OLDTNF-based evaluation method for handling recursive queries in deductive databases

CHEN Yangjun (陈阳军)
(Technical Institute of Changsha, Changsha 410004, China; Department of Computer Science, TU Chemnitz, 09107 Chemnitz, Germany)

Received August 28, 1997

Abstract  OLDTNF resolution is an important mechanism used in a Prolog interpreter. This mechanism is extended and improved for evaluating recursive queries in deductive databases. The key idea of the refinement is to distinguish between two classes of lookup nodes in an OLDTNF derivation and to handle them differently. First, reduce the search space by cutting off any subtree rooted at a lookup node of the first class. Further, speed up the evaluation by processing the second class in a second phase and generate many solutions directly from the solutions already produced (and the corresponding keys of solution lists) instead of evaluating them by expanding the corresponding subtrees in terms of the new solutions stored in solution lists.

Keywords: Prolog, SLD resolution, OLDTNF resolution, deductive databases, recursion.

It has been recognized for some time that first-order database query languages are lacking in expressive power\[1\]. Since then many higher-order query languages have been investigated in the context of deductive databases. A language that has received considerable attention recently is Datalog, the language of logic programs (known also as Horn-clause programs) without function symbols, which is essentially a fragment of fixpoint logic\[11\]. A canonical example of Datalog is the following program that computes transitive closure, where we think of the database as a directed graph.

\[
\begin{align*}
\text{path}(x, y) &: - \text{edge}(x, y), \\
\text{path}(x, y) &: - \text{edge}(x, z), \text{path}(z, y).
\end{align*}
\]

In this example, we take edge(·, ·) to be an extensional database (EDB) predicate, that is, representing basic facts stored in the database. For example edge(1, 5) is an EDB fact stating that there is an edge between nodes 1 and 5. The intensional database (IDB) predicate path(·, ·) represents facts deduced from the database via the logic program above: the first rule says that every directed edge forms a path, and the second rule tells how paths can be joined together. We can now query, for instance, path(1, 7) or path(2, v) to determine, respectively, whether there is a path from node 1 to node 7, or what nodes v are connected to node 2 by a path.

Recent works have addressed the problems of finding efficient evaluation methods for Datalog queries and developing optimization techniques for Datalog\[2-6\]. By “efficient”, we refer to the size of the database. A typical approach to the problem of efficient evaluation involves identifying “nice” properties of Datalog programs that facilitate efficient computation of programs with these properties. For example, Ullman and Van Gelder\[7\] have identified the polynomial fringe property of Datalog programs, where every proof involves a number of EDB facts (at the “fringe” of the proof tree) at most polynomials in the size of the database. Although the complexity of evaluating
arbitrary Datalog programs can be PTIME-complete\cite{8}, Ullman and Van Gelder have shown that the complexity of Datalog programs with the polynomial fringe property is in NC\cite{7}, that is, all facts can be deduced in parallel time polynomial in the logarithm of the size of the database, given a number of processors polynomial in the size of the database.

At first, the problem of optimizing Datalog queries does not seem to be too difficult, since every rule in a Datalog program can be viewed as a conjunctive query. Conjunctive queries constitute a fragment of the class of first-order queries for which the optimization problem is completely solved\cite{9}. Unfortunately, it is the recursive application of the rules that makes Datalog queries hard to evaluate. A possibility of optimizing Datalog queries is to remove recursions and to transform the original program to an equivalent one that contains no recursion. However, deciding whether it is possible to have some or all of the recursive predicates of a program removed, is undecidable\cite{10}. Then, we have to find an efficient method for a "direct" evaluation.

In this paper, we identify a new "nice" property, the solution similitude property, and try to extend OLDTNF resolution to a top-down but set-oriented method for evaluating recursive queries against a stratified database, thereby using the solution similitude property to improve the efficiency.

We say that a database is stratified if there exists a level mapping $r$, so that for every clause in the database (of the form):

$$B : - L_1, \ldots, L_n.$$  

If $L_i$ is positive, that is, an atom, then $r(L_i) \leq r(B)$, and if $L_i$ is negative, then $r(L_i) < r(B)$, for all $i, 1 \leq i \leq n$. So, in a stratified database, recursion via negation as in the following program is not allowed.

$$P : - q,$$

$$q : - p.$$  

The main obstacles which have to be faced in a top-down evaluation of recursive queries against a stratified database are

- Floundering (due to negated body literals), and
- Infinite derivations (due to the presence of recursions).

We say that a query is floundering if during the evaluation of the query a negated body literal with unrestricted variable is encountered. It leads to the problems of domain-dependency\cite{5} and inefficiency. Today, this problem is well understood. The simplest possibility to exclude floundering computations is to restrict the attention to allowed queries, which were introduced by Clark\cite{11}. (A rule is Clark-allowed if every variable appears in at least one positive body literal). However, to avoid the infinite derivations, the subsumption has to be checked to cut off any infinite branch. The problem is that cutting off an infinite branch may affect the completeness and thus a mechanism has to be designed to find the lost solutions again. To this end, OLDTNF resolution\cite{2,4} was proposed, which distinguishes between two kinds of nodes: solution nodes and lookup nodes, and handle them differently based on the tabulation technique. In this way, both termination and completeness can be guaranteed. We show that this method proceeds redundantly in certain cases and can be improved by further differentiating between two classes of lookup nodes. First, we try to reduce the search space that will be traversed by OLDTNF resolution. We do this by recognizing all similar portions of a graph and manage to produce all the relevant solutions by constructing only one of them (based on the identification of the first class of lookup