between atoms in the body of the same rule. Such constant bindings are expressed using *adornments* or *binding patterns* on atoms in the rules, to indicate which attributes are *bound* to some constant and which are *free*.

Magic set techniques simulate the pushing of constants and selections that happens in top-down evaluation to optimize bottom-up evaluation. In particular, they rewrite the original Datalog program into a new program whose seminaive bottom-up evaluation produces the same answers as the original one, as well as producing the same intermediate tuples as the top-down approaches such as QSQ.

Datalog with Negation

The language Datalog¬ extends Datalog by allowing negated atoms in the body of the rules. Unfortunately, the semantics described above don't extend naturally to Datalog¬ programs. For example, if we follow the fixpoint semantics, there are programs that do not have a fixpoint or have multiple least fixpoints, or even if there is a least fixpoint the constructive method described above does not converge or its limit is not the least fixpoint. For model-theoretic semantics, uniqueness of the minimal model is not guaranteed. For these reasons, the common approach is to only consider a syntactically restricted use of negation in Datalog¬ programs, called *stratification*, for which natural extensions of the usual Datalog semantics do not have these problems. A stratification of Datalog¬ is a partition of its rules into subprograms that can be ordered in *strata*, so that for each relation R in the program, all rules defining R (i.e., with R in the head) are in the same stratum and for all atoms in the bodies

of those rules, the definitions of those relations are in a smaller or the same stratum, if the atom is positive, or strictly in a smaller stratum, for negative atoms.

For stratified Datalog¬ programs, one can evaluate within each stratum considering atoms of relations defined in smaller strata as edbs. Then, a negated atom is satisfied in the body of the rule if the corresponding tuple does not appear in that relation (as it appears in the base data or computed for the subprograms of smaller strata).

KEY APPLICATIONS*

Current and potential users and the motivation of studying this area.

Although Datalog was originally proposed as the foundation of deductive databases, which never succeeded in becoming part of commercial systems, it has recently seen a revival in the areas of data integration and exchange. This is due to the similarity of Datalog rules with popular *schema mapping* formalisms (*GLAV* or *tuple generating dependencies* [5]) used to describe relationships between heterogeneous schemas. In particular, [3] proposed the *inverse rules* algorithm for reformulating queries over a target schema to queries over source schemas in data integration. Other work has used Datalog rules to compute *data exchange* [7] or *update exchange* [4] solutions. In all these cases, the authors employed an extension of Datalog with *Skolem* functions in the head of rules, to deal with existentially quantified variables in the target of mappings. Another extension of Datalog, *Network Datalog (NDlog)* [6] allows the declarative specification of a large variety of network protocols with a handful of lines of program code, resulting to orders of magnitude of reduction in program size.

CROSS REFERENCE*

RELATIONAL CALCULUS, CONJUNCTIVE QUERY,

RECOMMENDED READING

Between 5 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] N. Bidoit. *Bases de Données Déductives: Présentation de Datalog.* Armand Colin, 1992.

[3] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based Heterogeneous Information Systems*, 43(1), 2000.

[4] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Revised version available as University of Pennsylvania technical report MS-CIS-07-26.

[5] M. Lenzerini. Tutorial - data integration: A theoretical perspective. In *PODS*, 2002.

[6] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD Conference*, pages 97–108, 2006.

[7] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, 2002.

[8] R. Ramakrishnan and J. Gehrke. *Database Management Systems, Third Edition.* McGraw-Hill, 2003.

[9] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Addison-Wesley, 1989.