Enhancing a Genome Database Using the XSB Tabled Logic Programming System

Hasan Davulcu I.V. Ramakrishnan*
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{davulcu,ram}@cs.sunysb.edu

February 27, 1997

Abstract

LabBase is a generic database management system for implementation of laboratory information systems developed at the Center for Genome Research in the Whitehead Institute at MIT. It has become an important community database serving biological scientists engaged in genome research. Until now the language used to query LabBase has been non-recursive datalog without rules. Using the XSB Tabled Logic Programming System developed at SUNY, Stony Brook, we have now extended this query language into a full-fledged logic programming language. LabBase users can now use the full power of a logic programming language, especially rules and recursion, to compose complex queries. The XSB system extends and improves the evaluation strategy of Prolog. In particular it terminates for (recursive and non-recursive) Datalog programs thus enabling a smooth generalization of LabBase's query language. In addition it supports HiLog - a higher-order logic programming language, and by combing HiLog and tabling we generalize the limited aggregation features in LabBase.

Contact author: Hasan Davulcu
E-mail: davulcu@cs.sunysb.edu
Tel: (516) 632-9081
Fax: (516) 632-8334

*Supported in part by the NSF grants CCR-9404921, CCR-9510072, CDA-9303181, CDA-9504275 and INT-9314412.

1 Introduction

A major goal of the Human Genome Project [1] is to construct detailed physical maps of the human genome. A physical map is an assignment of DNA fragments to their locations on the genome. The maps consist of DNA markers spaced more or less evenly every few hundred bases along the genome.

The Center for Genome Research at the Whitehead Institute in MIT is engaged in several large-scale genome mapping projects. It is reported in [4] that these efforts will require the completion of over 2.5 million experiments, each of which requires several steps. Broadly speaking, the purpose of these experiments is to find short DNA sequences called *markers* and to determine or estimate their locations on the chromosomes of an organism, thereby generating a genome map.

To cope with this voluminous deluge of data, laboratory information systems are needed to store, retrieve and analyze large and valuable data sets as they are produced and organize the workflow of experiments. Database management systems (DBMS) constitute a critical component in realizing such an "automated laboratory notebook", a term commonly used in the genome research community.

As mentioned in [6], the database component requires powerful data models with nested, possibly recursive, data structures and rather general support for computed objects.

LabBase [7] is a database management system designed at MIT's Whitehead Institute of Biomedical Research, for managing laboratory data, *i.e.,* keeping track of laboratory materials, the experimental steps performed on them, and the results of these experiments. Since the data management requirements of a laboratory information system is somewhat non-conventional, LabBase provides special support for data management which goes beyond what is provided by a traditional DBMS. It is built on top of the ObjectStore persistent object system [5] and provides non-recursive datalog without rules as the query language. LabBase has been in operation for several years now and has grown into a community database, serving the needs of biological scientists engaged in genome research.

While the choice of the query language, dictated by its simplicity and ease of implementation, has so far been adequate there nevertheless appears a need now to enhance it [4] [5]. For instance, support for rules would increase the readability of long queries and recursion is necessary for handling queries requiring computation of transitive closure. Such queries are not only natural but at times necessary in the context of genome data.

In this paper we describe extensions to the LabBase query language that includes rules and recursion. Specifically, we extend the language into a full-fledged logic programming language that is executed by the XSB Tabled Logic Programming System developed at SUNY, Stony Brook [11]. The XSB system extends and improves the evaluation strategy of Prolog. In particular it terminates for (recursive and non-recursive) datalog programs thus enabling a smooth generalization of LabBase's query language. In addition it supports HiLog - a higher-order logic programming language [2]. By combing HiLog and tabling we generalize the limited aggregation features of LabBase. [1]

[1]The XSB - LabBase integration was done in collaboration with Steve Rozen, one of the principal designers of the LabBase System at Whitehead Institute.

The rest of this paper is organized as follows. To provide the appropriate context for the work described in this paper we provide an overview of the LabBase data model and its query language in the next section. In Section 3 we discuss extensions to the query language and implementation support for such extensions using the XSB system. We conclude in Section 4.

2 Genome-Mapping Database: An Overview

Laboratory information systems require powerful data models with nested, possibly recursive, data structures and rather general support for computed objects. LabBase, designed and implemented at MIT's Genome Research Center, is a database management system tailored to meet the needs of laboratory information systems [7]. It is designed to meet the requirements of managing laboratory materials, the experimental steps performed on them and the results of these experiments.

Data Model We review the conceptual data model of LabBase (details appear in [7],[8]). The primary abstractions provided by LabBase are *materials* and *steps*. Materials are the entities one works with in the laboratory, such as genetic markers, DNA fragments, etc. Steps represent operations that are performed on one or more materials and that generate experimental results.

Materials are grouped into *material kinds*. For instance, materials that represent the short DNA fragment is of kind `short_fragment` and those that represent the long DNA fragments is of kind `long_fragment`. Analogous to materials, steps are also grouped into *step kinds* such as `read_sequence_step`, `blast_step`, etc.

Apart from its *material kind* each material is also associated with a history (list) of steps that record the chronology of real-world operations performed on them. The static properties of a material like its type and id are associated with the step representing its creation. Materials are not denotable in the query language. They are retrieved by their id. For example, `marker_id(M,'D1234')` is true if there is a marker with `id D1234`, in which case `M` is bound to that marker.

A step is a set of pairs of the form (*tags*,*values*). Each tag is associated with a unique type. All steps are required to have tags *who* and *when*, which respectively record the person performing the step and the time at which the operation was performed. *e.g.*

```
sequence_step(material='D1723', sequence='ACTGGC',
              who='Rozen', when=1994:01:05).
```

In the above step, the *step kind* is `sequence_step` and, `'D1723'`, `'ACTGGC'`, `'Rozen'` and `1994:01:05` are the values of the `material, sequence`, `who` and `when` tags (respectively).

The Query Language A subset of datalog (non-recursive without rules) was chosen as the query language for LabBase. (See [8] for the reasons underlying this choice.) The language

supports sets and lists as values and a limited form of aggregates. The choice of datalog as a query language was motivated by its simplicity, ease of implementation, its expressive capabilities for ad-hoc data-dredging queries [9] and its extensibility *i.e.* domain specific builtin predicates can be readily added. Besides all of the above, datalog gracefully handles the structured data types in LabBase such as lists, list of lists and sets as function terms.

Queries can obtain both a *static* as well as a *historical* perspective on materials. The former is obtained by using predicates that are tag names, which act as accessors of the most recent value associated with a tag in the step history of a material. For example, to retrieve the `DNA sequence` of a `marker` with `marker id` `D1234` one would pose the query:

```
marker_id(M,'D1234'), sequence(M, Seq).
```

For *historical* perspective on materials one uses the `all_steps` predicate. Thus `all_steps(M,S)` binds `S` to all steps associated with material `M`. It is guaranteed that the bindings will occur in the historical order of the steps. For example to find all those markers that have any typing step following a mapmaker step, one would pose the query:

```
marker(M), all_steps(M,S1), all_steps(M,S2),
        mapmaker_step(S1), typing_step(S2),
        when(S1, W1), when(S2, W2), W2 > W1.
```

To restrict LabBase to the most recent step of a particular step kind, one can pose the following query:

```
marker(M), sequence_step(material=M, sequence=Seq,
                                        when=W).
```

In this query, `sequence_step` is the step kind, and `Seq` and `W` are bound to the values of the DNA sequence and `when` tag of the most recent `sequence_step` respectively.

LabBase has a small set of builtin predicates: *e.g.* for finding the cardinality of a `SET`, matching regular expressions against `strings`, doing aggregates and updates (insert, delete) etc.

3 Enhancing the Query Language

While the choice of restricted datalog as the query language has turned out to be adequate for handling most queries, the absence of recursion and rules has nevertheless been felt. For example, in one kind of genome mapping, called *genetic linkage mapping* [4], the *map-position* queries involve calculating a transitive closure over different forms of orderings for the markers over a chromosome. Furthermore the use of rules would enhance the readability of long queries. A full-fledged logic programming language as a query language would also reduce the need to code new builtin predicates in C++, and allow some applications to be coded entirely in the query language. We now describe our approach at extending the query language.

Recall that SLD's susceptibility to infinite looping and redundant subcomputations render Prolog unsuitable as a query language for deductive databases Tabulation methods have been proposed to address this and other shortcomings of Prolog. In particular the XSB Logic Programming system developed at SUNY, Stony Brook exemplifies such an approach. The XSB system implements SLG resolution [3], which is an extension of OLDT resolution [13] to handle general logic programs. In SLG, predicates are processed as *tabled* or *nontabled*. If no predicates are tabled, then evaluation proceeds as in SLD resolution. Thus for logic programming (LP) applications, SLG mimics Prolog evaluation. SLG is appropriate for deductive databases (DDB's), since evaluation based on SLG is asymptotically equivalent to 'Magic' transformations [12], and has polynomial data complexity for Datalog programs. Furthermore, SLG computes the major semantics proposed for nonmonotonic reasoning (NMR): directly computing the well-founded models [15], and computing (3-valued) stable models [10] after further processing. Thus the XSB system provides a computationally tight integration of LP, DDB's and NMR. The tightness of the integration is shown by the performance of the XSB system in the three areas mentioned above. As a logic programming system, XSB has performance comparable to several other popular Prolog implementations. Moreover, the XSB system has been shown to compute in-memory database queries about an order of magnitude faster than current semi-naive methods [2].

The first release of XSB was in June 1993; it is now in its sixth release. XSB has been ported to over a dozen platforms and has been installed in over a thousand registered sites. From a practical viewpoint availability of tabled logic programming systems such as XSB now makes it feasible to develop applications that were beyond the reach of Prolog-based logic programming systems. In particular extending the query language of LabBase is one such application since, XSB, unlike Prolog, will terminate on datalog programs thereby enabling a smooth generalization of LabBase's query language. The rest of this paper describes the mechanics of a query processor based on combining XSB and LabBase.

3.1 Integrated XSB-LabBase Query Processor

There were two primary concerns that had to be addressed in the integration. Firstly, LabBase is an evolving system and so it was important to render all changes completely transparent to users of the integrated system. Secondly, it was essential that the existing syntax of LabBase's query language be preserved by any extensions. To meet the above two requirements we adopted a conservative design. Specifically, our solution was to implement a meta-interpreter in XSB, which interprets dynamically loaded LabBase programs.

We define *flat datalog queries* as those comprised of LabBase and XSB builtins, and possibly other LabBase terms that provide static or historical perspectives on the laboratory data. To evaluate flat datalog queries the meta-interpreter makes a left to right scan over the query and for each term encountered, it checks the LabBase Data Dictionary to determine whether it is an application of a LabBase predicate or an XSB builtin. LabBase Data Dictionary stores the *mode* information for each builtin, *type* information for its arguments, all *material* and *step kinds*, and all the *tags* available for static or historical queries. If a predicate is found in the LabBase Data Dictionary, the meta-interpreter calls a syntax-

LabBase Data Type	XSB Data Type
date	atom
integer	integer
string	atom
float	float
list	list
list of lists	list of lists
DNA sequence	atom
material pointer	integer

Table 1: List of data type conversions.

directed *parser* which, while parsing the term also creates the corresponding data structures for its predicate name and its arguments in the LabBase engine. This parser returns a `handle` (a pointer) to the data structure representing the LabBase term and another pointer called `Bindings` which is a pointer to the list of uninstantiated arguments of the parsed term.

The XSB meta-interpreter uses `handle` to fork off evaluation of the LabBase term to the LabBase query evaluator. After the call is evaluated LabBase assigns values to the variables in the term and the meta-interpreter uses the `Bindings` pointer to collect these values and communicate them back to the XSB meta-interpreter. Otherwise, if there is no matching builtin in the LabBase Data Dictionary, XSB engine evaluates the predicate by invoking `call/1`.

LabBase has a rich set of data types represented as C++ classes. To interleave LabBase and XSB calls freely in a query and to communicate the bindings returned by each engine the following data type conversion builtins among LabBase's native data types and XSB's data types has been implemented. Table 1 lists these conversions: On scanning `Bindings`, appropriate calls are made to the datatype conversion builtins so that the necessary conversions are made for the values computed in either engine. For example, when an answer returns from an XSB term the bindings are communicated to the LabBase query engine. The LabBase Data Dictionary is checked to see the type of the current LabBase variable (e.g. *DNA sequence*) and the appropriate conversion builtin is called (e.g. atom to DNA sequence).

The meta-interpreter makes sure that the views of the LabBase and XSB are always consistent for the values of the arguments at every stage during the bottom-up evaluation. For example the following *flat datalog query* which interleaves XSB and LabBase builtins can now be made:

```
substring('GTTACACGC',1,5,X), /* LabBase builtin */
name(X,K),            /* XSB builtin     */
write(K),             /* XSB builtin     */
element(K,A),         /* LabBase builtin */
arg(2,f(1,2),J),      /* XSB builtin     */
```

```
   Q is J+A+1.          /* XSB builtin     */
```

The interface also has a clean implementation for rules and recursion. The meta - interpreter maintains a global stack of `Bindings` to compute rules and recursion. For each level in the top-down computation the global stack of `Bindings` is maintained so that the top element of the global stack of `Bindings` is always the correct bindings list to be used by the meta-interpreter to communicate the answers from the rule heads to the calling clauses. Whenever a new call to an intensional (IDB) predicate [14] is made, a new `Bindings` is pushed to the global stack of `Bindings` and it is popped after the meta-interpreter evaluates the new clause for the intensional predicate and communicates the computed answer back to the caller.

The current LabBase [7] implementation consists of approximately 9000 lines of C++ code and approximately 1000 lines of perl code. Persistent storage is provided by the Object Store Object-Oriented DBMS, which is essentially a persistent C++. Our interface is approximately 1500 lines of Prolog code and approximately 900 lines of C++ code.

The following program fragment illustrates rule-based query evaluation in the integrated system. The first `fill/0` call inserts the dictionary entries for materials, steps and tags and populates the database. The second query is a combination of LabBase and XSB builtin calls and intensional (IDB) predicate calls to `get_material/2` and `select_marker/3`.

```
fill :-
  insert(material_kind(dd_identifier=marker)),
  insert(dd_step_kind(dd_identifier=step1)),
  insert(tag(dd_identifier=tag1,type='INTEGER')).

fill :-
  insert(marker(marker_id='abc',who=steve,
                      when=1994:01:05:00:00:00)),
  insert(marker(marker_id='efg',who=steve,
                      when=1994:01:05:00:00:00)),
  marker_id(X,abc),
  insert(step1(material=X,tag1=3,who=steve,
                      when=1994:01:11:00:00:00)),
  marker_id(Y,efg),
  insert(step1(material=Y, tag1=6, who=steve,
                      when=1994:01:10:00:00:00)),
  write(done).

get_material(M,E) :-
 selectmarker(M),ith([1,2,3],0,E).

selectmarker(M) :-
  marker(M).
```

```
selectmarker(M,E,F) :-
  selectm(M),ith([7,8,9,10],E,F).


1) fill.
done

YES
2) ith([7,8,9,10],3,AS), get_material(M,E),
   substring('GTTACACAATTGGC',1,5,X),
   name(X,K), write(K), element(K,A),
   arg(2,f(1,2),J), Q is J+A+1, tag1(M,F),
   selectmarker(Q1,E,W).


AS=10
M=marker(abc)
E=1
X=TTACA
K=[84,84,65,67,65]
A=84
J=2 Q=87
F=3
Q1=marker(abc)
W=8
....
```

And the meta-interpreter backtracks to retrieve the rest of the 19 answers. Note that in the above query the LabBase builtin `ith/3` is used to break apart a list.

3.2 Aggregates

A notable area in which LabBase differs from Prolog is its handling of aggregates. The LabBase approach is illustrated by the following `gather_in_set` predicate:

`gather_in_set(`$term_1, ..., term_n$`, Element, Set).`

What the above predicate means is that: for each set of bindings for which $term_1, .., term_n$ is true, take the value bound to `Element` and make it an element of the `Set`.

In XSB we use HiLog [2] and tables to support aggregation. HiLog is a higher-order logic programming language with a higher-order syntax and a first order semantics. In HiLog one can manipulate predicates or the names of sets. Hence the results of a query can be collected in a set. So we can rewrite the above `gather_in_set` predicate as:

```
:- hilog results.
```

`results(Element) :-` $term_1$`, ...,` $term_n$`.`

The predicate `results` denotes a bag representing all the answers. (Note a bag may contain duplicates.) XSB provides aggregate predicates: `bagMin/2, bagMax/2, bagCount/2, bagAvg/2`, to compute the minimum, maximum, count and average, of a bag, respectively. Each aggregate predicate above assumes a hilog symbol as its first argument and returns the value of the aggregate function in its second argument. For example: `bagSum(results, TotalSum)` computes the sum of all the elements of the `results` and returns the sum in `TotalSum`. XSB also provides support for *"group by"* queries. For example the following aggregate query in LabBase finds those short fragments that in their most recent *blast_hit* have more than one element with probability `(P)` less than 10^{-6}:

```
short_fragment(S), blast_hits(S,Hits),
                count( element(Hits, Triple),
                       ith(Triple, 2, P),
                       P < 1.0e-06,
                       C),
                1 < C.
```

The above query can be rewritten in XSB by defining the parametrized predicate `fragments(Hits)` as a HiLog predicate to represent the bag of `fragments` whose `blast_hits` attribute is *Hits*. Then, XSB provides a predicate `bagCount/2` which can be used to sum up the elements in a named bag. An equivalent XSB query is:

```
:- hilog fragments.

moreThanOne(S, Hits, Count) :-
                  short_fragment(S),
                  blast_hits(S,Hits),
                  bagCount(fragments(Hits), Count),
                  Count > 1.

fragments(Hits)(Triple) :-
                  element(Hits, Triple),
                  ith(Triple, 2, P),
                  P < 1.0e-06.
```

Hence XSB provides a more general form of aggregation by taking advantage of HiLog and rules, and by providing a richer set of aggregate operators.

4 Conclusion

We described an integrated XSB-LabBase query evaluator that supports the full power and flexibility of logic programming as a database query language. The project was made possible by recent advances in table-driven evaluation of logic programming. Our system has been

in operation for several months now. While the performance appears to be acceptable there is room for improvement. Presently LabBase programs are not compiled into native WAM code. Moreover we create and maintain the corresponding LabBase data structures for each LabBase builtin. A tighter integration would require modifications to XSB's reader in order to detect, create and maintain the necessary data structures for LabBase builtins. In addition modifications to the LabBase architecture are necessary so that LabBase builtins can be redefined as XSB builtins and whenever a call is made to them, each LabBase builtin can extract the data structures needed for their execution directly from the XSB's heap. With these changes, we can compile the LabBase programs into native WAM code and thereby considerably enhance the performance of the integrated query evaluator.

References

[1] A. J. Bonner and E. Harlet. A flexible approach to genome map assembly. *Proc of the International Symposium on Intelligent Systems for Molecular Biology*, pages 161 – 169, August 1994.

[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, February 1993.

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

[4] N. Goodman, S. Rozen, and L. Stein. Requirements for a deductive query language in the mapbase genome-mapping database. *In Ramakrishnan, R., editor, Proceedings of the Workshop on Programming with Logic Databases In Conjunction with ILPS, Vancouver, B.C.*, pages 18 – 32, 1993.

[5] N. Goodman, S. Rozen, and L. Stein. Constructing a domain-specific dbms using a persistent object system. in m. atkinson, d. maier and v. benzaken, eds. *Proceedings of the Sixth International Workshop on Persistent Object Systems, Springer Verlag*, pages 526 – 541, 1994.

[6] N. Goodman, S. Rozen, and L. Stein. A glimpse at the dbms challenges posed by the human genome project. *Available in postscript and compressed postscript at http://www-genome.wi.mit.edu*, 1994.

[7] N. Goodman, S. Rozen, and L. Stein. Labbase: A database to manage laboratory data in a large-scale genome-mapping project. *IEEE Computers in Medicine and Biology*, 14:702 – 709, December 1995.

[8] N. Goodman, S. Rozen, and L. Stein. Labbase user manual. *Available in postscript and HTML at http://www-genome.wi.mit.edu*, 1995.

[9] N. Goodman, M. J. Daly S. Rozen, and Mary-Pat Reeve. Genome-map: Real-world test data and queries for logic databases. 1995.

[10] T. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4), 1990.

[11] K. Sagonas, T. Swift, and D.S. Warren. *The XSB Programmers Manual, Version 1.6.* Dept. of Computer Science, SUNY at Stony Brook, 1996.

[12] H. Seki. On the power of Alexander templates. *In ACM Symposium on Principles of Database Systems*, pages 150 – 159, 1989.

[13] H. Tamaki and T. Sato. OLDT resolution with tabulation. *International Conference on Logic Programming*, pages 84 – 98, 1986.

[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[15] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 1991.