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

In this paper we describe the XSB system, an in-memory deductive database engine. XSB began from a Prolog foundation, and traditional Prolog systems are known to have a number of serious deficiencies when they are 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 is provided that has the same properties. In addition XSB includes indexing capabilities greatly improved over those of standard Prolog systems. Also, it supports HiLog, which makes it a very flexible data modeling system [2].

The implementation of XSB derives from the WAM [24], an efficient Prolog engine. XSB inherits the WAM's efficiency and can take advantage of valuable 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, which forms the basic language of logic programming, is 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 again and again. 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, which may be appropriate for a programming language, but to be able to handle disk-resident data, set-at-a-time strategies have been deemed important.

The deductive database community has adopted datalog as an important 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 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]. And prototype deductive database systems, exemplified by LDL [6], Glue-Nail [7], and CORAL [12], have been designed and developed to incorporate these advances. We should mention 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 and a bottom-up evaluation strategy, it extends Prolog's SLD resolution in two ways: 1) adding tabling to make it 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 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 [24] (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 non-parameterized 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 *first-string indexing* [1], a variant of path-based indexing, which stores parts of clauses in a discrimination network. For I/O, XSB offers fast dynamic compilation along with linking and unlinking 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,

[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 and Motorola 680X0 chips have been tested; for operating systems, SUNOS, SOLARIS, IRIX, ULTRIX, LINUX, 386BSD, AMIGA-DOS, HP-UX, System V.3, and SCO Unix, and Mach have been tested. A port for Windows NT has been partially tested.

XSB performance, etc. 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 describes the XSB engine, including its functionality and how it is used. The HiLog syntax and its use are also covered there. Performance results are reported in section 5, which is followed by the conclusion.

2 Related Work

Table 2 summarizes aspects of some deductive database systems developed recently. The table was adapted from [13] which provides a wider comparison.

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	FO with Restricted HiLog extensions	Well-founded	Main-Memory
Syllog	Backchain Inferencing	Datalog	Stratified	Disk-Resident
XSB	SLG	FO with HiLog extensions	Well-founded	Main-memory

Table 1: Some Deductive Database Systems

Unlike XSB, most of the systems in table 2 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 2 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.

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 2, does not classify the various systems on the basis of their treatments of sets or aggregation. Most of the systems 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, etc. 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 2.

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. These features may be presented most clearly by describing the action of SLG first on definite programs, then on stratified programs, and finally on general logic programs.

On definite programs, SLG reduces to SLD resolution with memoing [25]. 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 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 this 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

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

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, and about three times slower than Quintus.[6]

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 directives, the cut (for SLG), 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, the encoding uses the apply symbol of arity $N+1$, where the first argument of `apply/(N+1)` is the functor of T, and the remaining N arguments are 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 **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 **L** such that

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

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'))`, `7`, `7(E)`.

[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.

[6] For these comparisons, Prolog-style hashing was used rather than XSB's first-string indexing. See [17] for details.

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]; here we describe only the syntax of the HiLog generalization of Horn clauses with negation. A *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: $\backslash+$ *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;

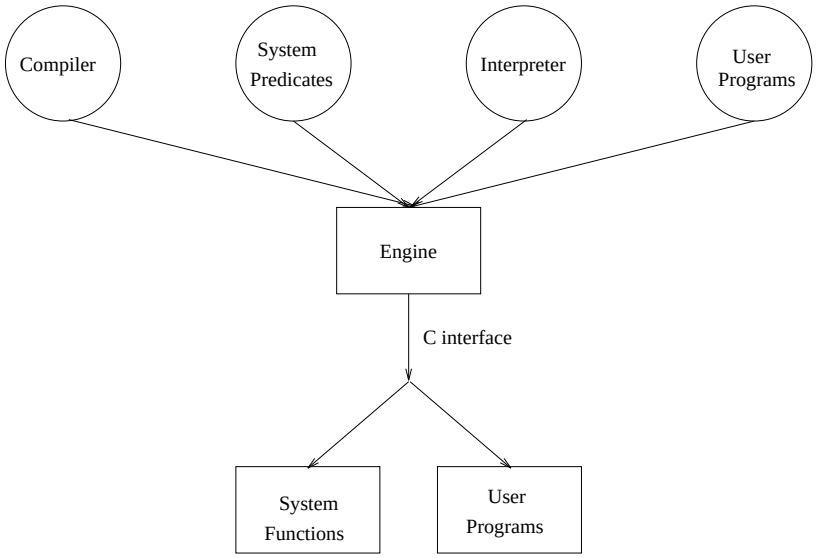

Figure 1: Overview of XSB Architecture

those 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. Code that needs to use full (i.e., nonstratified) SLG is (currently) executed using a meta-interpreter executing on top of the engine. The simplicity of the architecture allows the structure of the system to be easily understood and many 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.[7] As an alternate to compilation, XSB provides for *dynamic code*, predicates defined by facts (and rules) which can be modified during execution. This is the facility normally used for the extensional database. 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.[8] Seen from a database perspective, dynamic code provides a mechanism to insert, modify and delete data. Section 4.6 will discuss interfaces built upon them.

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 directives 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 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` directive is limited to the module in which it is given.

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

[8] 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 first-string indexing.

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 we want and then the `fail` causes the computation to 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 which indicates that `not_p(a,b)` is true. The interesting point here 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, and this may lead to significant computational savings. Some of the effects of cuts can be obtained in a bottom-up framework using *choice* operators [6].

This discussion of Prolog and cut is also relevant to SLG resolution, since it, too, is basically 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. So, in general, predicates being tabled must be completely evaluated in order to insure complete and correct answers. The XSB compiler will give an error if a simple static analysis of the program shows that a cut might close any partially computed tabled predicate.

However, in XSB we can determine that a table may have other subgoals depending on it. If there are no other users of the table, other than the one cutting over it, then the table can be deleted, without affecting the correctness of the algorithm. So we have added a new cut operator, `tcut/0`, which checks whether all tables being cut over can be safely freed, and if so, it frees them. If not, it 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.

Height	6	7	8	9	10	11
XSB / Default 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

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.

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.[9]

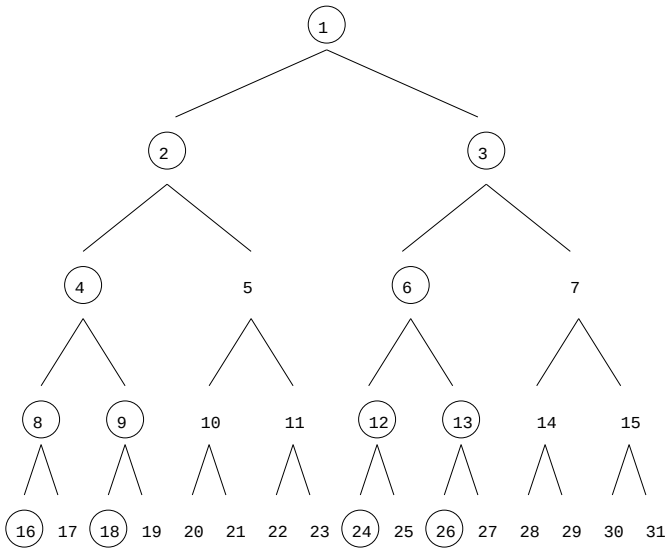

Figure 2: Calls to `win/1` over a binary tree

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. So in this case, where none of the subgoals are reused, using existential negation under SLG results in exactly the same nodes being visited as in SLDNF. In general, if it can be determined that negative goals will not need to be reused, as in this example, 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

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

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. XSB supports more flexible indexing which makes it better able to serve as a deductive database engine. Database applications require multi-field indexes as well as multiple distinct indexes on a single relation. Recall that XSB has two kinds of predicates: static predicates, which are compiled by the compiler but do not change during processing (unless they are reloaded in their entirety), and dynamic predicates, which are modifiable during execution one tuple at a time. XSB supports different kinds of indexing on these different types of predicates. In addition to indexes on user predicates, there are two uses of indexes 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.

For dynamic user predicates, those in which extensional database relations are normally represented, 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, programmer (or preprocessor) supplied directives may be used to indicate the desired options. Indexes can be constructed on any field, or set (of size three or less) of fields. Also any number of distinct indexes (of any kind) on the same predicate is supported. In addition, the size of the hash table to use is 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 indexes 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. All XSB hash-based indexing uses only the outer functor symbol of a given argument.

For static (or compiled) user predicates, the compiler accepts a directive that indicates the field on which a hash index is to be constructed. As an alternative to hash-based indexing, XSB offers *first-string* indexing for static predicates [1]. First-string indexing works by navigating a trie constructed at compile time. Tries are used as a means of discriminating strings of symbols. First-string indexing constructs a string from the preorder traversal of the heads of a predicate, where the traversal terminates as soon as a variable is encountered. It then constructs a trie out of the generated strings.

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).
```

which uses first-string indexing. The first strings constructed are $p_1 g_1 a_0 f_1$, and $p_1 g_1 a_0 f_1 a_0$, $p_1 g_1 b_0 f_1 1$, and $p_1 g_1$. After removing the first token, which is the predicate name, the trie looks as in figure 3.

The default indexing on compiled predicates is hash-based. Since the number of clauses is known by the compiler, it chooses an optimal hash table size. The user can provide declarations to cause first-string indexes to be built, either for single predicates or for all the predicates in a module.

Indexes 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.

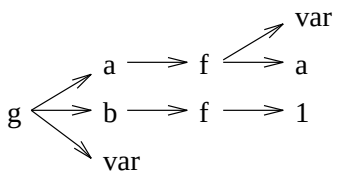

Figure 3: Trie for First-string Indexing

- 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 on 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). Using these interfaces, the developers of XSB have loaded relations of several hundred thousand tuples, and computed with them.

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 functor as if it were the first argument. The fragment

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

:- 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

```
apply(path(Graph),X,Y) :- apply(Graph,X,Y).
apply(path(Graph),X,Y) :- apply(path(Graph),X,Z), apply(Graph,X,Z).

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 first-string 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.

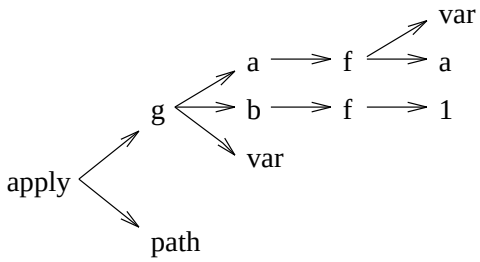

Figure 4: HiLog Discrimination Graph

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

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

apply_path(Graph,X,Y) :- apply(Graph,X, Y).
apply_path(Graph,X,Y) :- apply_path(Graph,X,Z), apply(Graph,X, Z).
```

In this form, the `path(Graph)/2` predicate is not much less efficient than if it were written in first-order syntax: there is extra overhead for the new argument (`Graph`) and there is overhead for the extra level of the discrimination graph of `apply/2`.

The flexible 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).

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

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

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 of sets can be extended to include intersection and union of sets. 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 very 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. The leftmost graph in 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.

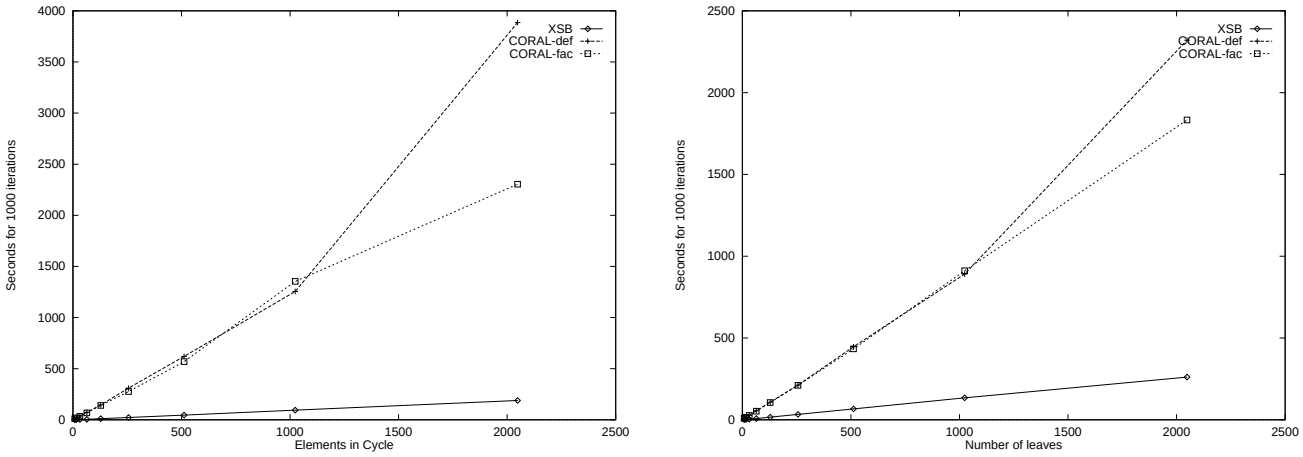

Figure 5: XSB and CORAL for cycles and fanout structures

XSB times appear to be about an order of magnitude faster than CORAL times. In figure 5 the lines labeled *CORAL-def* denotes time for CORAL under default settings, while *CORAL-fac* denotes 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 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 the right graph in figure 5 indicate that the CORAL/XSB ratios for these data structures are similar to the ratios for the cycles.

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. While Glue-Nail is not publicly available, published results from [9] provide a preliminary means to compare the performance of XSB to Glue-Nail 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 into 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 3 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.

Quintus	XSB	LDL	CORAL	Sybase
1	3	8	24	100

Table 3: Approximate Relative Join Speeds

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.

Clearly the join times from table 3 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 last 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 first-string indexing. XSB also benefited greatly from the involvement Kate Dvortsova and Juliana Silva, and from the 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, who have 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 *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 639–653. MIT Press, 1992.

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

[3] W. Chen, T. Swift, and D.S. Warren. Efficient implementation of general logical queries. Technical report, State University of New York at Stony Brook, 1993. Submitted.

[4] W. Chen and D.S. Warren. Towards effective evaluation of general logic programs. Technical report, State University of New York at Stony Brook, 1993. Manuscript.

[5] Weidong Chen and David S. Warren. Computation of stable models and its integration with logical query processing. Technical report, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, 11794-4400, Nov 1993. Submitted for publication.

[6] Daniette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim Naqvi, Shalom Tsur, and Carlo 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 *Proceedings of the SIGMOD 1993 Conference*, pages 147–156. ACM, 1993.

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

[9] S. Morishita. An alternating fixpoint tailored to magic programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.

[10] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument reduction through factoring. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 173–182. VLDB Endowment, 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 *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 238–249. VLDB Endowment, 1992.

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

[14] K.A. Ross. Modular stratification and magic sets for datalog programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 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 *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–159. ACM, 1989.

[17] T. Swift and D. S. Warren. XSB performance measurement. Technical report, State University of New York at Stony Brook, 1993.

[18] Terrance Swift and David S. Warren. The SLG-WAM Part I: Modularly stratified programs. Technical report, State University of New York at Stony Brook, 1993. in preparation.

[19] J. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 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 *Proceedings of the Seventh International Conference on Data Engineering*, pages 240–247, 1991.

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

[22] P. van Roy. Aquarius Prolog. *IEEE Computer*, 1990.

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

[24] David H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

[25] David S. Warren. Memoing for logic programs with applications to abstract interpretatino and partial deduction. *Communications of ACM*, 1992.