XSB as an Efficient Deductive Database Engine

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

Abstract

This paper describes the XSB system, and its use as an in-memory deductive database engine. XSB began from a Prolog foundation, and traditional Prolog systems are known to have serious deficiencies when used as database systems. Accordingly, XSB has a fundamental bottom-up extension, introduced through tabling (or memoing) [4], which makes it appropriate as an underlying query engine for deductive database systems. Because it eliminates redundant computation, the tabling extension makes XSB able to compute all modularly stratified datalog programs finitely and with polynomial data complexity. For non-stratified programs, a meta-interpreter with the same properties is provided. In addition XSB significantly extends and improves the indexing capabilities over those of standard Prolog. Finally, its syntactic basis in HiLog [2], lends it flexibility for data modelling.

The implementation of XSB derives from the WAM [25], the most common Prolog engine. XSB inherits the WAM's efficiency and can take advantage of extensive compiler technology developed for Prolog. As a result, performance comparisons indicate that XSB is significantly faster than other deductive database systems for a wide range of queries and stratified rule sets. XSB is under continuous development, and version 1.3 is available through anonymous ftp.

1 Introduction

The language of first-order Horn clauses extended with closed-world negation and arithmetic predicates, has proven an elegant language for data-oriented problems. By making various syntactic restrictions one obtains languages equivalent to known database languages. For example, by excluding function symbols and recursion, and constraining rules to be safe, one obtains a simple domain calculus language equivalent to the relational algebra of traditional relational databases. By relaxing the restriction that excludes recursion, one obtains datalog (with negation), the language that underlies deductive databases.

Logic programming has an efficient computational mechanism for the full Horn clause language, a backtracking search through the tree of SLD refutations. However, the SLD computational mechanism, which well serves the needs of a programming language, is clearly inadequate as a database computation strategy. Its most serious drawback is that it does not terminate for the datalog language. Datalog is a decidable language (one reason that makes it a reasonable candidate for a database language) but SLD refutation is not finite on it. This lack of finiteness is not a serious problem for a programming language — indeed it is a necessary property of Turing complete languages — but it is unacceptable in a database language. Another problem is SLD's tendency to recompute the same answer. In a programming context, this is acceptable since the programmer is responsible for avoiding such "poor" programs; in a database context, where the poser of the query is not expected to control the evaluation, it can be a serious problem. Another practical problem with SLD implementations (i.e., Prolog systems) is that their support of indexing has tended to be very primitive. Again the programmer is expected to write programs in a form that effectively use the available indexing. And finally, Prolog implementations are tuple-at-a-time, an appropriate strategy for a programming language, but one usually considered less useful for disk-resident data, where set-at-a-time strategies have been deemed important.

The deductive database community has adopted datalog as a leading database query language, identified these problems, and rectified them. Rewriting techniques have been developed to introduce goal-directedness into a bottom-up, set-at-a-time evaluation strategy. These techniques solve SLD's problems of lack of finiteness and redundant computation. Also, the interaction of closed-world negation and recursion in datalog has proven to be a major issue, and the deductive database community has made significant contributions to developing the necessary underlying semantics [21]. Further, prototype deductive database systems, exemplified by LDL [6], Glue-Nail [7], and CORAL [12], have been designed and developed to incorporate these advances. It should be mentioned that all these systems process all recursive data in memory.

XSB offers an alternative approach to creating a deductive database system. Rather than depending on rewriting techniques, it extends Prolog's SLD resolution in two ways: 1) adding tabling to make evaluations finite and non-redundant on datalog, and 2) adding a scheduling strategy and delay mechanisms to treat general negation efficiently. The resulting strategy is called SLG resolution [4, 3], which is complete and finite for non-floundering programs with finite models, whether they are stratified or not.[1] XSB's engine currently implements a version of SLG restricted to left-to-right modularly stratified programs [14]. A meta-interpreter is provided to evaluate non-stratified programs according to the well-founded semantics [21] or, equivalently, to the three-valued stable model [11] semantics for that program.

XSB's query engine is based on an extension of the WAM [25] (Warren Abstract Machine). Being based on a well-developed Prolog technology, it can take advantage of advanced techniques for efficient compilation. XSB evaluates stratified queries much faster than current bottom-up implementations, a claim that will be substantiated in section 5. Perhaps the most significant difference between the XSB implementation and conventional deductive database implementations is XSB's tuple-at-a-time evaluation strategy. This gives it an advantage in object-oriented applications where information can be kept in complex terms or distributed across many relations. In addition its syntax, which is based on HiLog [2] extended with Prolog operators, offers a useful means of knowledge representation for object-based schemas. HiLog predicates are fully compiled into SLG-WAM instructions, and execute only marginally slower than traditional Prolog predicates.

An advantage of using a top-down approach is that XSB's efficiency relies less on rewriting techniques and on alternate control strategies than is usual in bottom-up approaches. This allows XSB to maintain an operational interpretation of a program in almost as elegant manner as Prolog – but without relying on the user's knowledge of that interpretation to avoid infinite loops and redundancy.

In addition to its declarative language features, XSB offers flexible indexing along with efficient I/O mechanisms to link XSB with a modifiable backing store. XSB offers hash based (WAM-style) indexing on alternate arguments of a tuple or on combinations of arguments. It also implements an original indexing technique called *transformational indexing*, which stores parts of clauses in a non-deterministic discrimination network. For I/O, XSB offers fast dynamic compilation along with linking of dynamic code.

Version 1.3 of XSB has been tested on over a dozen hardware and operating system platforms[2] and on databases whose relations have on the order of hundreds of thousands of tuples. It is available through anonymous ftp from `cs.sunysb.edu`.

This paper is intended to present an overview of XSB and, to a lesser extent, of SLG resolution, which is not yet widely known. We give pointers to other papers which provide fuller treatments of SLG, HiLog, XSB performance, and other topics. After briefly discussing related work, this paper presents an overview of the SLG derivation in section 3. The main body of paper makes up section 4 and provides overviews of the functionality of HiLog syntax, the SLG-WAM, and XSB indexing and I/O. Performance results are reported in section 5.

2 Related Work

Table 1 summarizes aspects of some deductive database systems developed recently. The table was adapted from [13] which provides a wider comparison.

Unlike XSB, most of the systems in table 1 use an extension of the magic set approach. Seki in [16] proves that, for programs without negation, a "top-down" resolution method, QSQR, is asymptotically the same as a bottom up method, Alexander Templates. There are a number of technical details that would need to be worked out in order to use Seki's result to compare SLG on definite programs to magic templates. Even so, the similarities between the two methods are striking. For instance, the magic facts of the magic template method appear to correspond to the tabled subgoals of an SLG evaluation.

In terms of syntax, table 1 describes systems that allow functions in arguments and non-range restricted queries as first-order. It is interesting to note that Glue-Nail provides restricted HiLog functionality although the outer functor of a predicate can be a datalog term but not a general first-order term. For instance, the term `p(f(X))(Y,Z)` would not be allowable in Glue-Nail, but would be in XSB.

[1] A program flounders if there is an atom whose truth cannot be proven without making a call to a non-ground negative literal.

[2] Currently SPARC, MIPS, Intel 80X86, Motorola 680X0, and IBM RS6000 chips have been tested; for operating systems, SUNOS, SOLARIS, IRIX, ULTRIX, LINUX, 386BSD, AMIGA-DOS, HP-UX, System V r.3-4, SCO Unix, and Mach have been tested.

Name	Evaluation	Syntactic Restrictions	Negation	Data Requirements
Aditi	Magic Sets	Datalog	Mod. Strat	Disk-Resident
CORAL	Magic Templates	First-order	Mod. Strat.	Main-Memory
LDL	Magic Sets	First-order	Mod. Strat.	Main-Memory
Glue-Nail	Magic Rewriting	F-O with Restricted HiLog extensions	Well-founded	Main-Memory
Syllog	Backchain Inferencing	Datalog	Stratified	Disk-Resident
XSB	SLG	HiLog with F-O optimisations	Well-founded	Main-memory

Table 1: Some Deductive Database Systems

Both Glue-Nail and XSB are able to evaluate programs according to the well-founded semantics [9, 3], although they both seem to fall back onto interpretive techniques for non-stratified negation.

Finally, only Aditi [20] and Syllog [23] are able to manipulate disk-resident relations, while the others (including XSB) need to copy relations into main memory. It should be mentioned that nothing in XSB's use of SLG precludes the tabling of non-recursive predicates which are computed externally in a database system. Using this extension, XSB could evaluate queries on datasets that are too large for main memory, although the active sets of the queries would have to fit into memory. If the active set of a recursive query does not fit into main memory, its efficient evaluation is an open problem both for SLG and for magic evaluation.

Table 1, does not classify the various systems on the basis of their treatments of sets or aggregation. LDL and CORAL allow sets as objects of the universe, while the HiLog systems, Glue-Nail and XSB, use terms as the names of sets, and use negation to construct predicates for set equality, subset checks, and other operations. This approach in discussed further in section 4.7.

There are other differences not reflected in the table. Some of the other systems automatically optimize programs or queries, along the lines of traditional database systems, a task which XSB does not yet do. Rather, XSB can be thought of as an engine upon which optimizations such as literal reordering or index selection can be implemented. An implication of the results of section 3 is that such an optimizer could be added to XSB in a relatively straightforward manner due to the termination properties of SLG.

A final distinction is that XSB code is compiled to a lower level than is usual with database systems. Section 5, clarifies this distinction, and provides performance comparisons of XSB with some of the systems in table 1.

3 SLG evaluation

3.1 SLG Resolution

The evaluation strategy underlying the XSB system is called SLG resolution, which uses memoing to evaluate general logic programs. The details of SLG evaluation have been presented elsewhere [4, 3] and are beyond the scope of this paper; we give only a short summary of the important features of SLG here.

On definite programs, SLG reduces to SLD resolution with memoing [24]. Each call to a selected subgoal must check whether (a variant of the) subgoal has been previously called during the evaluation. If not, the subgoal is copied to a global *table* and program clauses are resolved against the subgoal exactly as in SLD. Later in the evaluation, when the subgoal is completely resolved away, the corresponding answer instance is copied back into the table. If at the time of a call, a variant subgoal *has* been previously called, the subgoal is resolved against answer clauses that are in the table. The evaluation is completed when all program clauses and all answer clauses have been resolved against all applicable subgoals. Answer clauses may be created during the course of the same evaluation that uses them, so that resolving a called subgoal with an answer clause may lead to the generation of another answer clause that must be resolved with the subgoal, and so on. Thus the evaluation may be seen as a sort of fixpoint computation. Subgoals whose evaluation has converged to the fixpoint are termed *completely evaluated.*

For programs with negation that are stratified, SLG *suspends* evaluation as necessary to guarantee that subgoals in lower strata are completely evaluated when their negations are required.[3] This strategy gives SLG a polynomial data complexity for datalog programs with negation.

Non-stratified programs require a more complex handling of negative subgoals than stratified programs. In SLG, non-stratified programs must *delay* at least some calls to negated subgoals. Delaying of a literal corresponds to a (perhaps temporary) assumption that

[3] As a practical matter, *Existential Negation* can provide an alternative behavior in SLG programs, and is explained later.

it is undefined in the well-founded model. Such delayed literals show up as conditions on answers in the table. A delayed literal may later be found to be false or true, in which case it is simplified away, and the simplification may lead to further answers. However, it will not always be the case that a subgoal can be proven either true or false; there may be cyclic dependencies among the delayed subgoals in a set of subgoals. In this last case, the subgoals whose delays cannot be eliminated have a truth value of *unknown* in the well-founded semantics.

In fact the answer clauses (answers conditioned by delays) can be seen as constituting a transformed program from which sets of 3-valued stable models can be computed, as explained in [5].

3.2 Implementation of SLG

Because of the similarity of SLG to SLD on definite programs, it is natural to ask whether an SLG meta-interpreter or preprocessor could be written based on Prolog using SLDNF. A meta-interpreter has been written but its speeds have turned out to be unacceptable for general programming. A preprocessor that transforms a program into a Prolog program that evaluates according to the SLG strategy might be expected to have better efficiency, but the fixpoint nature of the SLG strategy makes this difficult. In effect the strict backtracking nature of Prolog's search must be broken; work on a branch of the SLD search tree must be suspended while other branches are searched and later must be resumed. Interpreters and preprocessors which try to compute SLG (or OLDT) end up needing computational mechanisms not natural to the WAM, and as a result their performance suffers.

The SLG-WAM contains WAM extensions for efficient evaluation of SLG[4] and is roughly 100 times faster than its meta-interpreter running on a similar emulator. A separate memory area, the *table space*, was added to the WAM for table manipulation, and routines have been added to copy derived answers from the SLG-WAM program stacks to table space and back. An associated paper ([18]) will discuss the extensions in detail.

Using the SLG-WAM to execute Prolog's SLD resolution incurs only minimal overhead. The small overhead is due to the more complex trailing and testing involving memory pointers. Comparisons of the SLG-WAM with PSB-Prolog, from which it is derived, indicate that the SLG-WAM is usually less than 10% slower than PSB-Prolog's WAM, and is sometimes faster.[5] As a Prolog system using only SLD, XSB runs a bit slower than emulated SICStus, on standard benchmarks, and about three times slower than Quintus.

[4] The current implementation is limited to stratified programs, although detailed design has begun for the full SLG-WAM.

[5] The comparison is made murky by the fact that numerous improvements were made to the emulator in the course of implementing the SLG-WAM and in preparing XSB for release.

4 The XSB Engine

This section presents aspects of XSB that are of use to deductive database programmers and users. These aspects include HiLog syntax, and novel high-level mechanisms to control evaluation: such as tabling declarations, the table cut, and different operational models of negation. In addition we discuss more traditional deductive database features: indexing, interfaces with I/O, and features for set manipulation.

4.1 Syntax

XSB's syntax is based on that of HiLog [2]. HiLog provides a higher-order syntax for logic programs allowing arbitrary terms to occur as the functor of a term (or predicate symbol of an atom.) As an example, `X(bob,Y)` is a well-formed HiLog term. HiLog terms can be encoded into first-order logic using a family of *apply* function and predicate symbols. A full development is given in [2]; we sketch the intuition here. Briefly, for a HiLog term, T, of arity $N \geq 0$, the encoding uses the apply symbol of arity $N+1$, where the first argument of `apply/(N+1)` is the encoding of the functor of T, and the remaining N arguments are the encodings of the N arguments of T. The HiLog term above would translate into the term `apply(X,bob,Y)`. This encoding provides a first-order semantics for HiLog.

To be precise, let $\mathbf{L}$ be a language with a countably infinite set of variables $\mathcal{V}$ and a countable set of logical symbols $\mathcal{S}$. Then the set $\mathcal{T}$ of HiLog terms is the minimal set of strings over the alphabet of $\mathbf{L}$ such that

- $\mathcal{V} \cup \mathcal{S} \subseteq \mathcal{T}$
- If $t, t_1, ..., t_n$ are in $\mathcal{T}$, then $t(t_1, ..., t_n) \in \mathcal{T}$, $n \geq 1$.

In XSB, $\mathcal{S}$ consists of Prolog constants and integers, while $\mathcal{V}$ consists of Prolog variables. Examples of HiLog terms in XSB are the terms: `X`, `X(1)`, `parent('John','Mary')`, `r(X)(parent(X,'Mary'))`.

Logically, first order terms are simply a subset of HiLog terms, but operationally, they can be compiled into somewhat more efficient code. For the terms, `3('John')`, or `r(X)(parent(X,'Mary'))`, the functors `3` and `r(X)` can be deduced to be HiLog functors, and are compiled accordingly. From the terms `p(X,Y)` and `h(X,Y)`, where the functor is a constant, it is not possible to determine whether `p` and `h` are to be taken as HiLog functors or as first-order functors. XSB's design decision is to require the explicit declaration of HiLog constants.

```
:- hilog(h).
```

This declares that `h`, when it appears in a functor or predicate position, is to be read as a HiLog symbol and the appropriate translation is to be done. For example, the term `h(a)` will be read as `apply(h,a)`. Section 4.7 will further discuss HiLog implementation in XSB.

In XSB, a *fact* is simply a HiLog term followed by a period. Indeed, any of the HiLog terms in this section can be used to define database facts. Rules in XSB define logical predicates. Full XSB syntax is described in [15]. A *proper HiLog rule* (or clause) is of the form:

$$Term \text{ :- } L_1, L_2, ... L_n.$$

where each literal, L_i, is either an atom, which in HiLog is simply a HiLog term, or a negation of an atom: \+ *Term*.

For further convenience XSB integrates Prolog's ability to define operators with the HiLog syntax.

4.2 Architecture

The top-level architecture of XSB is shown in figure 1. Components coded in C are drawn within boxes; those

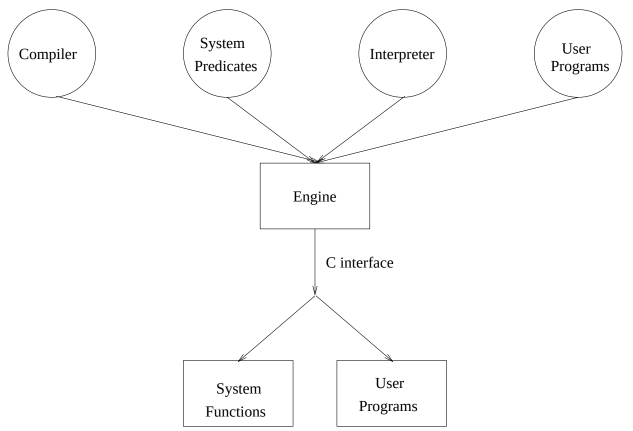

Figure 1: Overview of XSB Architecture

coded in HiLog are drawn in circles. HiLog programs are compiled into byte-code object files, which can then be loaded and executed by the SLG-WAM emulator, which is written in C. Loading of byte-code files is done dynamically at run-time, as they are needed. Most of the system code — such as the compiler or the I/O routines — underlying XSB is written in a mixture of HiLog and C. Both system and user code use the same C interface, which essentially causes the engine to trap calls to C functions and relinquish control. On several platforms, C code can be dynamically linked just as HiLog code can be. The simplicity of the architecture allows the structure of the system to be easily understood and extensions to be easily made.

Although XSB is normally invoked using its read-eval-print loop interpreter, it can also directly execute compiled user programs. This direct execution of object files allows programs written using XSB to execute directly from a shell in UNIX, AMIGA-DOS, etc.

XSB's compiler assumes that predicates are static and do not change during execution, whether they are defined by rules, facts, or a mixture of the two. Like most compilers, it is unsuitable for handling large amounts of data.[6] As an alternate to compilation, XSB provides for *dynamic predicates*, which have code that can be modified during execution. An extensional database is normally stored as dynamic predicates. Facts or rules can be asserted one at a time, and retracted either one at a time at the clause level or all at once at the predicate level.

In the XSB implementation, static code is fully compiled to take full advantage of the SLG-WAM; dynamic code is also compiled, but it uses a simplified compilation method in order to be faster. Each dynamic clause is compiled as though it were defined by a rule with a single literal as its body. The overall result is that dynamic database facts have almost identical representation as compiled facts and so execute at essentially the same speed.[7] Section 4.6 will discuss interfaces of dynamic predicates with backing store.

XSB also supports the grouping together of predicates into modules. The module system permits encapsulation, allowing terms (predicates, structures and constant symbols) to be hidden, imported, or exported. Most module systems for logic programming are predicate-based. XSB's module system is *term-based*, which we believe presents a more coherent way to structure HiLog programs. The module system is used by the compiler in a number of ways: import declarations provide information to allow dynamic loading of predicates on first use, and module declarations define the scoping for a number of declarations described in the next few sections.

4.3 Declaring Predicates Tabled

As mentioned in section 3, predicates in XSB are executed using SLDNF by default and can be declared tabled on a per predicate basis. As an alternative, the system will decide which predicates to table when given the declaration:

```
:- table_all.
```

This causes the compiler to table enough predicates to avoid infinite loops due to redundant calls. Determining precisely which predicates need to be tabled is the same as predicting whether a particular goal will be

[6] A large amount of data currently means files of more than about 15000 lines of code.

[7] The complication is in the treatment of indices, since dynamic clauses currently support only hash-based indexing, whereas static clauses support both hash-based and transformational indexing.

repeated on a path of an SLD tree and is undecidable in general. In this version of `table_all`, simplicity and speed were chosen over refinements in the precision of the algorithm. Intuitively, `table_all` constructs the call graph and chooses to table enough predicates to ensure that all loops are broken. It may happen that `table_all` chooses too many predicates. If this happens, the user can specify explicitly what to table, or alternatively, can separate the definitions of the offending predicates into another module, since the scope of the `table_all` declaration is limited to the module in which it is given.

4.4 Cuts and Negation

Prolog's implementation of SLD is a tuple-at-a-time strategy, which means that tuples of relations are produced on demand only, one tuple at a time. Therefore computation may be avoided by aborting the generation of tuples by a predicate partway through its computation, assuming that the remainder of the tuples are not desired. The computational savings may be significant because terminating the tuple generation of a predicate also terminates the tuple generation of all predicates being used in its computation. Prolog's `cut` operator (!) allows the programmer to specify such early termination.

Using cuts, of course, changes the semantics of programs. Prolog programmers depend on them for an efficient implementation of the conditional operator, as exemplified in the following definition of a simple predicate to transform null values:

```
transform_null(null,'date unknown') :- !.
transform_null(X,X).
```

When `transform_null` is joined with a relation that determines the first argument, if the first clause matches, the result `'date unknown'` is returned as the second field and the cut indicates that no other answers are to be found. If the first clause does not match, then the second clause simply returns the first argument. In either case, only a single tuple will appear in the relation `transform_null`.

Another example of the use of cut in Prolog is in the definition of `not` (\+). Consider the following definition of `not_p/2`:

```
not_p(X,Y) :- p(X,Y), !, fail.
not_p(X,Y).
```

Assuming `not_p` is joined with other relations in such a way that both its arguments are bound (i.e., a safe use), then if the tuple is in `p/2`, the cut indicates that this is the only answer needed, so that the `fail/0` predicate causes the computation to immediately fail, i.e., indicate that `not_p(a,b)` is false. If `p(a,b)` is false, then the cut is not executed and the second clause is used instead. The crucial point is that if `p/2` is itself defined using a number of other relations, then the cut, if executed, will close off the computation of all those relations as well, possibly leading to significant computational savings. Some of the effects of cuts can be obtained in a set-at-a-time framework using *choice* operators [6].

This discussion of Prolog and cut is also relevant to SLG resolution, since it, too, is a tuple-at-a-time strategy. However in the context of tabling, cut is more complicated. A problem may arise when a tabled predicate occurs in the scope of a cut. The cut indicates that the current user of this table doesn't desire any more tuples, but it may be the case that other users of the table do. In this case, closing the table at this time would result in incorrect answers. In general, predicates being tabled must be completely evaluated in order to insure complete and correct answers. In fact, the XSB compiler will give an error if a static analysis of the program shows that a cut might close any partially computed tabled predicate.

However, it can also be dynamically determined whether a table has subgoals depending on it that may be inadvertently affected by a cut. If all ancestors of a table in the scope of a cut are also ancestors of the cut, then the table can be safely deleted. Accordingly, version 1.4 of XSB contains the `tcut/0` operator, which checks whether all tables in the scope of a cut can be safely freed, and if so, it frees them. If not, `tcut0` is a simple noop.

XSB has two types of negation for evaluating modularly stratified programs under SLG: SLG negation (`tnot/1`) and Existential Negation (`e_tnot/1`). The difference is basically that SLG negation completely evaluates all tables, whereas Existential Negation uses `tcut/0` to delete tables after finding the first answer to the positive subgoal generated by a negative literal (when it is safe). Both types of negation guarantee correctness for modularly stratified programs, although either can be radically more efficient depending on the context in which it is used.

As an example of where existential negation is more efficient that SLG negation consider the well-known stalemate game:

Example 4.1 Consider the program

```
win(X) :- move(X,Y), \+ win(Y).
```

`win/1` is modularly stratified iff `move/2` is acyclic. Table 2 shows times for evaluating this program on complete binary trees of varying height using SLG negation, SLDNF (i.e. Prolog), and existential negation. The times are normalized to the time for existential negation.

Note that ratio of the times for existential negation and SLDNF is essentially constant, while the relative time for SLG negation increase as the depth of the tree increases.

Height	6	7	8	9	10	11
XSB/Def. SLG	4.5	4.25	7.6	8.2	15.4	15.7
XSB/SLDNF	.3	.24	.22	.24	.24	.23
XSB/E-Neg	1	1	1	1	1	1

Table 2: Comparisons of SLG implementations for complete binary trees

To see why SLG evaluation is worse than SLDNF, consider the calls made by SLDNF for the query *win(1)* over a binary tree of height 4 with 31 nodes. The calls are represented as circled nodes in Figure 2. Because SLDNF checks only for the existence of a solution for a negative subgoal, only 13 out of 31 possible subgoals are evaluated by SLDNF, and in general the execution of *win(1)* over a binary tree grows proportionally to $(\sqrt{2})^n$ in SLDNF rather than to 2^n.[8]

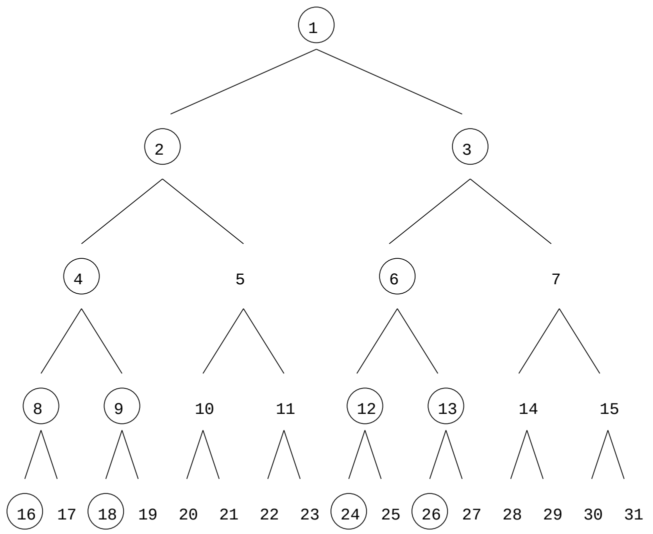

Figure 2: Calls to `win/1` over a binary tree

Existential negation differs from the SLG default in that it "cuts away" goals created in a negative context whenever it can, rather than fully evaluating them. When no subgoal created in a negative context is reused, using existential negation under SLG results in exactly the same nodes being visited as in SLDNF, and in such cases, existential negation should be used.

It is possible, however, that use of existential negation will delete tables that will be needed later. In this case existential negation can result in slower execution times. Indeed, there are datalog programs for which using existential negation rather than SLG negation will cause an exponential increase in the amount of work. So when it cannot be determined that tables are used once only, the default SLG negation should be used.

4.5 Indexing

Traditionally, Prolog systems index on only the main symbol of the first field in a relation, which is clearly inadequate for database applications. Database applications require multi-field indices as well as multiple distinct indices on a single relation. Recall that XSB has two kinds of predicates: static predicates, which do not change during processing (unless they are reloaded in their entirety), and dynamic predicates, which are modifiable during execution. XSB supports different kinds of indexing on these different types of predicates.

For dynamic predicates, XSB supports various indexing options, all of which are based on hashing of the main symbol in a field. The default is hashing on the first argument. However, declarations may be used to indicate the desired options. Indices can be constructed on any field, or set of fields (of size three or less). Also any number of distinct indices on the same predicate is supported. In addition, hash table sizes are specifiable. To take an example, a dynamic predicate `p/5` might have the following index declaration:

```
:- index(p/5,[1,2,3+5]).
```

which will cause indices to be maintained on `p/5` so that a retrieval will use the index on the first argument, if ground, otherwise on the second, if ground, and otherwise on the third and fifth combined.

For static predicates, the compiler accepts a declaration that indicates the field on which a hash index is to be constructed. As an alternative to hash-based indexing, version 1.4 of XSB offers *transformational indexing* for static predicates. Transformational indexing is a novel technique which uses a sequence of folding and unfolding operations to construct a non-deterministic indexing automaton at compile time. Transformational indexing, which will be discussed fully in a separate paper, compares quite favorably with techniques such as [1], while at the same time being much simpler to implement.

Example 4.2 Consider the predicate

```
p(g(a),f(X)).      p(g(a),f(a)).
p(g(b),f(1)).      p(g(X),Y).
```

when compiled with transformational indexing. Given a mode `p(+,-)`, Transformational indexing factors out the `g/1` structure before indexing on its first argument, and factors out the `f/1` structure before indexing on the terms `X` or `a`. The automaton produced can be seen as a sequential trie whose leaf nodes correspond to the input clause heads and furthermore occur in the same order.

The default indexing on compiled predicates is hash-based. Since the number of clauses is known by the

[8] The exact formula is $G(n) = 2^{\lfloor \frac{n}{2} \rfloor + 2} - 3 + 2(\frac{n}{2} - \lfloor \frac{n}{2} \rfloor)$.

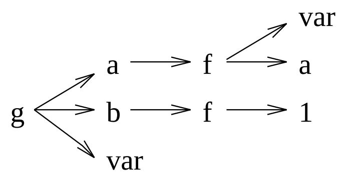

Figure 3: Trie for Transformational Indexing

compiler, it chooses an optimal hash table size. The user can provide declarations to invoke transformational indexing, either for single predicates or for all the predicates in a module.

In addition to indices on user predicates, there are two uses of indices in processing the tables generated by the engine: an index on call patterns to find quickly whether a call has been made previously, and an index on answers to determine quickly whether a new answer is a duplicate of a previous answer.

Indices on tables are important for the following reasons:

- Answer clauses need to be indexed to check for duplication. If an answer, A, is a duplicate of an answer that exists in a table for a call C, the path leading to A is failed (and in this way duplicate computation is avoided). Otherwise the answer is added and eventually resolved against all calls in the tree that are variants of C and use answer resolution.

- Tabled subgoals require indexing since the action taken for a subgoal depends on whether it has been previously called during an evaluation.

For subgoal indexing, XSB uses a simple first-argument hash index. For answer clause indexing, XSB uses a hash index that includes all arguments of the answer. In both cases the XSB user may specify the size of the hash table explicitly to override the default.

Work is currently under way to improve the indexing in XSB. A new implementation of a variant of first-string indexing is in development, which will allow it both to be more efficient and to still apply across variables in the indexed clauses. Also, trie-based indexing is currently being developed for answer clauses in the tables. The index is being integrated with the actual storing of the answers, which will both decrease the space and the time necessary for saving answers.

4.6 Interface with Persistent Store

Because XSB does not compute directly off of disk, efficient mechanisms for bulk communication with a backing store become especially important.

XSB provides interfaces both to ASCII text files and to object files. Code that becomes part of an XSB database might be read originally from an ASCII file, and partially compiled into dynamic code. Dynamic code can be loaded into and unloaded from the system at any time, and updates can be made to this dynamic code through `assert/1` and `retract/1`. Future development of XSB will allow dynamic code also to be backed up to an object file at a user-specified point.

ASCII data can be read by a general reader that handles operators and arbitrary HiLog terms. Because of the generality of the terms that can be read, the default ASCII reader is slower than most database readers and usually takes several milliseconds even for simple terms. For database applications, however, it is much more often the the case that data files are highly structured and do not require sophisticated parsing. To read such files, XSB provides a formatted read, which allows it to read and assert a fact in about a millisecond on a Sparc2, including simple index maintenance. (This time seems roughly equivalent to the data load times of other deductive database systems).

Static code is translated by the XSB compiler into object files, which contain SLG-WAM byte-code. Since object files contain precompiled code, loading an object file is about 12x faster than loading through the formatted read and assert. Currently object files are supported only for static code. Extending object files to dynamic code will allow loading of data in a time roughly equivalent to the time commercial databases require for bulk loads followed by an index.

Intelligent management of large segments of data to and from disk remains a research topic for XSB, as it is for other memory resident deductive database systems. XSB can reclaim space allocated for tables when the entries are completed, without compromising correctness. Such space reclamation may, however, compromise efficiency.

4.7 HiLog and Sets

In this section we describe briefly how HiLog is implemented in the XSB system and how it can be used to manipulate sets. XSB implements HiLog by translation into a first-order form, applying source-level optimizations and compiling the optimized first-order form. The translation simply wraps the arguments of a HiLog term with the symbol `apply/N`, treating the term in the functor position as if it were the first argument. The fragment

```
:- table path(_)(_,_).
path(G)(X,Y) :- G(X,Y).
path(G)(X,Y) :- path(G)(X,Z), G(Z,Y).

:- hilog p.
p(g(a),f(X)).           p(g(a),f(a)).
p(g(b),f(1)).           p(g(X),Y).
```

would be encoded as

```
:- table apply(path(_),_,_).
apply(path(G),X,Y) :- apply(G,X,Y).
apply(path(G),X,Y) :-
    apply(path(G),X,Z), apply(G,Z,Y).

apply(p,g(a),f(X)).    apply(p,g(a),f(a)).
apply(p,g(b),f(1)).    apply(p,g(X),Y).
```

Note that the transformed program can be compiled since it is a Prolog program. The only problem may be its efficiency. To address the efficiency issues, note first that the obvious problem of indexing can be solved by using XSB's transformational indexing. The discrimination graph for this fragment is shown in figure 4, and is essentially a union of the graphs of the predicates in the fragment.

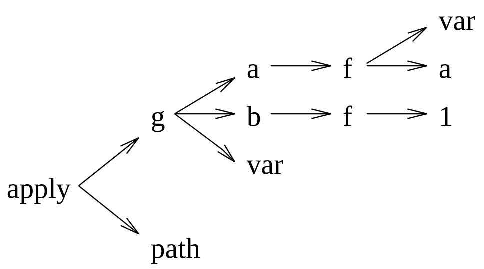

Figure 4: HiLog Discrimination Graph

Clearly, HiLog brings little overhead to p/2, but the overhead introduced for path(G)/2 may be significant. In fact XSB optimises the latter predicate by automatically performing a source code transformation for path(G)/2 though *compile-time specialization* of known calls to HiLog predicates. For this example, the path(G)/2 predicate above would become:

```
apply(path(G),X,Y) :- apply_path(G,X,Y).

:- table apply_path(_,_,_).
apply_path(G,X,Y) :- apply(G,X,Y).
apply_path(G,X,Y) :-
    apply_path(G,X,Z), apply(G,Z,Y).
```

In this form, the path(G)/2 predicate is not much less efficient than if it were written in first-order syntax; there is only some minimal overhead for the extra argument (G) that makes this predicate generic.

Table 3 provides performance results for two sets of example programs. The first set consists of the naive reverse Prolog benchmark (nrev), and a database program (query) that searches populations and areas to find countries of approximately equal population density. The second set of benchmarks (path and map programs) compares generic higher-order versions of a transitive closure and a maplist predicate to non-generic versions of these predicates in which the base relation is hard-coded. For example, the generic transitive closure is as predicate path(G) above but it is right recursive

	nrev	query	path	map
HiLog	3.4	3.9	14.76	11.3
HiLog specialised	1	1	1.12	1.1
Prolog	1	1	1	1

Table 3: Normalized Performance of HiLog

and it is not tabled, while the non-generic transitive closure is as follows (where p is not a HiLog symbol).

```
path_for_p(X,Y) :- p(X,Y).
path_for_p(X,Y) :- p(X,Z), path_for_p(Z,Y).
```

p/2 took the form of a chain. Table 3 presents relative times for the HiLog encoded programs with and without specialization.

HiLog syntax provides an elegant way to construct and manipulate sets. Recall that a complex term can be a predicate symbol. Thus we can use such a term to represent its predicate (i.e., set) of a given arity. For example, in the database:

```
package1(health_ins, required).
package1(life_ins, optional).

package2(health_ins, required).
package2(free_car, optional).
package2(long_vacations, optional).

benefits('John', package1).
benefits('Bob', package2).
```

we use the term package1 to denote the set of John's benefits. Benefits are a set of binary tuples indicating the type of benefit and whether it is optional or required. The query

```
?- benefits('John', P), P(X, Y).
```

binds P to the name of the set of John's benefits, package1, and then retrieves the tuples that describe his benefits explicitly.

This simple representation can be extended to include set intersection and union. For example, we can represent the intersection (or union) of two sets S1 and S2 of binary tuples with the term intersect_2(S1,S2) (or union_2(S1,S2)) and define the extension as follows:

```
intersect_2(S1,S2)(X,Y) :- S1(X,Y), S2(X,Y).

union_2(S1,S2)(X,Y) :- S1(X,Y).
union_2(S1,S2)(X,Y) :- S2(X,Y).
```

To find the common benefits that John and Bob have:

```
?- benefits('John',P), benefits('Bob',Q),
   intersect_2(P,Q)(X,Y).
```

Many other set functions, such as set membership or set equality can be defined in a correspondingly simple manner. See [2] for further examples.

HiLog has a first order semantics, and there are certain second-order functions like *count* and *sum* that

HiLog and tabling alone cannot compute. To solve these problems XSB offers `findall/3`, `bagof/3`, and `setof/3`, as defined in Prolog. XSB also includes an extension of `findall/3` for use in SLG predicates, called `tfindall/3`. If `findall/3` is called on an SLG predicate that has been made a subgoal but has yet not been completed, it will capture solutions from an incomplete list of answer clauses. To avoid this problem, `tfindall/3` *suspends* until the table has been completed. Given the assumption that programs are stratified, the correct answer will be computed, just as in the case of negation.

5 XSB Performance

Detailed comparisons with other deductive comparisons are not complete since many systems are not publicly available or are not directly comparable. In the case of Glue-Nail, which has only recently become publicly available, the comparisons that are provided below are made using published times for certain queries. Aditi is a multi-user system which allows direct computation on disk-resident data, and so comparisons with a single user system for memory-resident queries seem pointless. As for LDL, [12] presents a comparison of CORAL with LDL, and indicates that for most simple queries, CORAL is usually significantly faster than LDL (an assessment largely echoed in [8]). As a result, the comparisons in this section are primarily made against against CORAL, although we include supporting information about Glue-Nail and LDL performance when appropriate. This section summarizes results detailed in [17], which includes not only range-restricted datalog queries, but queries involving functions, non-range-restricted queries, and negation.

The simple range-restricted datalog program

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), edge(Z,Y).
?- path(1,X), fail.
```

on `edge/2` predicates of the form

```
edge(1,2).  edge(2,3).   ...  edge(N,1).
```

can serve as an initial basis of comparison of XSB and CORAL. Figure 5 indicates the time needed to iterate 1000 times on cycles of length 8 to length 2k. The range of these data structures does not reflect any limitations in XSB or CORAL in handling larger relations.

XSB times appear to be about an order of magnitude faster than CORAL times. In figures 5,6 the lines labeled *CORAL-def* denote times for CORAL under default settings, while *CORAL-fac* denote times when the factoring option [10] has been turned on. It should be noted that the memory management and indexing of CORAL were somewhat more sophisticated than that of XSB when the comparisons were made, though these

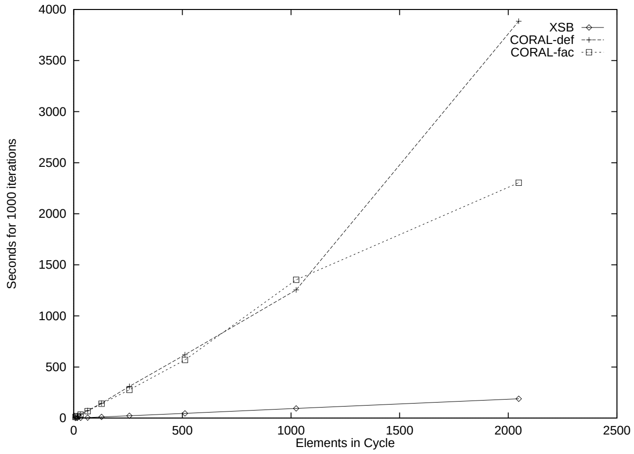

Figure 5: XSB and CORAL for cycles

factors should only account for a small percentage of the speed difference.

It might be argued that this comparison is potentially unfair in that the bottom-up algorithm returns only a single answer per iteration when `path/2` is evaluated over a chain. This leads to bias in favor of the tuple-at-a-time evaluation of XSB over the set-at-a-time evaluation of CORAL. To account for this bias, a second comparison can be made using "fanout" structures of the form

```
edge(1,1).  edge(1,2).  ...  edge(1,N).
```

These are trivial data structures all of whose answers can be found in the first iteration of a bottom-up fix point, when called with `?-path(1,X)`. Results shown in figure 6 indicate that the CORAL/XSB ratios for these data structures are similar to the ratios for the cycles.

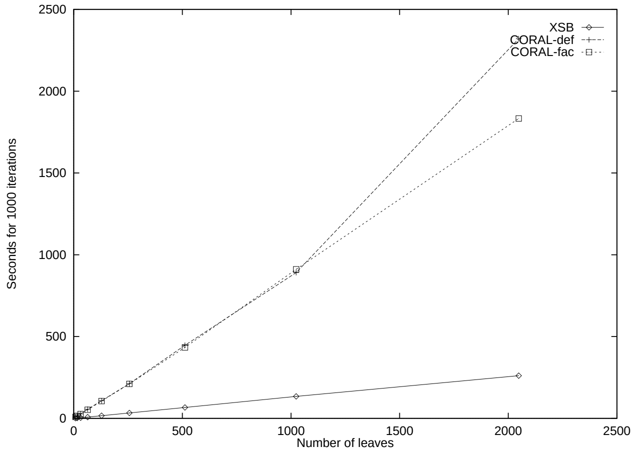

Figure 6: XSB and CORAL for fanout structures

Generally similar ratios hold when XSB is compared to CORAL for the linear right-recursive `path/2` predicate, the double recursive `path/2` predicate, the `same_generation/2` program and on the program

```
win(X) :- move(X,Y), \+win(Y).
```

This last program has the property of being (modularly) stratified iff move/2 is acyclic. The results from [17] indicate that XSB is at least an order of magnitude faster than CORAL for this program as well. For Glue-Nail, published results from [9] provide a preliminary means to compare its performance with that of XSB in win/1. [3] provides numbers that indicate that XSB's engine may be much faster than Glue-Nail for modularly stratified programs, while XSB's meta-interpreter, when run by Quintus Prolog, is comparable to Glue-Nail for both stratified and non-stratified programs.

As an alternative to the datalog programs, tests of CORAL against XSB were also performed for append/3, using both top-down techniques (SLD in XSB, pipelining in CORAL), and bottom-up techniques (SLG, and various compilation options in CORAL). In version 1.4 of XSB, table copy optimizations for ground structures are not complete. As a result, SLG is quadratic for this query. As expected, SLD was the fastest of all approaches. Pipelined CORAL was faster than SLG for lists of length greater than about 10, while CORAL compiled bottom-up with suggested optimizations was faster than SLG for lists of length greater than about 200 or so. Future versions of XSB will include table copy optimizations for ground structures. However, for non-ground lists, XSB will be quadratic in the length of the list, while CORAL will be quadratic in the number of variables in the list.

In [17] we present a fuller analysis of these results than we are able to present here. We hypothesize that one reason for XSB's efficiency for deductive database applications is its compilation into the relatively low-level SLG-WAM code, rather than into the more interpretive code of the other deductive database systems. To partially substantiate this claim, consider the time needed to join two relations in XSB, in Quintus Prolog, in LDL, in CORAL, and in Sybase. Table 4 gives approximate relative times for an indexed join for various systems. All data was in RAM, either under the control of UNIX or in the Sybase system buffer.

Sybase uses a fundamentally different paradigm than the other systems: all except Sybase are optimized for memory-resident queries, and none except Sybase have made special provisions for concurrency or recoverability. The two WAM-oriented systems are the fastest: for this problem Quintus is 3 times faster than XSB, mostly due to the fact that Quintus is written directly in assembler. These results also indicate the usefulness of separating out concurrency from a query engine, as well as the advantages of using specialized (e.g. indexing) techniques for memory-resident queries.

Quintus	XSB	LDL	CORAL	Sybase
1	3	8	24	100

Table 4: Approximate Relative Join Speeds

Clearly the join times from table 4 indicate that one factor in XSB's efficiency is its engine, which relies on the fact that data and rules are fully compiled. Indeed, XSB executes (restricted) SLG at the speed of compiled Prolog, as can be seen by comparing the path/2 predicate above against its right-recursive SLD form using edge/2 predicates representing both binary trees and chains. Prolog evaluation is linear for the queries to the chain and the tree since neither contains a redundant path. However, the left-recursive SLG derivation takes nearly the same time as right-recursive SLD for the chain and tree (about 20-25% longer), and it would, of course, terminate in the presence of cycles. The similarities in speed on the chain is especially significant since the SLG times include time taken to copy answer clauses to Table Space, and to abolish and reclaim Table Space at the end of each iteration.

Instruction profiles indicate that XSB spends the majority of its time in WAM instructions (rather than the new SLG-WAM instructions). We hypothesize that XSB is faster for datalog programs because the WAM itself is a more refined execution model than is available for set-at-a-time. Set-at-a-time evaluations will no doubt become more efficient. Still, creation of an execution model at the level of the WAM will prove difficult at best, since the WAM depends fundamentally on the tuple-at-a-time strategy. Furthermore, Prolog evaluation has progressed beyond the WAM. Optimizations based on native code generation [22], and on mode analysis, may bypass the WAM for heavily-used predicates, and have been shown to give an order of magnitude improvement for certain problems. There are also a number of datalog optimizations such as recursive selection and projection pushing, and factoring that XSB does not currently employ [10, 19]. It is not unreasonable to expect that full implementation of these techniques, some of which are already under way, will lead to significant speedup over the execution time of version 1.4 of XSB.

6 Conclusion

All deductive database systems must address both declarative and procedural issues: Glue-Nail does so with two different languages, while LDL and CORAL integrate a tuple-at-a-time (pipelined) mode with a set-at-a-time default. XSB's approach is to maintain the tuple-at-a-time strategy, but to extend it with bottom-up declarativity (SLG), and with a more expressive syntax (HiLog). The performance measurements displayed in the previous section may provide the strongest evidence for the viability of our approach.

There are other advantages to this approach: the rich and proven environment of Prolog can be included in XSB so that a tight and efficient coupling can be made of its procedural and declarative aspects. In the last half-decade or so, research into advanced database languages seems to have bifurcated into two distinct branches: object-oriented and deductive databases. A case can be made that the fundamental computational distinction between the two is that object-oriented databases use a strategy analogous to tuple-at-a-time whereas traditional deductive databases, use a set-at-a-time strategy in their bottom-up engines. The approach of XSB — which uses a tuple-at-a-time strategy in its bottom-up engine — indicates a possibility for a true integration of the two.

7 Acknowledgments

The authors would like to express their indebtedness to Weidong Chen, for his major contributions to SLG evaluation, and to Steve Dawson, who implemented transformational indexing. XSB also benefited from the involvement of Kate Dvortsova and Juliana Silva, and from suggestions of I.V. Ramakrishnan's research group. Not least, we would like to thank the many users who have helped us port our code, reported bugs, and sometimes even fixed them for us.

References

[1] T. Chen, I.V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In *JICSLP*, pp. 639–653, 1992.

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

[3] W. Chen, T. Swift, and D.S. Warren. Efficient implementation of general logical queries. SUNY at Stony Brook, 1993. Submitted.

[4] W. Chen and D.S. Warren. Query Evaluation under the Well-Founded Semantics In *PODS*, pp. 168-179, 1993.

[5] W. Chen and D.S. Warren. Computation of stable models and its integration with logical query processing. SUNY at Stony Brook, 1993.

[6] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2:76–89, 1990.

[7] M. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *SIGMOD 1993*, pp. 147–156.

[8] P. Hsu and C. Zaniolo. A new user's impressions on LDL++ and CORAL. TR, ILPS'94 Workshop on Programming with Logic Databases, 1993.

[9] S. Morishita. An alternating fixpoint tailored to magic programs. In *PODS*, pp. 123-134, 1993.

[10] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument reduction through factoring. In *VLDB*, pp. 173–182, 1989.

[11] T.C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 1989.

[12] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations, and logic. In *VLDB*, pp. 238–249, 1992.

[13] R. Ramakrishnan and J. Ullman. A survey of research on deductive database systems. University of Wisconsin, 1993. manuscript.

[14] K.A. Ross. Modular stratification and magic sets for datalog programs with negation. In *PODS*, pp. 161–171, 1990.

[15] K. Sagonas, T. Swift, and D.S. Warren. *The XSB Programmer's Manual*, 1993.

[16] H. Seki. On the power of alexander templates. In *PODS*, pp. 150–159, 1989.

[17] T. Swift and D.S. Warren. XSB performance measurement. SUNY at Stony Brook, 1993.

[18] T. Swift and D.S. Warren. The SLG-WAM Part I: Modularly stratified programs. TR SUNY at Stony Brook, 1993. In preparation.

[19] J. Ullman. Implementation of logical query languages for databases. *ACM TODS*, 10(3):289–321, 1985.

[20] J. Vaghani, K. Ramamohanorao, D. Kemp, Z. Somogyi, and P. Stuckey. Design overview of the Aditi deductive database system. In *7th Int'l Conf. on Data Engineering*, pp. 240–247, 1991.

[21] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3), 1991.

[22] P. van Roy and A.M. Despain. High performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, pp. 54–68, 1992.

[23] A. Walker. Backchain iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *JAR*, 11(1):1–23, 1993.

[24] D.S. Warren. Memoing for logic programs. *CACM*, 35(30):93–11, 1992.

[25] D.H.D. Warren. An abstract Prolog instruction set. TR 309, SRI, 1983.