Efficient Tabling Mechanisms for Logic Programs

I.V. Ramakrishnan
Prasad Rao
Konstantinos Sagonas
Terrance Swift
David S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{ram,prasad,kostis,tswift,warren}@cs.sunysb.edu

Abstract

The use of tabling in logic programming allows bottom-up evaluation to be incorporated in a top-down framework, combining advantages of both. At the engine level, tabling also introduces complications not present in pure top-down evaluation, due to the need for subgoals and answers to access tables during clause resolution. This paper describes the design, implementation, and experimental evaluation of data structures and algorithms for high-performance table access.

Our approach uses tries as the basis for tables. Tries provide complete discrimination for terms, and permit a lookup and possible insertion to be performed in a single pass through a term. A novel technique of *substitution factoring* is also proposed. When substitution factoring is used in conjunction with tries, the access cost for answers is proportional to the size of the answer substitution, rather than to the size of the answer itself. As a special case, the access cost of answers to a datalog query is proportional to the number of non-ground arguments of the query, giving a means of dynamic argument reduction. Answer tries can be implemented both as interpreted structures and as compiled WAM-like code. When they are compiled, the speed of computing substitutions through answer tries is competitive with the speed of unit facts compiled as WAM code, or asserted. Because answer tries can also be created an order of magnitude more quickly than asserted code, they form a promising alternative for representing dynamic terms, even in SLD evaluation.

1 Introduction

Bottom-up evaluation of logic programs offers well-known advantages over top-down: programs terminate if they have finite models; redundant subcomputations are eliminated; and non-stratified programs can be evaluated without using the possibly exponential number of negative contexts that top-down evaluation requires. Of course *direct* bottom-up evaluation is unacceptable for query evaluation, since it evaluates all possible queries to a program. As a result, a persistent theme of logic programming has been to investigate how to combine the advantages of bottom-up evaluation with the goal-orientation of top-down techniques. This effort has led to many systems based on magic evaluation and related strategies (see [11] for a survey of research on deductive database systems).

The raw speed of top-down engines, though is sometimes neglected in the literature. At least for loop-free, stratified programs with few redundant subcomputations, top-down engines can be substantially faster than bottom-up. Thus, rather than adding goal-orientation to a bottom-up engine, a natural approach to evaluating in-memory queries is to add bottom-up capabilities, or tabling, to a Prolog engine. The XSB system [14] follows this latter approach. The goal of XSB is to evaluate *tabled* predicates (using SLG evaluation [3]) in approximately the same time as *non-tabled* predicates (using SLD). Based on our experience, it appears that the greatest efficiency gains under present technology can be made at the level of engine design. This paper reports on our recent experiences with engine enhancements for tabling that yield substantial performance improvements. Specifically, we present results of experiments for reducing the time for an engine to access tabled information and more generally to access dynamically created terms (*e.g.,* asserted facts).

Consider the modalities of table access for SLG evaluation of definite programs:

- *Call Check/Insert.* When a tabled goal is called, a check must be made to see whether the goal is redundant or not. In the case of XSB, this amounts to a *variant check* of whether the new goal is a variant of one that exists in the table.If it is, the goal is termed *active* and answer clauses are resolved against it. If not, the goal is termed *generating*, entered into the table, and program clause resolution is used instead. We associate with each tabled goal a set of answers.

- *Answer Check/Insert.* When an answer is derived for a particular goal, a check is made to determine whether it has already been entered into the table for the goal. If it has, the derivation path fails, a vital step for ensuring termination. If not, the computation continues, and the answer is scheduled for return to the applicable active goals.

- *Answer Backtracking.* When an active goal is created, it backtracks through answers in the table in the course of its evaluation.

These table access modalities, are not essentially different from those of other tuple-at-a-time tabling methods. Observe that naive table lookups and inserts can result in repeatedly rescanning terms and thereby may degrade performance considerably. For in-memory computations, the achievement of the goal of Prolog speed for tabled programs is only possible if the above three operations are performed with very little overhead. Specifically, in the case of the call check/insert step, a call to a tabled predicate must take nearly the same time as a call to a non-tabled predicate. Next, the time of answer check/insert should be small relative to the time required to derive an answer, since this operation occurs for each solution to a tabled predicate. And finally, backtracking through answer clauses must take roughly the same time as backtracking through unit program clauses. Needless to add, engine modifications to enable efficient storage and retrieval of subgoals and answers in tables cannot compromise the performance of the system for any class of problems.

While compile-time approaches can partially alleviate these problems — for instance, such approaches can indicate which predicates should be tabled and which should not — their ultimate solution must be dynamic. We may thus speak of the *Table Access Problem* as one of designing efficient algorithms and data structures for accessing tabled data.

Our results regarding the *Table Access Problem* are as follows. Firstly, we devise a trie-based method for storing subgoals and their returns in tables. Tries eliminate repeated rescanning of tabled terms during lookups and inserts. Secondly, using tries in conjunction with *substitution factoring*, a technique developed in this paper, reduces the overheads in table lookup and insert operations even further. Thirdly, we devise a technique for dynamically compiling tries, leading to the ability to backtrack through answer clauses at speeds comparable to compiled WAM code. In fact, our technique for dynamically compiling tries is generally applicable for accessing other dynamically created terms such as asserted facts. As a final result, we demonstrate this generality by exhibiting significant speedups over existing methods, when processing asserted facts.

The rest of this paper is organized as follows. The next section describes trie-based methods for storing subgoal and answer tables. In Section 3 we develop the concept of *substitution factoring* for realizing table lookup and insert operations with low overhead. Implementation aspects of trie-driven tabling are discussed in Section 4. The technique of dynamically compiling tries is described in Section 5. In Section 6 we present performance results which provide strong evidence that our techniques can indeed facilitate tabled logic programs to attain speeds comparable to Prolog programs. We conclude with a discussion of the relevance of this work to in-memory query optimization techniques. Because of space limitations, we assume knowledge of the WAM [18].

Trie-based tabling, substitution factoring and compiled tries are all implemented in Version 1.4.2 of XSB. XSB has been installed in hundreds of sites for educational, research and commercial use, and runs under DOS and most UNIX platforms.

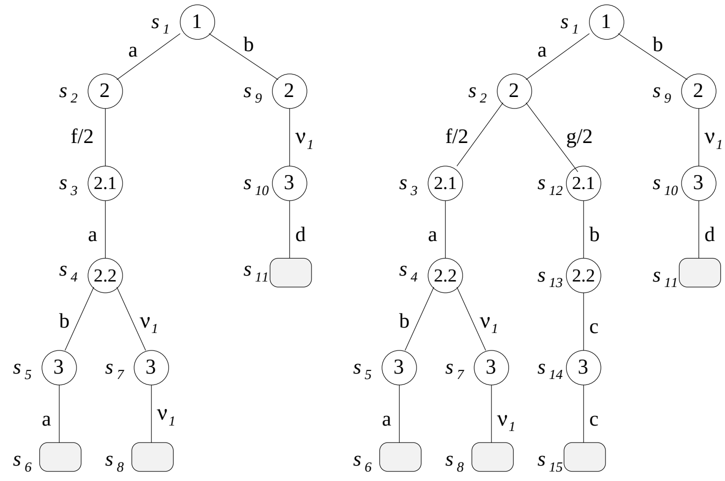
2 Tabling Tries

We assume the standard definitions of term and the notions of substitution and subsumption of terms. A *position* in a term is either the empty string Λ that reaches the root of the term, or $p.i$, where p is a position and i is an integer, that reaches the i^{th} child of the term reached by p. The symbol t (possibly subscripted) denotes terms; f, g denote function symbols; and all capital letters (possibly subscripted) denote variables. We use the terms *call* and *subgoal* interchangeably, as well as the terms *answer* and *return*.

The trie data structure was originally invented to index dictionaries [7] and has since been generalized (as discrimination nets) to index terms (see [2] for use of tries in indexing logic programs and [1, 8, 10, 17] for automated theorem proving and term rewriting). We will use a variant of the discrimination net in [1] as the data structure for tabling calls and returns. We refer to it as the *tabling trie*.

The essential idea underlying a tabling trie is to partition a set T of terms based upon their structure. It is used for both looking up and inserting an input term into the table. It is a tree-structured automaton with the root as the start state and each leaf state is associated with a term in T. Each state specifies the position to be inspected in the input term on reaching that state. The outgoing transitions specify the function symbols expected at that position. A transition is taken if the symbol in the input term at that position matches the symbol on the transition. On reaching a leaf state we say that the input term *matches* the term associated with the leaf state. The root-to-leaf path taken to reach the leaf state corresponds to a preorder traversal of the matching term. When no outgoing transition from a state can be taken, a lookup operation fails. On the other hand, for an insert operation we add an outgoing transition for the symbol and a new destination state for this transition. The position that will be associated with the new state is the next position in the preorder traversal of the input term. We illustrate the operations on a tabling trie using the example in Figure 1.

```
rt(a,f(a,b),a).
rt(a,f(a,V),V).
rt(b,V,d).
```



(a) (b) (c)

Figure 1: (b) is a trie for the terms in (a).

To look up the term `rt(a,f(a,b),a)` we begin at state s_1. Since position 1 in the input term is `a`, we make a transition to state s_2. In this state we inspect the next preorder position of the input (position 2) and make a transition to state s_3. Transition from this state on seeing `a` (in position 2.1) leads to the state s_4. Continuing thus we finally reach the leaf state s_6 and declare a match. To insert the term `rt(a,g(b,c),c)` we again start at state s_1 and make a transition to state s_2. Since there is no outgoing transition labeled `g/1` we create a new transition for `g/1` and a new destination state for this transition. In this new state we will inspect position 2.1 which is next in preorder traversal of the input term. Continuing in this fashion we will create three more new states yielding the trie shown in Figure 1(c).

Our point of departure from the trie in [1] is in our treatment of variables. Recall that our lookup/insert operations perform a variant check, i.e. two terms match if they are variants of each other. To realize variant checks in our tabling trie we *standardize* the representation of a term as follows. We treat each variable as a distinct constant. Formally, we define a bijection $numbervar_t$ from the set of variables in t, denoted by $vars(t)$, to the sequence $\langle \nu_1, \nu_2, \ldots, \nu_n \rangle$ such that $numbervar_t(V) < numbervar_t(W)$ if in the preorder traversal of t, V is encountered before the first occurrence of W. For example in $f(g(Y,Z),X,Z)$, $numbervar_t(X)$, $numbervar_t(Y)$ and $numbervar_t(Z)$ are ν_3, ν_1 and ν_2 respectively. Let $numbervar(t)$ denote the the term $t\theta$ where $\theta(V){=}numbervar_t(V)$ for all V in $vars(t)$. Thus $numbervar(f(g(Y,Z),X,Z))$ is $f(g(\nu_1,\nu_2),\nu_3,\nu_2)$. Two terms are variants of each other iff $numbervar(t_1) = numbervar(t_2)$.

Converting a term to standard form can be done concurrently with the process of lookup and inserts. Details are in Section 4. From the description above it is easy to see that:

Proposition 2.1 For the *subgoal* and *answer check/insert* steps, each element of the input term is examined only once.

In summary, trie-based tabling has two major advantages over hash-based tabling.

1. *Complete discrimination.* Tries completely discriminate between terms no matter where in a term the discriminating element lies. In contrast, hashing based on a limited prefix of the term, suffers if the discriminating element is deeply nested. Furthermore, the trie-based approach can actually reduce space because of greater sharing between terms.

2. *Single pass check/insert.* For the subgoal and answer check/insert, a single traversal of the term is necessary, regardless of whether the term needs to be copied into the table. In hash-based tabling lookup alone may require multiple passes over the term due to hash collisions. Furthermore, insertion will require a separate pass. Given the prevalence of these operations in a tabling system, the savings in time over a two-pass operation can be substantial.

The above advantages are evident in the performance results presented in Section 6. Although we have optimized each lookup and insert operation in the call and answer tables, by treating the two tables as independent entities, we have not exploited sharing between a specific call and its answers. We can further enhance the performance of tabling operations by exploiting this sharing using the concept of *substitution factoring* described in the following section.

3 Substitution Factoring

Recall that we associate an answer table with every subgoal in the call table. Given a subgoal G, all answers A for the subgoal are subsumed by G, and can be represented as $G\theta_A$. We call θ_A an *answer substitution* for G. Note that the sum of the sizes of terms in θ_A is less than the size of $G\theta_A$. The core idea of substitution factoring is to store only the answer substitutions, and to create a mechanism of returning answers to active subgoals that takes time linear in the size of θ_A rather than the size of $G\theta_A$. In other words, substitution factoring ensures that answer tables contain no information that also exists in their associated call table. Operationally this means that the constant symbols in the subgoal need not be examined again during either answer check/insert or answer backtracking.

Let G denote a subgoal and $V_1, V_2, \ldots, V_n$ denote the variables in G such that $numbervar(V_i) = \nu_i$. An answer substitution θ_A for G is of the form $\{V_1 \leftarrow t_1, V_2 \leftarrow t_2, \ldots, V_n \leftarrow t_n\}$. We call the sequence $\langle t_1, t_2, \ldots, t_n\rangle$ as the *answer tuple* corresponding to the answer substitution θ_A. Observe that we can reconstruct the answers given the subgoal, the variable sequence $\langle V_1, V_2, \ldots, V_n\rangle$ and the answer tuples. Hence if we store the variable sequence with the subgoal, then we need only store the answer tuples in the answer table. In particular, the answer is stored in the answer trie as a sequence $\langle t_1, t_2, \ldots, t_n\rangle$. The variable sequence is determined when the subgoal is standardized for insertion into the call table. Thus the storage requirement for an answer $G\theta_A$ corresponding to the subgoal G is linear in the size of θ_A. Furthermore implementing answer tables as tries ensures that:

Proposition 3.1 Let G be a subgoal and A be an answer for G. Using substitution factoring both answer check/insert and answer backtracking can be performed in time proportional to the size of the answer substitution of A.

To illustrate this, consider a subgoal `p(f(X,Y),g(X))` with an answer `p(f(a,b), g(a))`. In this case the answer has six symbols, whereas the substitution θ has only two symbols. For an access operation on an answer, either check/insert or return, using substitution factoring only two symbols are traversed as opposed to six.

In terms of related work, substitution factoring bears a certain resemblance to the factoring of [9] (hereafter termed *NRSU-factoring*) in that both reduce the number of arguments copied into or out of a table. However, substitution factoring has different characteristics than NRSU-factoring, mainly because it is a dynamic rather than static technique. Whether a predicate is NRSU-factorable is undecidable in general; hence NRSU-factoring is applicable only to certain classes of Datalog programs. Consequently, substitution factoring may reduce arguments of predicates that are not reduced by NRSU-factoring. Furthermore, contrary to NRSU-factoring, substitution factoring is applicable to and can be very effective for non-Datalog programs. On the other hand, [9] introduces additional optimizations based on the factored program which are not performed by substitution factoring. These optimizations can transform certain right and double recursions into left recursions, an important transformation not performed by substitution factoring.

4 Implementation Aspects of Tabling Tries

Tabling tries are implemented by representing each state by a node, and transitions by pointers to nodes. The structures of subgoal and answer trie nodes are shown in Figure 2 and are explained throughout this section. The label on a transition is

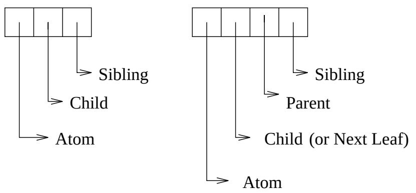


Figure 2: Data structures of subgoal and answer trie nodes.

placed in the *atom field* of the node representing the destination state. The outgoing transitions from a node are traced using its *child pointer* and by following the list of *sibling pointers* of this child. Recall that in order to lookup or insert a term into a tabling trie, the term is traversed in pre-order. If the symbol inspected in this traversal is the label of an outgoing transition from the current state, that transition is taken. Otherwise, a new destination state is created, and the transition to this state is taken. The matching outgoing transition is found using sequential search whenever the number of outgoing transitions from this state is small, otherwise hashing is used to select the matching transition. Hash collisions are reduced by dynamically expanding the hash tables.

Recall that terms inserted in the trie are *standardized.* This standardization process is performed while a term is inserted in the trie. The variables in the term are replaced by their *numbervar* values, by binding the variable cell to a unique number, and tagging the cell with an unused type tag. This *single* binding for each variable effectively replaces all occurrences of the variable with its *numbervar* value. Thus, nonlinearity is handled without duplicate checks. The bindings are undone as soon as the insertion of the term in the trie is complete. Observe that this operation realizes the *numbervar* bijection in a single pass of the input term.

4.1 Implementation of Substitution Factoring

In order to explain the implementation of substitution factoring, we first consider the creation of an (SLG-)WAM choice point for a tabled predicate. The WAM creates a choice point by copying the program registers at the time of the call, including registers containing each argument of the subgoal. If the subgoal is non-redundant, a (*generator*) choice point is created which will backtrack through program clauses. However, instead of *arguments* of the subgoal, only its *variables* are copied into the choice point. The set of variables in the subgoal is obtained during the traversal of the subgoal. If the subgoal is redundant, program clause resolution is not used. Instead, an *active subgoal frame* is created, which serves as an environment into which answers can be returned by means of an *answer-return* operation. Like the generator choice points, these active subgoal frames contain only the variables of the subgoal. In either case, bindings to the dereferenced values of these variables are trailed through forward execution, whether it is program clause or answer clause resolution. The values are untrailed through backtracking just as argument cells would be in WAM execution. When an answer to a tabled subgoal is detected, the dereferenced values of the variable cells are copied from the generator choice point directly into the table, and are loaded directly into the active subgoal frames thus returning the answers. Figure 3 shows an example of a tabling trie incorporating substitution factoring for answers to the subgoal `p(f(X),g(Y))`.

Subgoals:

```
p(X,X)
p(f(X),a)
p(f(X),g(Y))
p(f(X),g(X))
```

Answer substitutions for `p(f(X),g(Y))`:

```
X = a, Y = a
X = a, Y = b
X = b, Y = a
X = b, Y = b
```

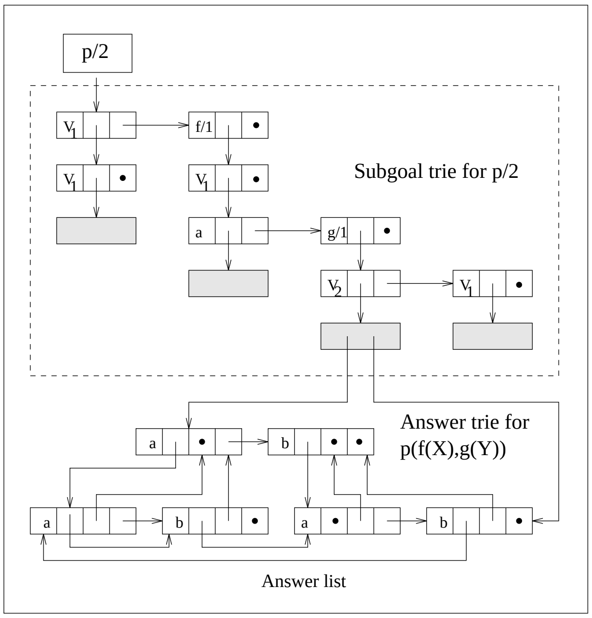


Figure 3: Substitution factoring illustrated for the subgoal `p(f(X),g(Y))`.

4.2 Returning Answers to Active Subgoals

Recall that in a tabling framework answers need to be returned to applicable *active* subgoals. Answer tries of subgoals for which new answers may be derived are termed *incomplete* (see [16]). Since answer insert and answer return operations can be interleaved, and new answers can be inserted *anywhere* in the trie, it is not possible to perform the *answer-return* operation by sequentially backtracking through such a trie starting from its root. Therefore, an explicit list of returns (uniquely identified by leaf nodes of the answer trie), has to be maintained. The list is implemented by having the *child field* of leaf answer nodes point to the next return. The order of this list reflects the creation times of its members. For example, in Figure 3 the answers are created in the order $\{X = b, Y = b\}$, $\{X = a, Y = a\}$, $\{X = a, Y = b\}$, and $\{X = b, Y = a\}$. Answers are returned for incomplete answer tries by traversing this list, constructing the answers by a leaf-to-root traversal. To efficiently perform this latter traversal, every node of the return trie maintains a back pointer to its parent node (denoted as *parent pointer* in Figure 2).

Note, however, that answers can be returned from a *completed* answer trie by sequentially backtracking from its root. The WAM is a highly optimized engine for performing backtracking. To exploit this power of the WAM, we dynamically compile answer tries into WAM code as presented in the next section. The idea of compiling dynamically created terms has been around for quite some time in logic programming languages; for example it is used in some implementations of `assert/1`. Recently, this idea has also been used in the context of general theorem proving to efficiently perform forward subsumption (i.e. pattern matching) of terms that are dynamically created [17].

5 Dynamic Compilation of Tries

We describe how answer tries are dynamically compiled into WAM-like instructions, called *trie instructions.* The trie instructions operate in the same manner as compiled program clauses, constructing their information directly onto the local stack or heap. We refer to the tries that consist of these instructions as *compiled tries*, and to those described in Section 4.2 as *interpreted tries*.

To motivate the new WAM instructions, we first show how a return trie can be represented as regular Prolog clauses. We then consider how WAM-style instructions might implement those clauses, and finally we create "mega"-instructions that constitute an space efficient representation of return tries. We use the following example throughout the development. Consider the return trie of a subgoal that contains three variables shown in Figure 4(b). The four answers in this trie could be represented by the Prolog facts of Figure 4(a).[1] Alternatively these facts could be prefix-factored into a *full-trie* [4] represented as a set of Prolog clauses shown in Figure 4(c). Note that each clause in figure 4(c) corresponds to a single edge in the return trie of Figure 4(b), and that the order of the clauses reflects a breadth-first, left-to-right traversal of the edges of the trie.

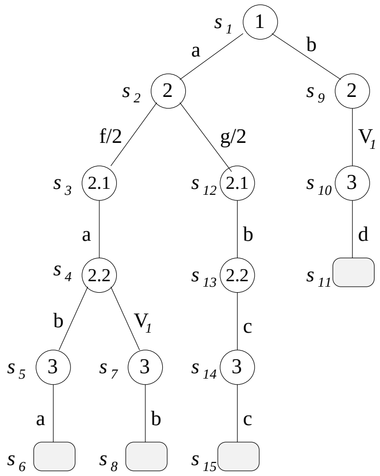
We base the following discussion on two premises of Section 4. Firstly, we assume that backtracking is only performed on completed tables, so that no answers will be added to a trie through which we are backtracking. Next we assume that variables have been gathered out of a subgoal as in substitution factoring. Operationally this means that when answer clause resolution is to be used for a subgoal with n variables, the first n registers have been initialized to hold these variables.

Figure 5 shows WAM-like code segments for the first three clauses of Figure 4(c). The code for the first two clauses starts with a choice-point instruction, here a try_me_else or a trust_me_else instruction. The second instruction is a get_*type* instruction. The shift-left instruction is not contained in the WAM. Its function is

[1] For simplicity of presentation we consider linear answers first, and describe later on how non-linearity is handled.

```
rt(a,f(a,b),a).
rt(a,f(a,V),b).
rt(a,g(b,c),c).
rt(b,V,d).
```

(a)



(b)

```
rt(a,X2,X3) :- rt_a(X2,X3).
rt(b,X2,X3) :- rt_b(X2,X3).
rt_a(f(X1.1,X1.2),X2) :-
        rt_af(X1.1,X1.2,X2).
rt_a(g(X1.1,X1.2),X2) :-
        rt_ag(X1.1,X1.2,X2).
rt_b(V,X2) :- rt_bV(X2).
rt_af(a,X2,X3) :-
        rt_afa(X2,X3).
rt_ag(b,X2,X3) :-
        rt_agb(Y,X3).
rt_bV(d).
rt_afa(b,X2) :- rt_afab(X2).
rt_afa(V,X2) :- rt_afaV(X2).
rt_agb(c,X2) :- rt_agbc(X2).
rt_afab(a).
rt_afaV(b).
rt_agbc(c).
```

(c)

Figure 4: A return trie, (b), and two Prolog representations of its returns, (a), (c).

to shift all the registers to the left by some number of positions (here one). This function is needed to set up the arguments for the final instruction, the execute, which branches to the next clause. Now consider the code for the third clause, whose first argument is a structure. Here again the first instruction is a choice-point instruction, and the second instruction is a get_structure instruction. However get_structure is followed by a shift_right instruction which shifts the registers right to make room for the arguments of the structure symbol; (the required number of positions is always one less than the arity of the structure symbol). The shift_right instruction is followed by an argument-construction instruction, unify_variable for each argument. Finally, there is again an execute instruction to branch to the next clause.

rt_1:	try_me_else	rt_2	rt_2:	trust_me_else	fail	rt_a_1:	try_me_else	rt_a_2
	get_constant	a, A_1		get_constant	b, A_1		get_structure	f/2, A_1
	shift_left	1		shift_left	1		shift_right	1
	execute	rt_a/2		execute	rt_b/2		unify_variable	V_1
							unify_variable	V_2
							execute	rt_af/3

Figure 5: Possible WAM-like code segments for the first three clauses of Figure 4(c).

Code segments like the ones in Figure 5 will construct the returns one-at-a-time on the stack through backtracking. For efficient implementation we coalesce these sequences of instructions into *trie instructions*. Note that there are five major parameters to a code segment: (1) the *choice* alternative; (2) the get_*type* alternative; (3) the constant, structure symbol, or variable to match (*Symbol*);[2] (4) the address of the next code segment down the trie (*ContLabel*); and, (5) the address of the alternative to try on failure (*FailLabel*). So, the general form of the trie

[2]Note that the argument register involved in the get_*type* instructions is always register 1 (A_1).

instructions is:

$$\text{trie\_}\mathit{choice\_type} \quad Symbol, \quad ContLabel, \quad FailLabel$$

Since each of the trie instructions may appear as the first, an intermediate, the last, or the only instruction in a sequence of alternatives, we denote the *choice* possibilities as try, retry, trust, and do respectively. As for *types*, constants, structures, lists as well as, uninstantiated and instantiated variables should be handled. Figure 6 presents the set of the trie instructions thus created. In the special case of answer

	Unique	*First*	*Intermediate*	*Last*
Constant	trie_do_constant	trie_try_constant	trie_retry_constant	trie_trust_constant
Structure	trie_do_structure	trie_try_structure	trie_retry_structure	trie_trust_structure
List	trie_do_list	trie_try_list	trie_retry_list	trie_trust_list
Variable	trie_do_variable	trie_try_variable	trie_retry_variable	trie_trust_variable
Variable	trie_do_value	trie_try_value	trie_retry_value	trie_trust_value

Figure 6: Trie instructions for all possible *choice* and *type* combinations.

tries that have no variables, another trie instruction needs to be introduced, named trie_proceed which has exactly the functionality of the WAM's proceed. Note that functionality of a proceed is also needed after the leaf of a trie is reached.

As a concrete example, Figure 7 shows the WAM code generated for the clauses of Figure 4(c), and, as a consequence, for the answer trie of Figure 4(b). Since no choice points are laid down for the trie_do_? instructions, their *FailLabel* fields

rt_1:	trie_try_constant	a,	rt_a_1,	rt_2	rt_bV:	trie_do_constant	d,	*proceed*	
rt_2:	trie_trust_constant	b,	rt_b,	fail	rt_afa_1:	trie_try_constant	b,	rt_afab,	rt_afa_2
rt_a_1:	trie_try_structure	f/2,	rt_af,	rt_a_2	rt_afa_2:	trie_trust_variable	V,	rt_afaV,	fail
rt_a_2:	trie_trust_structure	g/2,	rt_ag,	fail	rt_agb:	trie_do_constant	c,	rt_agbc	
rt_b:	trie_do_variable	V,	rt_bV		rt_afab:	trie_do_constant	a,	*proceed*	
rt_af:	trie_do_constant	a,	rt_afa_1		rt_afaV:	trie_do_constant	b,	*proceed*	
rt_ag:	trie_do_constant	b,	rt_agb		rt_agbc:	trie_do_constant	c,	*proceed*	

Figure 7: WAM code for the clauses of fig. 4(c).

are not used. Notice the correspondence between the labels of the instructions in Figure 7 and the names of the atoms in the clauses of Figure 4(b). For facts, the trie instruction has a *ContLabel* of *proceed* to indicate that the final operation of the trie instruction should be a proceed rather than an execute.[3]

Due to space limitations, we do not present pseudocode that realizes the trie WAM instructions. However, their operational semantics should be evident if one thinks of them as macros that define WAM code segments like those of Figure 5. For example, the three code segments of Figure 5 present the operations performed (for some values of the parameters) by the trie_try_constant, trie_trust_constant, and trie_try_structure instructions, respectively. We note that the shift_left and shift_right operations could be implemented efficiently in an engine that stores the registers as an array, simply by modifying the base of that array. Alternatively, a separate array of pseudo-registers could be used for the trie instructions only, which would

[3] A slight optimization would be to create specialized versions of trie instructions that encode the last operation as a *proceed*. Such instructions would be needed only for constants and variables.

allow it to perform efficiently as a *register stack*. The latter is the implementation scheme chosen by XSB. Non-linearity is handled by adding another array to the WAM, called the *var-array*. The trie_?_variable instructions initialize the indicated var-array entry on the heap, setting the top element of the register stack to point to it. The trie_?_value instructions then unify the top register of the register stack with the indicated var-array variable.

The tabling framework described performs variant checks for subgoals, so that substitution factoring allows all uses of the get_*type* subinstructions in the return trie to actually *build* their values, i.e., to run in write mode. We also note that there is the same need for indexing in compiled tries as in interpreted tries. Accordingly, the set of trie instructions described in this section has been extended with two more hashing instructions to perform this indexing. While useful for the check/insert step, when a variant subgoal check is used the hashes do not provide any extra efficiency once a subgoal is completed because backtracking visits all answers.

Finally, observe that the trie instructions are immediately applicable for dynamic facts. In particular, they can be viewed as an alternative instruction set that can be used for the compilation of asserted facts (when their order need not be preserved). Naturally, the full unification power of the get_*type* instructions would be needed in such a case. Section 6.2 provides experimental demonstration of the performance gains of using this instruction set to access and create asserted facts, over existing compilation methods for dynamic code.

6 Performance Results

The following sections evaluate various methods for creating and accessing dynamic terms. We first compare alternative tabling methods used in XSB: a hash-based method of Version 1.4.0, and two different trie-based methods. The first trie-based method, of Version 1.4.0, does not compile tries into instructions, while the second, that of Version 1.4.2, does. After testing alternative tabling methods, we measure the benefit of trie compilation by testing creation and access times for completed tables against asserted clauses in XSB and Quintus, Release 3.1.1.[4]

6.1 Trie-based vs. Hash-based Table Structures

Hash-based table structures have a simple form. Each tabled predicate has its own subgoal hash table. For the subgoal check/insert step, the subgoal is hashed and compared against any other subgoal in the hash bucket, using a variant-check. If the subgoal is not present, it is entered into the chain of the proper hash bucket. Each subgoal has its own answer hash table which resembles the subgoal hash tables in its essential details, and also requires a variant-check in the case of hash collisions. Subgoals are hashed on the outer functor symbol of their first argument, while answers are hashed on the combination of the outer functor symbols of all their arguments. Note that this is optimal for Datalog. As a result, hash-based tabling consists of a quick insert, but a slow check if hash collisions occur. On the other hand, trie-based tabling consists of a relatively slower insert than the hash-based —it must set parent and sibling pointers— but combines the check and insert steps, and thereby may need to copy less information for answers. Substitution factoring is implemented only in the trie-based methods.

[4] All benchmarks were run on a SparcStation 2 with 64MB of main memory running SunOS 4.1.3. Sizes of the benchmark programs do not reflect limitations in any of the systems evaluated.

We begin by comparing the hash-based methods to the trie-based method without dynamic compilation. The first set of tests use standard left (Figure 8(a)), and right (Figure 8(b)) recursive transitive closures. A Datalog binary tree was used as

```
a(X,Y) :- p(X,Y).                a(X,Y) :- p(X,Y).                a(Y) :- query(X), p(X,Y).
a(X,Y) :- a(X,Z), p(Z,Y).        a(X,Y) :- p(X,Z), a(Z,Y).        a(Y) :- a(X), p(X,Y).
          (a)                              (b)                              (c)
```

Figure 8: (a) Left, (b) right, and (c) NRSU-factored recursive transitive closures.

the EDB relation (shown in Figure 9(a)). As an additional test, the tree was nested in a unary structure (Figure 9(b)). *Unification factoring* [5] was used to compile the

(a) $\mathtt{p}(1,2), \mathtt{p}(1,3), \ldots, \mathtt{p}(2^n-1, 2^{n+1}-1)$
(b) $\mathtt{p(f}(1)\mathtt{,f}(2)\mathtt{)}, \mathtt{p(f}(1)\mathtt{,f}(3)\mathtt{)}, \ldots, \mathtt{p(f}(2^n-1)\mathtt{,f}(2^{n+1}-1)\mathtt{)}$

Figure 9: (a) Datalog, and (b) structured binary trees for the programs of Figure 8.

structured EDB. Unification factoring processes the heads of the `p/2` clauses into a non-deterministic net which, in this case, provides perfect indexing. Figure 10 provides times for 25 iterations of the queries `?- a(1,X)` to non-structured EDBs and `?- a(f(1),X)` to structured EDBs. In the tables, Height refers to the height of the tree, while Trie and Hash indicate the use of trie and hash-based methods

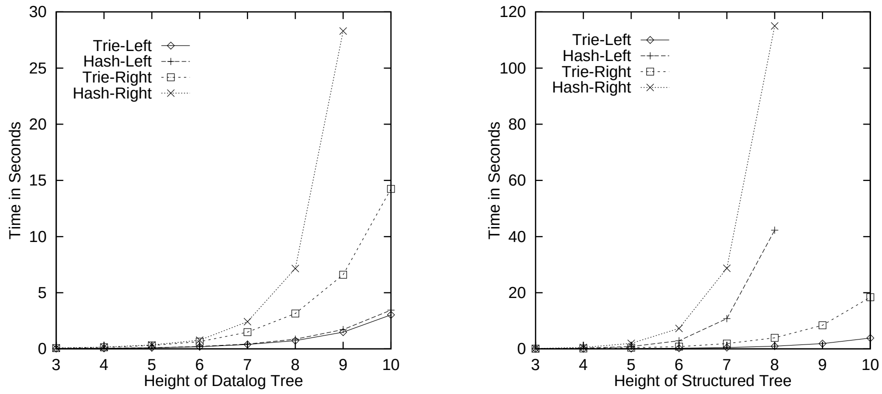


Figure 10: Perfomance Times for Transitive Closures on Trees.

respectively. Left and Right stand for left and right-recursive definitions of transitive closure.

The graphs in Figure 10 indicate the power of tries. For the Datalog cases, and especially for left recursion, times for hash and tries are generally similar, with tries having a slight advantage for large data sets where the effect of hash collisions is more noticeable. However as soon as discriminating information is nested within structures, the times for tries become far more efficient than those for hashing. This divergence is due to the trie's ability to effectively index subgoals and answers on constants within the symbol `f/1` in the structured data, an ability not shared by hash-based tabling. While substitution factoring provides improvement to the trie speed, it does not account for the divergence. To see this, note that answer substitutions to the query `?- a(f(1),X)` have the form `X=f(`i`)` for some i, so that hashing on the outer functor symbol alone would be no more effective for answer substitutions than for answers. In general, complete indexing as provided by tries is necessary for efficient answer check/insert.

Further evidence for the adaptability of trie-based tabling is obtained by comparing two versions of the *same generation* program. The program `sg2/2` (shown in Figure 11(b)) differs from `sg1/2` (Figure 11(a)) in having an unorthodox ordering of body literals. The parent relation (`cyl/2`) for this test is a 24x24x2 randomly generated cylinder. This cylinder can be thought of as an array of 24x24 nodes, where each node in each row (except the last) is connected to two nodes in the next row. A same generation query was made for a node on the first row of the cylinder for each program. Figure 11(c) indicates that the poor literal ordering of `sg2/2` had

```
sg1(X,X).
sg1(X,Y) :-
    cyl(X,Xp),
    sg1(Xp,Yp),
    cyl(Y,Yp).
```
(a)

```
sg2(X,X).
sg2(X,Y) :-
    cyl(X,Xp),
    cyl(Y,Yp),
    sg2(Xp,Yp).
```
(b)

program	sg1/2	sg2/2
Trie-based	1.04	15.48
Hash-based	1.29	201.65

(c)

Figure 11: Two versions of same generation, (a), (b), and their execution times (c).

a smaller effect on the trie-based tabling than on the hash-based. The degeneration in the hash times of `sg2/2` is due to the generation of a large number of subgoals of the form `sg2(`*bound*, *bound*`)`. While the times for the hash-based access method could be improved for this example by hashing subgoals on the outer functor of both arguments rather than on their first argument alone, the trie-based method adapts itself automatically for the subgoal check/insert operation on this example, just as it did for the answer check/insert in Figure 10.

6.1.1 Substitution Factoring

In order to test the effectiveness of substitution factoring in performing dynamic argument reduction, the performance of a *NRSU-factored* left recursive program (shown in Figure 8(c)) was compared against that of a non-factored left-recursive program (Figure 8(a)) using Datalog chains as EDB relations. Table 1 shows that the times (for 25 iterations) were nearly indistinguishable.

Length	32	64	128	256	512	1K	2K	4K	8K
a/1	.05	.10	.17	.31	.62	1.20	2.45	5.15	11.18
a/2	.06	.10	.17	.33	.65	1.29	2.54	5.21	11.53

Table 1: Comparison of static (`a/1`) vs. dynamic (`a/2`) argument reduction.

6.2 Accessing and creating dynamic terms

After a table for a particular goal is completed, program clause resolution will no longer be used for that goal: instead resolution will be made against answers in the table. Since answers are computed dynamically, one measure of the success of our table access methods can be obtained by comparing the time to create and access tables against the time to create and access asserted code. Accordingly, the first two columns of Table 2 show times required to backtrack through various sets of dynamically created terms using an open call. Specifically, the first column presents the time to retrieve answers from a completed table by interpreted tries, and the second column by compiled tries. For comparison, we provide times for asserted code in XSB and Quintus. XSB partially compiles code when it is asserted, so that for unit clauses, asserted code is almost identical to compiled. Quintus indexes asserted code as compiled, but performs variable bindings for asserted code in an interpretive manner. (Quintus compiled code is 2-3 times faster than asserted for unit clauses). All times in Table 2 represent 100 iterations except for the "Binary Tree" cases which represent 10000 iterations.

Form and number of Terms		Interpreted Tries	Compiled Tries	asserted (XSB)	asserted (Quintus)
p(i)	$0 \le i \le 4K$	4.95	3.71	3.71	7.10
p(a,i)	$0 \le i \le 4K$	6.5	3.76	4.91	7.85
p(i,a), p(i,b)	$0 \le i \le 2K$	6.63	4.78	4.88	7.84
p(i,2,...,10)	$0 \le i \le 4K$	18.75	18.91	15.44	14.55
p(i,2,...,100)	$0 \le i \le 200$	7.68	8.39	6.41	4.35
Binary Tree	Level 6	9.67	4.88	7.38	8.58
Binary Tree	Level 7	24.07	9.07	16.23	19.10
p(f(i))	$0 \le i \le 4K$	6.13	3.75	4.31	7.40
p(f(i,2,...,10))	$0 \le i \le 4K$	14.78	18.93	9.04	9.60
p($\overbrace{\mathtt{f}(\ldots\mathtt{f}(}^{10}i)\ldots)$)	$0 \le i \le 4K$	16.08	3.77	16.94	11.64
p($\overbrace{\mathtt{f}(\ldots\mathtt{f}(}^{10}i,2,\ldots,10)\ldots)$)	$0 \le i \le 4K$	24.85	18.99	21.67	14.12

Table 2: Times for accessing dynamically created terms of various forms.

We may define a *common prefix* measure for a set, S, of terms as

$$1 - \frac{\text{number of transitions in the trie for } S}{\text{sum of the sizes of the terms in } S}$$

For Table 2 the common prefix measure ranges from about 91% (for the structured data in the second to last row), to no sharing at all (the unary, 10-ary, and 100-ary Datalog terms in rows 1, 4, and 5).

As expected, the performance of compiled tries increases with the common prefix measure. In the admittedly extreme case of p(f(...(f(i))) in the second to last row, compiled tries achieve speed-ups of 4 times over tries without code. However, when the common prefix measure is low, the performance of compiled tries is slightly slower than that of interpreted tries, especially for terms that contain structures. In the case of p(f(i,2,...,10)) the slowdown is due to the fact that compiled tries effectively perform the transformation p(f(i,2,...,10)) :- p(i,2,...,10). so that variables within the f/10 structure lie below the last choice point. These variables must be present as cells within the choice point and must also be trailed. In contrast, the other methods recreate the f/10 structures on the heap. A second point is that for a binary tree, compiled tries will execute about twice the number of choice point instructions as the other methods. (Compiled tries will execute a choice point instruction for every edge of the tree, while the other methods will execute an instruction for every leaf of the tree). However this trade-off of choice points for binding generally seems to be beneficial, according to results in [5] for static code. The comparison of compiled tries against asserted facts in XSB (in the third column of the table) justifies our claim that trie code is equivalent to WAM code, since unlike many Prologs, XSB compiles asserted facts into WAM instructions.

6.2.1 Measuring the Cost of Dynamic Compilation

Having compared the performance of accessing compiled and interpreted answers we next measure the time required for *creating* the tries. Clearly creation time is a critical factor since the code generation phase is performed during run-time during query evaluation. Table 3 presents times for completing tables with and without

Length of Chain	1K	2K	4K	8K	16K	32K	64K
Interpreted Tries	.05	.10	.19	.37	.74	1.50	3.11
Compiled Tries	.05	.11	.21	.40	.81	1.56	3.18

Table 3: Table creation times with and without compilation (using left recursion).

the code generation phase using the left (Figure 8(a)) recursive transitive closure predicates on Datalog chains (similar times can be obtained for right recursion). As the times show, the extra code generation phase incurs only a minimal overhead (less than 5%) to the table creation process. We note that in these benchmarks no answers from completed tables are ever used; they thus provide an upper-bound of the actual cost of code generation. In cases where the completion of a table involves resolution with answers from already completed tables, the overhead from code generation is usually balanced by the speedup in the time to access these answers. As a last set of benchmarks, we compare the time needed to assert a set of terms with the time needed to create them as compiled tries. Tables 4, and 5

Size	4K	5K	6K	7K	8K	9K	10K
Asserted Code (XSB)	1.51	1.98	2.35	2.84	3.18	3.64	3.96
Compiled Tries (XSB)	.10	.12	.15	.17	.21	.25	.28
Asserted Code (Quintus)	1.73	2.15	2.58	3.01	3.50	3.86	4.35

Table 4: Creation times for unary Datalog data ($\mathtt{p}(i)$, $1 \leq i \leq \mathsf{Size}$).

present times to create unary and 10-ary Datalog facts. In addition, Table 6 shows times to create a unary fact used in Table 4 when its argument is nested in a unary function symbol.

As shown in Tables 4, and 5, storing terms as code in trie-based answer tables is about 10-20 times faster than using `assert`. Note that all these terms are perfectly indexed on their first argument. As soon as the discriminating information is nested within structures and hash collisions start to occur with the use of `assert`, storing the terms in the trie-based table structures exhibits an even bigger performance improvement. Table 6 shows that the use of tables for storing dynamic terms in the presence of hash collisions is faster than `assert` by two orders of magnitude. Given the competitive retrieval speed of tries, their complete discrimination, and

Size	4K	5K	6K	7K	8K	9K	10K
Asserted Code (XSB)	2.45	3.30	4.08	4.95	6.06	6.85	7.86
Compiled Tries (XSB)	.24	.33	.41	.49	.52	.66	.76
Asserted Code (Quintus)	1.85	2.28	2.66	3.12	3.72	4.17	4.48

Table 5: Creation times for 10-ary Datalog data ($\mathtt{p}(i, 2, \ldots, 10)$, $1 \leq i \leq \mathsf{Size}$).

their superior creation time, they appear an intriguing alternative to asserted code for dynamic data when their order need not be preserved.

Size	4K	5K	6K	7K	8K	9K	10K
Asserted Code (XSB)	8.00	12.44	18.36	24.09	31.40	39.45	49.02
Compiled Tries (XSB)	.15	.19	.21	.26	.28	.34	.35
Asserted Code (Quintus)	8.62	13.38	18.67	25.00	31.22	39.48	48.25

Table 6: Creation times for unary structured data ($\mathtt{p(f(}i\mathtt{))}$, $1 \leq i \leq \mathsf{Size}$).

7 Discussion

The trie-based approach with which we address the *table access problem* has important properties in its ability to index data of different forms, and in its single pass check/insert operation. When extended with substitution factoring this approach provides dynamic argument reduction, and indeed, reductions within complex terms. Further, when tries are dynamically compiled, their access time compares well with WAM code, and the amount of binding on backtracking can in some cases be greatly reduced.

This approach reflects the dynamic nature of subgoal and answer creation, a characteristic which distinguishes the results of this paper from other recent work. Fundamentally, tabling tries must partition dynamically changing sets of terms. In contrast, the unification factoring automata of [5] compiled a static set of program clause heads into a trie-like structure for which optimality properties were proven. Finally, as mentioned in Section 3, both the dynamic nature of substitution factoring and its applicability to non-Datalog programs separates it from static methods such as NRSU factoring.

As mentioned earlier, our tabling tries are variants of discrimination nets. In particular, the call and incomplete answer tries can be viewed as discrimination nets over ground terms. However, the relationship between a completed answer trie and a discrimination net is a little subtle. Firstly, our completed answer tries are compiled whereas traditionally discrimination nets have been interpreted. Secondly, our completed tries perform unification operations whereas discrimination nets do match operations.

Our work has extended that reported in [16], which described the SLG-WAM as a whole, but did not examine table access mechanisms in depth, or consider either substitution factoring or compiled tries. While our approach has been developed for the XSB system, we believe that tabling tries and substitution factoring may also prove useful to the portable SLG system of [12].

Little else has been published concerning algorithms for table access, although [15] and [6] describe structure-sharing algorithms for tabling in the context of an evaluation engine. While useful bounds can be derived for the amount of copying needed by a structure-sharing approach, such approaches may be subject to high constant overheads, and in any case do not appear suitable for a WAM-based implementation. In general, implementing logic as needed by deductive databases is a difficult task, and one for which a complete solution — that evaluates in-memory queries as well as a programming language, and queries to disk-resident data as well as a database system — is not yet at hand. Under various guises, the *table access problem* is central to deductive databases. The performance of the trie-based approach gives reason to expect that it will form a part of future deductive databases as it does in present versions of XSB.

Acknowledgements

The authors thank C. R. Ramakrishnan for his help in the preparation and proofreading of this paper, and the anonymous referees for suggestions that helped improve the clarity of the presentation. This research was supported in part by NSF grants CCR-9404921, CDA-9303181, and INT-9314412.

References

[1] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74. Springer-Verlag, April 1993.

[2] T. Chen, I. V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 639–653, Washington, USA, 1992. The MIT Press. Also in *Software Practice and Experience*, Vol 24(12), pages 1097–1119, December 1994.

[3] W. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.

[4] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–440, July 1977.

[5] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for the efficient execution of logic programs. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 247–258, San Fransisco, California, January 1995.

[6] E. V. de la Clergerie. Layer sharing: an improved structure-sharing framework. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 345–359, 1993.

[7] D. E. Knuth. *The Art of Computer Programming: Vol 1 Fundamental Algorithms.* Addison Wesley, 2^{nd} edition, 1973.

[8] W. W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:147–167, 1992.

[9] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, 1989.

[10] H. J. Olbach. Abstraction tree indexing for terms. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 479–484, Stockholm, Sweden, August 1990. Pitman Publishing, London.

[11] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 1995. To appear.

[12] R. Ramesh and W. Chen. A portable method of integrating SLG resolution into Prolog systems. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 618–632. The MIT Press, 1994.

[13] P. Rao, I. V. Ramakrishnan, T. Swift, and D. S. Warren. Dynamic argument reduction for in-memory data queries. In *Second International Workshop on Logic Programming and Deductive Databases*, ICLP-94, Italy, June 1994.

[14] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994.

[15] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 557–574. The MIT Press, 1993.

[16] T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 633–652. The MIT Press, 1994.

[17] A. Voronkov. Implementing bottom-up procedures with code trees: a case study of forward subsumption. Technical Report 88, Uppsala University, Uppsala, Sweden, October 1994.

[18] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.