Using Logic Programming to Efficiently Evaluate Recursive Queries

Juliana Freire
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
juliana@cs.sunysb.edu

1 Introduction

Much of the success of the relational database model can be attributed to the declarativeness of its query language SQL. Unfortunately, SQL is not expressive enough. There are many useful queries (e.g., queries that involve recursion) which cannot be expressed in this language. Usually, when one wants to reason about the contents of a database, it is necessary to leave the relational model by embedding SQL into a lower-level language such as C, which results in an *impedance mismatch* and consequent loss of declarativeness.

Deductive databases [4, 5] address this problem by adopting logic programming or a restriction such as Datalog [11] as the query language. Prolog engines have been used to evaluate Datalog queries, but they have proven to be unacceptable for data-oriented queries for two major reasons: their poor termination and complexity properties for Datalog, and their tuple-at-a-time strategy.

A number of approaches have been used to attack the termination and complexity problems, most notably: (1) tabling [13], which has added features of database evaluation to logic programming languages, and (2) magic evaluation [1], which has embedded recursive goal-orientation in the framework of database evaluation. Magic evaluation closely resembles tabling. Both magic and tabling combine top-down goal orientation with bottom-up redundancy checking. Indeed, for range-restricted programs, they have been proven to be asymptotically equivalent [10, 8]. Despite these well-known equivalences, magic-style systems have traditionally differed from tabling systems. Magic-style systems, such as Aditi [12], LDL [2] and CORAL [7], are built upon set-at-a-time engines, and can use set-at-a-time operations like relational joins that may be made efficient for disk-resident data, while tabling systems, such as XSB [9], use a tuple-at-a-time strategy that reflects their genesis in the logic programming community. Presently for in-memory Datalog queries, due mainly to the use of Prolog compilation technology, the fastest tabling systems show an order of magnitude speedup over magic-style systems [9]. However, because of their tuple-at-a-time strategy, tabling systems cannot be efficiently extended to disk, and thus, they are not practical for applications that deal with massive amounts of data.

Examining tabling closely, we found that there is no reason why a set-at-a-time strategy cannot be closely integrated into a tabling engine. Doing so offers tremendous advantages: in-memory predicates can be evaluated at the best in-memory speeds, while queries to disk can have the same access patterns as the best set-at-a-time methods.

In what follows we give a brief overview of magic, introduce the basic ideas of tabling and describe how a set-at-a-time tabled evaluation can be formulated.

2 Evaluating Recursive Queries: Magic vs. Tabling

In semi-naive bottom-up evaluation [11], a derivation is made as follows: previously derived (or stored) facts are unified with each goal in the body of a rule; the head of the instantiated rule is the fact that is derived. Semi-naive is an incremental iterative fixpoint algorithm. It is iterative in that it repeatedly generates facts by applying program rules. It is incremental in the sense that no derivation step is repeated in the evaluation, that is, a rule uses a fact in a given position only once for further derivation.

A central difficulty of pure bottom-up evaluation, such as semi-naive, is that it is not goal-oriented: to answer a query, the entire model of a program must be constructed. Consider the following example.

Example 2.1 The recursive program below determines whether two individuals belong to the same generation.

```
sg(X,Y):- person(X), X=Y.
sg(X,Y):- par(X,Xp),sg(Xp,Yp),par(Y,Yp).
```

The first rule indicates that a person `X` is in its own generation, whereas the second rule indicates that `X` and `Y` are in the same generation if their corresponding parents belong to the same generation.

Given the following facts in a database

```
person(joe).     person(dave).   person(mary).    person(ann).
person(cain).    person(abel).   person(adam).    person(eve).
par(joe,ann).    par(joe,dave).  par(mary,ann).   par(mary,dave).
par(cain,adam).  par(cain,eve).  par(abel,adam).  par(abel,eve).
```

we wish to know the individuals which are in the same generation as `joe`, that is, we issue the query `sg(joe,Y)`. □

Under semi-naive evaluation, in order to answer this query, a number of facts irrelevant to our query would be generated (e.g., sg(cain,abel)). It is important that we answer this query examining only the part of the database that involves individuals somehow connected to `joe`. *Magic templates rewriting* (see e.g. [6]) addresses this problem by means of a program transformation. Magic is well-discussed in the literature [11]; here we give an example to illustrate the idea.

A magic transformation of the rules in Example 2.1 would be:

```
query(Y):- sg(joe,Y).
magic(sg(joe,Y)).
sg(X,Y):- magic(sg(X,Y)),person(X),X=Y.
sg(X,Y):- magic(sg(X,Y)),par(X,Xp),sg(Xp,Yp),par(Y,Yp).
magic(sg(Xp,Yp)):- magic(sg(X,Y)),par(X,Xp).
```

Intuitively, the `magic` facts correspond to queries or subgoals. The set of `magic` facts is used as a "filter" in the rules defining `sg` to avoid generating facts that are not answers to some subgoal. Thus a semi-naive evaluation of the rewritten program achieves *goal-directedness* — only facts relevant to the original query are created. For example, the semi-naive evaluation of the magic-transformed (SNMT) same generation program would proceed as follows.

- Iteration 0: `magic(sg(joe,Y))` added.
- Iteration 1: `sg(joe,joe)`, `magic(sg(ann,Y))`, `magic(sg(dave,Y))` added.
- Iteration 2: `sg(ann,ann)`, `sg(dave,dave)` added.
- Iteration 3: `sg(joe,mary)`, `sg(joe,joe)` each derived twice, `sg(joe,mary)` added.
- Iteration 4: Fixpoint.

In a tabled evaluation, a resolution-style search tree ensures goal-orientation rather than a rewrite of the program. Evaluation starts from the query as a goal, and uses the rules from head to body to create more goals. Tabled evaluations can be conveniently represented as forests of trees as in Figure 1, which represents the same generation program and database in Example 2.1 at the end of its evaluation. Consider the operations performed by a tabled evaluation. Analogous to the generation of magic facts in an SNMT evaluation, the first time a subgoal S is encountered during a tabled evaluation, S is registered in the table, and a new tree created with root S: Figure 1 contains three trees whose roots (*generators nodes*) are labeled `sg(joe,Y)`, `sg(2,Y)`, and `sg(3,Y)`. Program clause resolution is used to obtain the immediate children for each generator node. In addition, the node calling S becomes a *consuming node*, so named because it will consume answers produced by S's tree. Alternatively, if S is not new to the evaluation (S is contained in the table), no new tree is required for S. However, a consuming node is still created for S as in the previous case. Processing of answers is analogous: the first time an answer to a subgoal is derived during an evaluation it is added to the table and returned to relevant consuming subgoals; any subsequent derivations of the answer are failed. In this manner, redundant subcomputations (including loops) that sometimes happen in Prolog are prevented by tabling. Tabled resolution can be mixed with the program clause resolution of Prolog. In this case, we refer to tabled predicates and non-tabled predicates depending on the form of resolution used for each. In Figure 1, consuming nodes are represented by bold italics, and Prolog-style nodes are represented by non-bold italics.

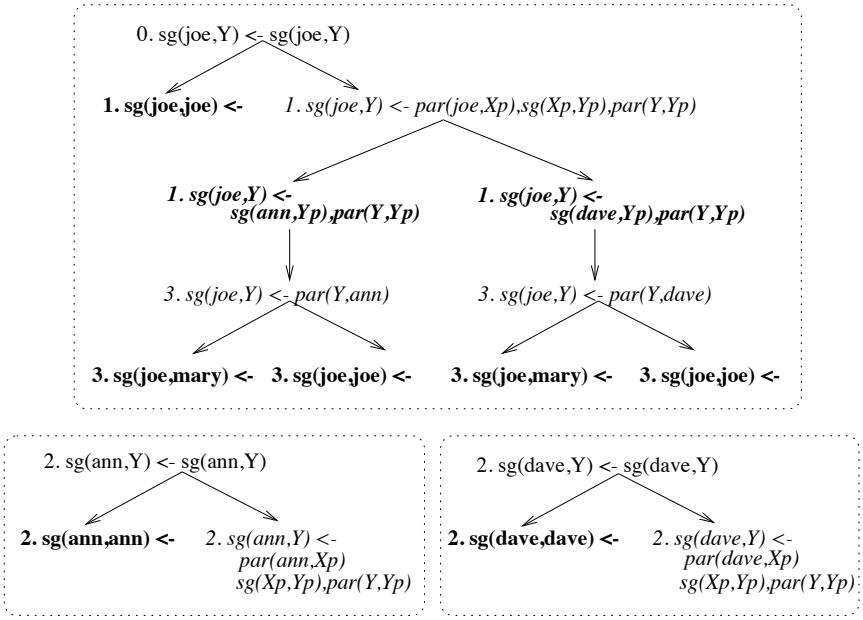

Figure 1: *SLG forest*

3 A Set-at-a-Time Tabulation Strategy

Note from the description of tabling that no assumptions are made about the linkage of the production of answers by trees in the forest and their return to consuming nodes. There is an intrinsic asynchrony between these two operations and this asynchrony can be exploited to construct a search strategy that resembles that of semi-naive. Consider the following strategy.

Tabled Breadth-First Search: Like the semi-naive evaluation of a magic-transformed program, tabled evaluations can also be seen as incremental iterative fixpoint computations. Resolution proceeds iteratively, in the following manner. Conceptually, all consuming nodes in all trees are visited in each iteration, although only answers derived during previous iterations can be returned to them. In addition, when a tabled subgoal S is called in iteration t, and S is new to the evaluation, the evaluation must wait until iteration $t+1$ to create a new tree with S as root. Notationally, we can denote the set of all answers added at iteration t or earlier as Ans_t, and the set of answers added at iteration t itself as a *delta set* of answers, δAns_t. Similarly for subgoals we define the sets $Subg_t$ and $\delta Subg_t$

Let us consider the actions of tabled breadth-first search for the same-generation program in Example 2.1. Iteration 1 begins when the query `sg(joe,Y)` is called. Program clause resolution is used to derive the first answer `sg(joe,joe)`. Execution then backtracks to the second program clause for `sg/2` which eventually selects the subgoal `sg(ann,Yp)`. Note that program clause resolution for `sg(ann,Y)` will not be used until iteration 2. Rather, the engine suspends this subgoal — at this point this subgoal is simply added to the delta set of subgoals $\delta Subg_1$ — and fails, causing it to execute the next available clause, producing the selected subgoal `sg(dave,Yp)`. Eventually, there will be no available clauses and the engine will perform a *fixpoint check*, and then start iteration 2. The engine scans the delta sets of subgoals and answers from iteration 1 to resume suspended subgoals (which will use program clause resolution) and consuming nodes (which will consume answers created in the previous iteration). For our example, iteration 2 begins by performing program clause resolution for `sg(ann,Y)`. This produces, among other clauses, the answer `sg(ann,ann)`. This answer is added to the delta set of answers δAns_2 and will be returned in iteration 3 to the consuming node `sg(joe,Y)` $\leftarrow$ `sg(ann,Yp),par(Y,Yp)`. During iteration 2, program clause resolution is also performed for `sg(dave,Y)`, and its actions parallel those of `sg(ann,Y)`. The process continues, with the engine scanning nodes via the delta sets at each fixpoint iteration, and then either performing program clause resolution for the delta set of subgoals or returning answers in the delta set of answers to consuming nodes. The evaluation terminates when neither new answers nor new subgoals exist. Note that, at each iteration, the breadth-first tabling engine generates the same answers as the SNMT evaluation.

Tabled Breadth-First Search was formally defined in [3] where we prove that any fact derived by this strategy is also derived by SNMT, that is, on an iteration-by-iteration basis, Tabled Breadth-First Search is equivalent to Semi-Naive Magic Template evaluation. Experimental results presented in [3] show that an implementation of the breadth-first strategy gives excellent times for queries that involve disk-resident data without sacrificing in-memory performance.

4 Conclusion

We have presented a breadth-first scheduling strategy for tabled evaluations that combines fast in-memory processing of top-down logic programming systems with set-at-a-time access to external

data. Thus, serious database interaction, such as that needed in data mining, decision support, or a variety of other applications, can be done efficiently within the structure of first-order logic and the programming environment of Prolog.

Even though this concept has been implemented in a system, a number of improvements are needed. For instance, the set-at-a-time interface used to couple the breadth-first tabling engine and relational databases is in an initial state of development. When these practical issues are resolved, our framework will contain an efficient procedural component and an efficient data retrieval component, both using the language of first-order logic and tightly integrated in a tabling abstract machine. We believe this framework will form a computational basis to combine the fields of logic programming and deductive databases.

Acknowledgements: I would like to thank David S. Warren and Terrance Swift for useful discussions and for their contributions to this work. This work was supported in part by Capes-Brazil, and NSF grants CDA-9303181 and CCR-9404921.

References

[1] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3):255–299, 1991.

[2] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2:76–90, 1990.

[3] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *Proceedings of the International Conference on Logic Programming (ICLP)*, 1997. To appear.

[4] H. Gallaire and J. Minker. *Logic and Databases*. Plenum Press, New York, 1978.

[5] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16:153–185, 1984.

[6] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11:189–216, 1991.

[7] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations, and logic. In *Proc. of the 18th International Conference on Very Large Data Bases (VLDB)*, pages 238–250, 1992.

[8] K.A. Ross. Modular stratification and magic sets for datalog programs with negation. *JACM*, 41(6):1216–1266, 1994.

[9] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proceedings of SIGMOD*, pages 442–453, 1994.

[10] H. Seki. On the power of Alexander templates. In *Proceedings of PODS*, pages 150–159, 1989.

[11] J. Ullman. *Principles of Data and Knowledge-base Systems Vol I and II*. Computer Science Press, 1989.

[12] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi deductive database system. *The VLDB Journal*, 3(2):245–288, 1994.

[13] D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.