Taking I/O Seriously: Resolution Reconsidered for Disk

Juliana Freire, Terrance Swift and David S. Warren
Department of Computer Science
State University of New York at Stony Brook
{juliana,tswift,warren}@cs.sunysb.edu

Abstract

Modern compilation techniques can give Prolog programs, in the best cases, a speed comparable to C. However, Prolog has proven to be unacceptable for data-oriented queries for two major reasons: its poor termination and complexity properties for Datalog, and its tuple-at-a-time strategy. A number of tabling frameworks and systems have addressed the first problem, including the XSB system which has achieved Prolog speeds for tabled programs. Yet tabling systems such as XSB continue to use the tuple-at-a-time paradigm. As a result, these systems are not amenable to a tight interconnection with disk-resident data.

However, in a tabling framework the difference between tuple-at-a-time behavior and set-at-a-time can be viewed as one of scheduling. Accordingly, we define a breadth-first set-at-a-time tabling strategy and prove it *iteration equivalent* to a form of semi-naive magic evaluation. That is, we extend the well-known asymptotic results of Seki [10] by proving that each iteration of the tabling strategy produces the same information as semi-naive magic. Further, this set-at-a-time scheduling is amenable to implementation in an engine that uses Prolog compilation. We describe both the engine and its performance, which is comparable with the tuple-at-a-time strategy *even for in-memory Datalog queries*. Because of its performance and its fine level of integration of Prolog with a database-style search, the set-at-a-time engine appears as an important key to linking logic programming and deductive databases.

1 Introduction

It is often necessary to leave the relational model to reason about the contents of a database, a problem which deductive databases seek to remedy. Deductive databases choose as their data model first-order logic or a restriction such as Datalog. First-order logic has proven expressive as a data query language, and many evaluation strategies, most notably magic evaluation, have been developed to embed recursive goal-orientation in the framework of database evaluation. While magic adds goal-orientation to database evaluation, tabling methods have added features of database evaluation to logic programming languages.

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] under certain assumptions. Despite these well-known equivalences, magic-style systems have traditionally differed from tabling systems. Magic-style systems, such as Aditi [15], CORAL [7], and LDL [3], are built upon set-at-a-time semi-naive

engines, while tabling systems, such as XSB [9], use a tuple-at-a-time strategy that reflects their genesis in the logic programming community. Each class of systems has its advantages and disadvantages. Presently for in-memory Datalog queries, the fastest tabling systems show an order of magnitude speedup over magic-style systems due to the tabling systems' use of Prolog compilation technology [9]. However, the tuple-at-a-time strategy of tabling systems is not efficiently extendible to disk.

A close look at tabling indicates 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 have the same access patterns as the best set-at-a-time methods. Furthermore, both of these approaches can be integrated fully into the well-known Prolog environment. This paper presents both theoretical and practical results on using Prolog compilation technology to efficiently implement a set-at-a-time tabling system for definite programs. The major results of this paper are:

- *Derivation of a Tight Equivalence between Tabling and the Semi-Naive Evaluation of a Magic-Transformed Program (SNMT).* Broad equivalences between tabling and magic-style methods have long been known. In [10] Seki obtained an asymptotic equivalence between a naive evaluation of a program rewritten using Alexander Templates and a version of a tabling method. After specifying a *breadth-first* search strategy for tabling, we extend the equivalence of Seki in two ways. First, we use a semi-naive evaluation rather than naive evaluation for both the tabling and the rewrite method. Second and more importantly, we define a natural measure of iteration equivalence between set-at-a-time tabling and SNMT. Using iteration equivalence, we demonstrate that every answer of each iteration of our tabling strategy is produced at the corresponding iteration of SNMT.
- *Design and Implementation of an Engine to Evaluate Breadth-First Tabling.* Other tabling methods with set-at-a-time properties have been developed, most notably the SLD-AL strategy of [16]. In addition to iteration equivalence to magic, the engine described here has the advantage of using the low-level data structures and compilation techniques of Prolog technology. As presented in [13], this technology, as implemented in the SLG-WAM, leads to an extremely fast, robust, and flexible implementation of a deductive database engine. The resulting implementation of the Breadth-First XSB is available upon request. Other versions of XSB are currently installed at nearly 1000 registered commercial, academic and governmental sites.
- *Performance Analysis of the Set-at-a-time Engine.* Surprisingly, the resulting set-at-a-time engine is only marginally slower than previously published SLG-WAM times for Datalog programs with in-memory data on a representative set of programs, and can be much faster for queries involving external relations. An analysis of the performance of different engines is presented.

2 Preliminaries

In this section we give a brief description of magic templates and introduce the basic ideas of tabling.

Semi-Naive Evaluation and Magic Rewriting

The well-known semi-naive evaluation algorithm [14] 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 a given rule uses a given fact in a given position only once for further derivation. To support this incrementality, a form of timestamp denoting the iteration number is explicitly represented in both answers and rules. 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. If answers for a particular predicate P are desired, they may be denoted as Ans_t^P or δAns_t^P.

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. *Magic templates rewriting* (see e.g. [6]) avoids this problem by means of a program transformation. Magic is well-discussed in the literature; here we present only the magic templates transformation along with an example of its use. Note that this transformation requires a statically-determined computation rule. We assume without loss of generality that all such computation rules have a left-to-right order.

Definition 2.1 (Magic Templates Rewriting [11]) Let P be a program and let Q be a query to P. The *magic rewrite* of P for Q, or $M(P,Q)$ is constructed as follows.

1. Add a seed fact $magic(Q)$.
2. For each rule R in P add the *modified version* of the rule to *M(P)*. For each *R'* in *M(P)*, if *R'* has head $p(\vec{X})$ add the literal $magic(p(\vec{X}))$ to the body of R as the first goal.
3. For each rule R with head $p(\vec{X})$ and each occurrence of a derived literal $q(\vec{Y})$ in its body, add a *query rule*, whose head is $magic(q(\vec{Y}))$ and whose body contains the literals preceding $q(\vec{Y})$ in R.

Example 2.1 Consider the same generation program

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

The magic templates rewrite of this program for query `:-sg(1,Y)` is written as

```
query(Y):- sg(1,Y).
magic(sg(1,Y)).
sg(X,Y):- magic(sg(X,Y)), 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).
```

Clearly optimizations can be made to the above program. For instance, the magic rewrite could take advantage of the instantiation pattern of the original query, and of the left-to-right order of rule evaluation to infer that all calls to `sg/2` would have the first argument instantiated and the second free. In this case, the second argument of `sg/2` would not need to be used by the magic facts. However, such an optimization would not affect the complexity of the ensuing program.

Given the simple EDB:

```
par(1,3).  par(1,4).  par(2,3).  par(2,4).
```

The semi-naive evaluation of the rewritten same generation program would proceed as follows.

- Iteration 0: `magic(sg(1,Y))` added.

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

Tabled Evaluations

Like SNMT evaluations, tabled evaluations can also be seen as incremental iterative fixpoint computations. In a tabled evaluation, it is a resolution-style search tree that ensures goal-orientation rather than a rewrite of the program. In fact, tabled evaluations are conveniently modeled 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. Analogously 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 trees whose roots are labeled `sg(1,Y)`, `sg(2,Y)`, and `sg(3,Y)` (we will also refer to these roots as *subgoal* or *generator* nodes). Program clause resolution is then used to obtain the immediate children for each root 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) are prevented by tabling. Of course, tabled resolution can be mixed with the program clause resolution of SLD. In this case, we refer to tabled predicates and non-tabled predicates depending on the form of resolution used for each. Nodes whose selected literal is non-tabled are called *interior* nodes. In Figure 1, consuming nodes are represented by bold italics, and interior nodes (that use SLD resolution) are represented by non-bold italics. Note that each node is labeled with the corresponding iteration.

In a tabled evaluation, groups of mutually dependent subgoals are called *strongly connected components* or SCCs. When all program and answer clause resolution has been performed for the subgoals in an SCC, the subgoals are termed *completely evaluated* or *completed*. At completion, all trees for subgoals in the SCC can be disposed since at this point the table contains all pertinent information for the subgoals. The notion of completion is necessary for evaluation of programs with negation, as well as being useful for space reclamation.

To summarize, tabled resolution in general has five types of operation:

1. SUBGOAL CALL: creates a consuming node, along with a new tree for the subgoal if it is not present in the evaluation.
2. PROGRAM CLAUSE RESOLUTION: used for all non-tabled (SLD) subgoals, and for immediate children of the root of each tree.
3. NEW ANSWER: adds a new answer to the table.
4. ANSWER RESOLUTION: resolves the selected literal of a consuming node against an answer from a table.
5. COMPLETION: determines when a set of subgoals is completely evaluated, and disposes of their trees.

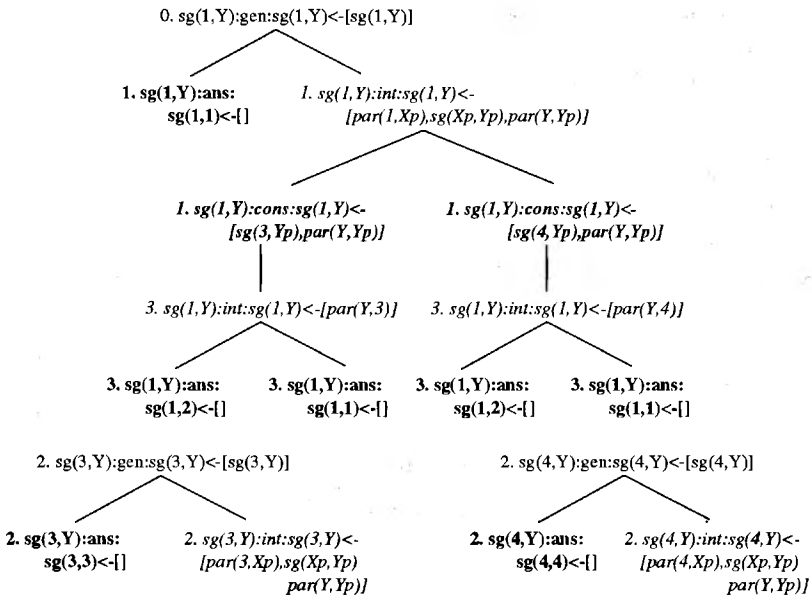


Figure 1: *SLG forest*

In describing search strategies, it is often convenient to represent a particular node within a tree. To do this, the information in each node can be prepended by the label of the root of its tree (or *root goal*) and the node's *status*. The statuses *consuming*, *answer*, *interior* and *generator* have been introduced (see Figure 1); we also use the status *new* for leaf nodes for which program clause resolution may be applicable, but has yet to be done.

3 Tabling and Magic: Iteration Equivalences

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 may be exploited to construct a search strategy that resembles that of semi-naive. Consider the following strategy.

Breadth-First Search: Resolution proceeds iteratively, in the following manner. Conceptually, all consuming nodes in all trees are visited in each iteration, although answer resolution may only use answers derived during previous iterations. 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.

Some notation will be useful to describe the Breadth-First Search in detail.

- $\delta Subgoals_t$ is the set of non-completed tabled subgoals added during an iteration t;
- δCns_t^S is the set of consuming nodes in tree S added during iteration t (i.e., $S : consuming : S' \leftarrow Body$ might be in δCns_t^S for some t);

- δAns_t^S is the set of answers that have been added to tree S during iteration t (or equivalently, the set of all answers that have been added to the table for S during iteration t).

The sets of all answers, consuming nodes and subgoals for S at t can be taken by unioning the deltas for S over all times less than or equal to t. We denote these sets as Ans_t^S, Cns_t^S and $Subgoals_t$, respectively.

Breadth-First Search is specified by Algorithms 3.1, 3.2 and 3.3. The iterations of Breadth-First Search are captured by the *repeat* loop of Breadth-First Main, which is executed until a fixpoint condition is satisfied — when no operations of the five types listed in Section 2 are left to perform. In Breadth-First Search, this condition occurs when at some iteration t, δAns_t, δCns_t, and $\delta Subgoals_t$ are empty for all subgoals. (This is represented by the check Fixpoint Condition(t) in line 18 of Algorithm 3.1, Breadth-First Main). Breadth-First Main starts at time $t = 0$ by placing the initial goal in the set $\delta Subgoals_0$. Its iterations then work as follows. First, program clauses are resolved against any subgoals created in the previous iteration (or against the initial goal). This occurs in line 5 of Algorithm Breadth-First Main, where these resolutions create new nodes for the tree S, which are kept temporarily in $ClauseCache_{BFM}$ during each iteration. Next, in lines 7-15, resolutions are performed for the *new* nodes created within the present iteration. If the node has an empty goal list, the status of the node becomes *answer* and it is added to δAns_t^S (line 9-10). If the body is not empty and the selected literal is tabled, the status becomes *consuming*, and the selected literal added to $\delta Subgoals_t$, if necessary (lines 11-13). Finally, in order to intermix tabling with SLD, if the selected literal is not tabled, the subtree rooted in the node is expanded using program clause resolution via Algorithm Get Program Clause Closure (lines 14-15). As can be seen from Algorithms 3.1–3.3, Get Program Clause Closure has its own fixpoint operation, so that the search is breadth-first only if all derived predicates are tabled. The following statements in lines 16-17 perform answer resolution as necessary. The routine Perform Answer Resolution is called twice, once to resolve old answers against new consuming nodes and again to resolve new answers against old consuming nodes. In the course of its evaluation, specified in Algorithm Perform Answer Resolution, Get Program Clause Closure will again be called.

Algorithm Perform Answer Resolution visits all consuming nodes in a particular input set. In the second call to Perform Answer Resolution in line 17 of Breadth-First Main, the routine is called with δCns_t^S as its consuming set. However in line 10 of Perform Answer Resolution, new consuming nodes are added to this same consuming set. The pseudo-code should be read as executing using call-by-reference so that all consuming nodes created at time t have returned to them any answer created before t.

The strategy described in Algorithms 3.1–3.3 is in fact breadth-first according to a distance metric presented in the full version of this paper[1], and the following theorems show that this metric is a natural one for tabling. Theorem 3.1 shows that Breadth-First Search explores a derivation tree in a manner closely akin to SNMT evaluation. As a notational device, for a given program, P, let $T(P)$ be a program in which each intensional predicate is declared as tabled: we may refer to $T(P)$ as a fully tabled program.

Theorem 3.1 Let P be a definite program and let $M(P,Q)$ be its magic rewrite for a query Q and $T(P)$ be the fully tabled program, and assume Q is an element

[1]The expanded version is available at *http://www.cs.sunysb.edu/~sbprolog*.

Algorithm 3.1 Breadth-First Main

Initialize all sets to $\emptyset$; $\delta Subgoals_0 = \{\text{initial goal}\}$; $t = 0$;
Repeat
  increment t; $ClauseCache_{BFM} = \emptyset$
  For each subgoal S in $\delta Subgoals_{t-1}$
    Resolve S against each unifying program clause and add the result to $ClauseCache_{BFM}$;
  For each subgoal S in $Subgoals_{t-1}$
    For each $S : new : S' \leftarrow Body \in ClauseCache_{BFM}$
      If $Body$ is empty
        If S' is not redundant in Ans_t^S add $S : ans : S'$ to δAns_t^S
      Else if the selected literal L of $Body$ is tabled
        Add $S : consuming : S' \leftarrow Body$ to δCns_t^S
        If L is not in $Subgoals_t$ add L to $\delta Subgoals_t$
      Else if the selected literal L of $Body$ is non-tabled
        Get Program Clause Closure($S : new : S' \leftarrow Body$)
    Perform Answer Resolution(Cns_{t-1}^S,δAns_{t-1})
    Perform Answer Resolution(δCns_t^S,Ans_{t-1})
Until (Fixpoint Condition(t))

Algorithm 3.2 Get Program Clause Closure(NewClause)

$ClauseCache_{PCC}$ = NewClause;
Repeat
  Choose from $ClauseCache_{PCC}$ a clause $Clause = S : new : S' \leftarrow Body$
  For each program clause C unifying with the selected literal L of $Body$
    Produce $C_{new} = S : new : ((S' \leftarrow NewBody)\theta)$,
      where θ is the mgu of C and L
      and $Newbody$ is the resolvent of C and $Body$ on L
    If $NewBody$ is empty
      If $S'\theta$ is not redundant in Ans_t^S add $S : ans : (S'\theta)$ to δAns_t^S
    Else if the selected literal L' of $NewBody$ is tabled
      Add C_{new} to δCns_t^S
      If L' is not in $Subgoals_t$ add L' to $\delta Subgoals_t$;
    Else if the selected literal L' of $NewBody$ is non-tabled
      Add C_{new} to $ClauseCache_{PCC}$
Until (ClauseCache is empty)

Algorithm 3.3 Perform Answer Resolution(ConsumingSet,AnswerSet)

While there exists an unvisited node in $ConsumingSet$
  Choose a node Cns in $ConsumingSet$, and let S be the root goal for Cns
  Mark all answers in $AnswerSet$ as unvisited;
  While there exists an unvisited answer in $AnswerSet$
    Choose an unvisited answer Ans from $AnswerSet$;
    Let $NewCns$ represent the Resolvent of Cns with Ans
    If the body of $NewCns$ is empty
      If $NewCns$ is not in Ans_t^S add it to δAns_t^S;
    Else if the selected literal L of $NewCns$ is tabled
      Add $NewCns$ to δCns_t^S, marked as unvisited
      If L is not redundant in $Subgoals_t$ add L to $\delta Subgoals_t$;
    Else if the selected literal L of $NewCns$ is non-tabled
      Get Program Clause Closure($NewCns$)
    Mark Ans as visited;
  Mark Cns as visited

of $Subgoals_0$. Then, at each iteration t:

1. An SNMT evaluation of P for Q derives a non-magic fact A if Breadth-First Search derives an answer A.
2. An SNMT evaluation of P for Q produces a fact *magic(S)* if Breadth-First Search adds S to $Subgoals_t$.

Proof: The details of the proof are given in the full version of this paper. □

Soundness of Breadth-first Search follows immediately from Theorem 3.1 and correctness of SNMT evaluation. Completeness of Breadth-First Search is proven separately by the following theorem.

Theorem 3.2 Let P be a fully tabled definite program evaluated by Breadth-first Search, and M_P be the least model of P and G be an element of M_P. Then, at some iteration t there is a subgoal S such that G is subsumed by an element of Ans_t^S.

Proof: The details of the proof are given in the full version of this paper. □

Taken together, Theorems 3.2 and 3.1 indicate that any fact derived by Breadth-First Search will also be derived by SNMT *in the same iteration*[2]. While it has long been known that logic is an expressive data query language, these results now indicate that compilation techniques of logic programming can have a direct practical impact on implementing database queries.

4 The Breadth-First SLG-WAM

In this section we describe the changes to make the search procedure of the SLG-WAM [12] breadth-first. Because of space limitations, some implementation details have been omitted; more details on scheduling in the SLG-WAM can be found in [4], and in the full version of this paper.

We begin by briefly presenting some data structures used by the SLG-WAM engine. The table maintains information about all (tabled) subgoals encountered by the evaluation as well as answers for each subgoal. These answers are maintained in a trie-like structure whose leaves are chained together in an *answer list*, which represents the sequence in which answers are derived by the evaluation and, by extension, delta sets. A table entry is created when a new tabled subgoal S is called during the SUBGOAL CALL operation. At this time a *generator* choice point and a *completion frame* are created for S. The choice point frame contains a superset of the information present in a regular WAM [1] choice point, and is used to schedule program clause resolution for S. One important difference is that the trust instruction sets a completion instruction onto the instruction field of the generator choice point, rather than disposing of the choice point as in the WAM. A completion instruction is thus not invoked until after all program clause resolution is performed for a subgoal. The completion frame for S resides on the SLG-WAM *completion stack* and, among other functions, provides an entry point to a chain of consuming nodes for S, as well as to the generator choice point for S. If S is not new, a consuming choice point is created, which will be used to schedule ANSWER RESOLUTION; information in the consuming choice point will be used to reconstitute

[2]The theorems also indicate that, for non range-restricted programs Breadth-First Search may be more efficient than SNMT. Differences between the two methods arise in non-ground programs, but we believe that these these differences can be obviated by the use of alternate magic rewriting techniques developed to reduce the complexity of magic with respect to Prolog [11].

the environment in which the subgoal was called, so that answers can be returned to this environment as they are derived.

The original strategy used by the SLG-WAM is tuple-at-a-time, both because it schedules ANSWER RESOLUTION as answers are derived and because it evaluates subgoals as soon as they are called. For Breadth-First SLG, however, answers and subgoals have to be batched so that they are resolved in the appropriate iteration, as defined in Section 3. The following example provides a high-level illustration of the actions of the breadth-first engine on the same-generation program of Example 2.1.

Example 4.1 Consider how the same generation program from Example 2.1 might be executed by a WAM-style breadth-first tabling engine (see Figure 1). Iteration 1 begins when the query `sg(1,Y)` is called. Initial bookkeeping is done for the tabled subgoal: a table entry created, a frame placed on the completion stack and a choice point is set up. Then program clause resolution is used to derive the first answer `sg(1,1)`. Execution then backtracks to the second program clause for `sg/2` which eventually selects the subgoal `sg(3,Yp)`. The engine creates a choice point and table entry for `sg(3,Y)`, but will not use program clause resolution for `sg(3,Y)` until iteration 2. Rather, the engine suspends this subgoal and fails, causing it to execute the next available clause on the choice point stack, producing the selected subgoal `sg(4,Yp)`. Eventually, there will be no available clauses on the choice point stack and the engine will perform a *fixpoint check* (during the completion instruction), and then start iteration 2. The engine uses the completion stack to scan for either suspended subgoals to resolve using program clause resolution, or for consuming nodes with unresolved answers. For our example, iteration 2 begins by performing program clause resolution for `sg(3,Y)`. This produces, among other clauses, the answer `sg(3,3)`. This answer will be returned in iteration 3 to the consuming node `sg(1,Y):cons:sg(1,Y)`$\leftarrow$`[sg(3,Yp),par(Y,Yp)]`. During iteration 2, program clause resolution is also performed for `sg(4,Y)`, and its actions parallel those of `sg(3,Y)`. The process continues, with the engine scanning nodes via the completion stack at each completion instruction, and then either performing program clause resolution for the delta set of subgoals or performing answer resolution for the delta set of answers and consuming nodes. The engine terminates when neither new answers nor new subgoals exist.

We describe the scheduling steps for breadth-first search performed by the basic tabling operations presented in Section 2 [3].

Subgoal Call: In order to preserve search equivalence with magic, tabled subgoals may need to be batched until the next iteration (e.g. when Algorithm 3.1 iterates through $\delta Subgoals_{t-1}$) [4]. Any subsequent tabled subgoal is *suspended* when it is called, after its choice point, table entry, and completion stack frame are created. These suspended subgoals are reinvoked at the next iteration (through the completion instruction) when the engine exhausts all program and answer clauses from the previous iteration.

New Answer: In the strategy of the original SLG-WAM [12] bindings are shared between the calling environment of a subgoal and the root node of the tree for that subgoal (e.g., under tuple-at-a-time, in Figure 1, the variable Y_p in node 1 would be the same as the variable Y in node 2). In breadth-first scheduling however, this optimization is no longer applicable as it would violate magic equivalence by allowing answers to be returned in the same iteration they are created. So, in the

[3] We omit Program Clause Resolution since no scheduling takes place during this operation.

[4] An exception is the first tabled subgoal encountered in a breadth-first evaluation, called a *leader* or *root* node. Note that, in principle, the leader can be embedded in a larger evaluation.

breadth-first engine when a new answer is created, instead of executing the success continuation of the corresponding tabled subgoal (which would effectively return the answer to the calling environment), the engine simply adds the answer to the table and fails, executing the failure continuation at the top of the choice point stack.

Answer Resolution: Actual resolution of answers is done by the AnswerReturn instruction, which is invoked in one of two ways. AnswerReturn is executed whenever a new consuming node is created to return all answers from previous iterations (this case reflects line 16 of Algorithm 3.1 and line 10 of Algorithm 3.3). It is also invoked by the completion instruction to return the delta set of answers to consuming nodes present at the beginning of the iteration (reflecting line 15 of Algorithm 3.1). The Breadth-First SLG-WAM uses pointers into the answer list to maintain the delta answer set for each consuming node. For in memory queries, AnswerReturn backtracks through all the answers in the table and returns them one at a time to the proper consuming node. For queries to disk-resident data, an alternative mechanism is provided in the set-at-a-time database interface — this mechanism is described in the full version of the paper.

Completion: For the tuple-at-a-time SLG-WAM, the completion operation simply marks completely evaluated subgoals as completed and reclaims heap and stack space for these subgoals. In the breadth-first engine completion still marks subgoals as completed, but it also schedules resolution steps necessary for a new iteration. First, the engine has to perform program clause resolution for suspended subgoals (reflecting the actions on $\delta Subgoals_{t-1}$ in Algorithm 3.1). It must also schedule any unresolved answers as explained above. In fact, it is the completion check for the leader that controls the iterations for the breadth-first scheduling through the use of WAM-style failure continuations. The completion instruction for the leader will continue failing back to itself until a fixpoint is reached and the query is completely evaluated.

5 Performance Analysis

In previous papers [13, 9] the WAM-style tabling implementation of XSB v. 1.4 was shown to be about an order of magnitude faster than other deductive database systems for a variety of in-memory queries. Since many deductive databases, including XSB, are under continual development, the difference in speed may change over time; nevertheless the comparisons of [13] indicate the importance of compilation technology and of low-level engine optimizations.

Due to space limitations, we do not compare our new tabling strategies to other deductive databases; rather, we compare the new engine of Section 4 to the previous tuple-at-a-time engine of XSB, and for terminating queries, to Prolog itself. All benchmark programs were run on a SPARC 2 with 64MB of memory, and the engines considered in this section are:

- **XSB v. 1.4**: uses Single Stack Scheduling [4], the original depth-first (tuple-at-a-time) strategy for the SLG-WAM.
- **Breadth-First**: breadth-first (set-at-a-time) strategy described in Section 3.

It is worth pointing out that these two emulators differ *only* in the scheduling strategy. As will be shown, the set-at-a-time strategy has an advantage for evaluations which use disk, or for those involving aggregates or constraints in which

pruning can be useful. In this section we turn first to in-memory queries that do not benefit from pruning, since these may be regarded as a worst-case cost of a set-at-a-time evaluation.

Tests for In-memory Datalog Queries

For programs in which the complexity of Prolog's SLD strategy is the same as a bottom-up strategy (magic or tabling) it is generally found that Prolog is faster than the bottom-up method. However, for transitive closure, Example 5.1 derives times in which both tabling engines are almost as fast as SLD (run under XSB).

Example 5.1 To compare the efficiency of the right-recursive

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

using Prolog (SLD), against the left-recursive

```
:- table path/2.
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z),edge(Z,Y).
```

the query `path(X,Y)` was run on chains of varying lengths, as well as on complete binary trees of varying heights.

Table 1: *Normalized times for left-recursive transitive closure on linear chains and complete binary trees*

Emulator/Chain Length	1k	2k	4k	8k	16k
Breadth-First	1	1	1	1	1
XSB v. 1.4	0.891	0.9	0.846	0.88	0.88
SLD	0.718	0.63	0.64	0.66	0.657
Emulator/Tree Height	9	10	11	12	13
Breadth-First	1	1	1	1	1
XSB v. 1.4	1.154	1.073	1.041	1.096	1.122
SLD	0.891	0.81	0.807	0.877	0.947

Two points are worthy of note from Table 1. The first is that the breadth-first strategy is usually as fast as (and sometimes faster than) previously published XSB v. 1.4 times (for binary trees, where more answers are returned at each iteration, breadth-first is 10% *faster* on average than XSB v. 1.4, whereas for chains it is around 13% slower). The second is that the breadth-first strategy is roughly comparable to Prolog execution, indicating that at least for these examples, a disk-oriented set-at-a-time method is attainable at Prolog speeds.

For left-recursion on chains the breadth-first strategy has shown a consistent *slowdown* compared to XSB v. 1.4, and for trees, a slight *speed up*, what is expected given the well-defined structure of these graphs. Next we examine some different and ostensibly more realistic graphs. *Words* and subsets of it with fewer vertices and edges, and *Roget* were generated with Knuth's Stanford Graph Base [5]. *Genome* is a piece of a DNA sequence, while *Cylinder* is a 24x24 (2-connected) cylinder.

In these graphs, which can be considered as having a structure between chains and trees, the times to compute transitive closure for the breadth-first engine are about the same as for XSB v. 1.4, as shown in Table 2. Similar results are borne out

Table 2: *Normalized times for left-recursive transitive closure*

Engine/Graph	Words	Words3000	Roget	Genome	Same Gen.
Breadth-First	1	1	1	1	1
XSB v. 1.4	0.99	0.99	0.99	1.04	0.97

Table 3: *Normalized times for left-recursive transitive closure on linear chains and binary trees using the breadth-first engine and a variation using the consuming node optimization*

Emulator/Chains-length	1k	2k	4k	8k	16k
Breadth-First	1	1	1	1	1
Breadth-First with optimization	0.72	0.78	0.78	0.73	0.75
Emulator/Trees–height	9	10	11	12	13
Breadth-First	1	1	1	1	1
Breadth-First with optimization	0.79	0.81	0.83	0.77	0.79

in the same-generation program of Example 2.1, in which a bottom-up evaluation such as tabling can show an arbitrary speedup over Prolog. Run on a 24x24 cylinder, the times for the different engines are comparable, as can be seen in Table 2.

A source of overhead for Breadth-First comes from the fact that the tuple-at-a-time engine makes use of an optimization not available to the breadth-first strategy. Recall that when calling a new subgoal S, a consuming node is created with selected literal S and a new tree is also created with root S. The tuple-at-a-time emulator shares variable bindings between the generator node for S and its parent, but in a breadth-first engine this optimization would allow the return of answers through the generator node before the proper iteration. In order to measure the cost of this extra consuming node, we created a variant of the breadth-first engine that uses the tuple-at-a-time consuming node optimization. (Such an engine is not breadth-first and is used only for performance analysis). Notice in Table 3 that the added overhead for the extra consuming node on linear chains is between 30% and 40%, whereas for binary trees this overhead varies from 20% to 30%. These tables indicate that the absence of the tuple-at-a-time consuming node optimization accounts for most of the overhead of Breadth-First for chains.

An important aspect to take into account is memory usage, since it is well-known that high space utilization is a drawback of breadth-first strategies. For programs that do not have subgoal suspensions (e.g., left-recursive transitive closure), Breadth-First shows an improvement over XSB v. 1.4. This improvement is derived from the fact that, by *batching* answers (and not returning them eagerly as XSB v. 1.4), the breadth-first engine reduces the amount of movement in the search space, consequently decreasing the need to freeze branches and decreasing the number of *trapped* nodes in the stacks (more details can be found in [4]). However, if there are suspensions (e.g., right-recursive transitive closure), Breadth-First can use a much larger amount of memory, since suspended subgoals usually lead to a bigger number of consuming nodes. In the tuple-at-a-time engine, subgoals are evaluated at the time they are called, therefore, if later a variant of it is called, there is a better chance that this subgoal is completed, in which case the creation of a new consuming node can be avoided, as answers can be used as facts. Table 4 shows the sum of the maximum stack sizes (in bytes) for left and right-recursive transitive closure on different graphs: a 1024-long chain, a complete binary tree of height 9 and on a piece of a DNA sequence. Notice that the space utilization for Breadth-First is significantly lower than for XSB v. 1.4 for left recursion, whereas for right recursion it is considerably higher.

Comparisons for Optimization Tasks

In the previous discussion we considered problems that can be regarded as worst cases for a breadth-first search. Now we turn to some examples where breadth-

Table 4: *Memory utilization for left and right-recursive transitive closure*

Engine	Left-Recursion		Right-Recursion	
Graph	Breadth-First	XSB v. 1.4	Breadth-First	XSB v. 1.4
Chain 1k	2416	18,821,404	202,072	116,280
Tree 9	2416	473,728	566,020	13,188
Genome	2500	218,486	521,364	17,704

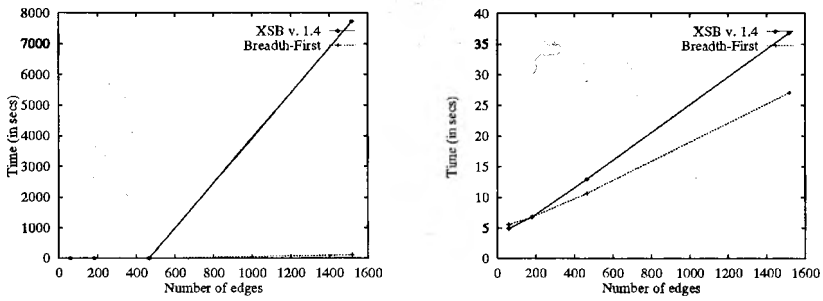
first is expected to perform well. Finding the shortest path of a graph is one such example, and we make use of the shortest path program described in Example 5.2, where `bagMin/2` is an aggregate predicate[5] that maintains the length of the shortest-path between two nodes (all non-minimal answers are deleted).

Example 5.2 Our shortest-path program is defined as follows:

```
sp(X,Y)(D) :- arc(X,Y,D).
sp(X,Y)(D) :- bagMin(sp(X,Z),D1), arc(Z,Y,D2), D is D1+D2.
```

We considered subsets of the Stanford Graph Base Words graph, containing common 5-letter words[6]. Figure 2(a) shows the times for 500 runs of the query `sp(words,X)(D)`, to find the shortest path between "words" and all the other words reachable from it. Note that for the small graphs, with up to around 500 edges, the two engines spend about the same time. For larger graphs however, Breadth-First is considerably faster then XSB v. 1.4.

Figure 2(b) shows the times for 1000 runs of the query, `sp(there,white)(D)`, where the distance between the two words is fixed (`D=7`) for all graphs under consideration. For this case, Breadth-First also has better performance, even though the differences in times are not as striking. For example, to find the shortest-path (of length 24) between "words" and "spots" in Words1000, XSB v. 1.4 takes 15.67 seconds, whereas Breadth-First takes just 0.34 seconds.



(a) (b)

Figure 2: *(a) shows the CPU time to compute* `sp(words,X)(D)` *500 times, and (b) shows the CPU time to compute* `sp(there,white)(D)` *1000 times*

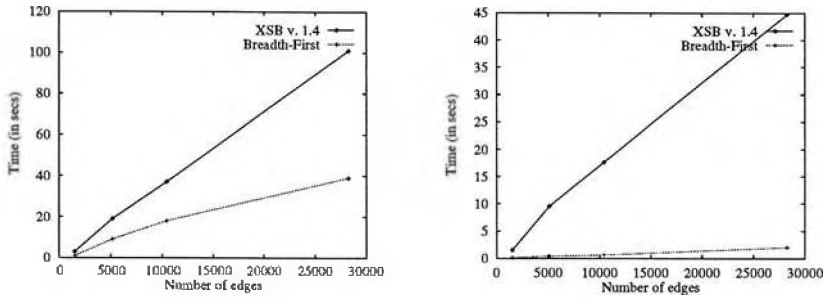
[5] It is worth pointing out that XSB provides an efficient implementation aggregates using HiLog [2] syntax. For more information on these aggregate predicates, consult the XSB Manual (available at *http://www.cs.sunysb.edu/~sbprolog/manual/manual.html*).

[6] It was not possible to run this program with XSB v. 1.4 for bigger graphs (with more than 1000 words), due to memory limitations.

Accessing External Relations

The main goal of the Breadth-first engine is to efficiently access relations stored in an external database, and in this section, we compare the performance of the breadth-first engine using a set-at-a time interface[7] against the XSB v. 1.4 engine using a tuple-at-a-time interface currently distributed with the XSB engine.

In order to compare the evaluations for left-recursive transitive closure, subsets of the Words graph were stored in Oracle as tables indexed on the first column.



(a) Elapsed time

(b) XSB CPU time

Figure 3: *Times to find all words reachable from "words" in Words1000, Words2000, Words3000 and Words indexed on the first argument of the tables*

The times to evaluate the query `reach(words,X)` are shown in Figure 3. We considered both the XSB CPU time, the time spent by the XSB engine, and the elapsed time, the total time used to evaluate the query (including I/O). As expected, the graphs in Figure 3 show that the set-at-a-time processing of Breadth-First is considerably faster than the tuple-at-a-time processing of XSB v. 1.4. It is worth pointing out that at the present time the set-at-a-time database interface is in an initial stage of its development, and many optimizations have not yet been implemented (e.g., reducing the number of times dynamic SQL queries are parsed).

6 Conclusion and Future Work

The equivalence results of Section 3 indicate that, on an iteration by iteration basis, breadth-first tabling and Semi-Naive Magic Template evaluation are the same. The performance results of Section 5 amply bear this out: the tabling engine gives excellent times for disk-resident data without sacrificing in-memory performance. The disk access times also indicate that the internal data representation of the engine meshes well with that needed by an SQL database such as Oracle.

Thus, serious database interaction, such as that needed in data mining, decision support, or in a variety of other applications, can be done within the structure of first-order logic and the programming environment of Prolog. Of course for practical database systems, queries can be restricted to a decidable subset of first-order logic such as Datalog, or query forms can be restricted to those provably terminating. Such a system can be created by fully integrating the breadth-first engine into the programming environment of XSB. This involves such issues as: (1) The extension

[7] For more details about the implementation of the set-at-a-time database interface the reader is referred to the full version of this paper.

of the breadth-first formalism and engine to evaluate negation according to the well-founded semantics as is done by the tuple-at-a-time tabling in XSB; (2) Refinement of the engine to integrate different search strategies within the same evaluation; (3) Investigating automatic abolishing of tables after an evaluation is finished with them.

When these 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 the SLG-WAM. We believe this framework will form a computational basis to combine the fields of logic programming and deductive databases.

Acknowledgements: Prasad Rao implemented the low level tabling predicates to suppot aggregate computation and Hasan Davulcu implemented the tuple-at-a-time XSB-Oracle interface. We would like to thank Raghu Ramakrishnan for originally suggesting the idea to us. This work was supported in part by CAPES-Brazil, and NSF grants CDA-9303181 and CCR-9404921.

References

[1] H. Aït-Kaci. *WAM: A Tutorial Reconstruction.* MIT Press, 1991.

[2] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

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

[4] J. Freire, T. Swift, and D.S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *8th International Symposium PLILP*, pages 243–258. Springer-Verlag, 1996.

[5] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison Wesley, 1993.

[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 *Proceedings of the 18th 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] S. Sudarshan. *Optimizing Bottom-up Query Evaluation for Deductive Databases.* PhD thesis, University of Wisconsin, 1992.

[12] T. Swift and D. S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. In *Proceedings ILPS*, pages 633–654, 1994.

[13] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of ILPS*, pages 219–238, 1994.

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

[15] 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.

[16] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.