Constructing a Domain-Specific DBMS using a Persistent Object System *†

Steve Rozen Lincoln Stein
Nathan Goodman
steve@genome.wi.mit.edu voice: 617-252-1923 fax: 617-252-1902
Whitehead Institute for Biomedical Research
One Kendall Square
Cambridge MA 02139

Constructing a light-weight domain-specific database management system (DBMS) is one way "...to design applications that effectively exploit...persistent technology". We have implemented a domain-specific DBMS, LabBase, on top of the ObjectStore persistent object system (which is basically a persistent C++). LabBase is tailored to the application domain of laboratory information systems: it is designed to record experimental steps and results in a high-throughput laboratory production line—for example, one of those operated as part of the Whitehead/MIT Genome Center's genome-mapping projects.

Given the task of representing the materials and experimental steps of a laboratory production line in C++, one could take two approaches:

1. Model each laboratory material and experimental step directly as a C++ class. A system constructed in this way can record the operations of a single laboratory production line.
2. Build a layer of abstraction in C++ in the form of a data definition facility based on *general* notions of laboratory materials and experimental steps. A system constructed in this way can be adapted to various laboratory production lines by supplying appropriate data definitions for the laboratory materials and experimental steps particular to specific production lines.

For LabBase we chose the second approach—that of creating a domain-specific DBMS—in light of our previous experiences working with a system based on the first approach.

We detail the considerations that led to this choice, and we describe LabBase's design and our experiences implementing it. We use our experiences with LabBase to illuminate the advantages and disadvantages of this approach to exploiting persistent technology. We also analyze how ObjectStore's particular characteristics shaped LabBase's design.

*This work was supported by funds from the U.S. National Institutes of Health, National Center for Human Genome Research, grant number P50 HG00098.

†To appear in *Sixth International Workshop on Persistent Object Systems*. Available as ftp://genome.wi.mit.edu/pub/steve/Y1994/labbase-design.ps.Z.

1 Introduction

LabBase is a light-weight domain-specific DBMS implemented on top of the ObjectStore persistent object system [1, 2, 3]. The implementation of LabBase is evidence that constructing a domain-specific DBMS is one workable solution to the problem of "[h]ow to design applications that effectively exploit...persistent technology".

Constructing a laboratory information system using a persistent C++ to provide database facilities presents a number of challenges. The most salient of these are:

- The need to provide multilingual access—an essential requirement of laboratory information systems—to a monolingual persistent object store.
- The need to accommodate constant re-engineering of the experimental production line as the laboratory seeks to lower costs and increase throughput.
- The need to view experimental results both historically—in the context of the sequence of experiments that produced the results—and statically—in terms of the current "final" result, even when experimental steps are repeated.

In addition, the solution of creating a domain-specific DBMS is important because it allows us, to some extent, to sidestep the problem of "[h]ow to build, maintain and operate large persistent applications". Creating a domain-specific DBMS allows us to build a relatively small persistent application (around 10,000 lines of C++ code) and lots of loosely coupled non-persistent applications. C++ classes are used to model the *generic* attributes of the application domain, while a data definition facility is used to specialize these attributes for specific applications within the domain. The additional layer of abstraction offered by a domain-specific DBMS also allows us to adapt more easily to re-engineering of the experimental production line.

The approach of creating a domain-specific DBMS offers some distinct advantages over the alternative of modeling the application classes directly in C++. We have, of course, also found disadvantages to using a domain-specific DBMS. We discuss both the advantages and disadvantages below in section 4, after having first described the application domain of laboratory information systems and how LabBase implements a domain-specific DBMS in ObjectStore.

1.1 Laboratory Information Systems

LabBase is designed to store data for laboratory information systems. A laboratory information system records the experimental steps performed and experimental results obtained during the operation of a laboratory production line. Examples of laboratory information systems are described in [4, 5, 6]. Laboratory information systems share much in common with classical information systems: in particular, the database component must provide a central repository of carefully administered, mission-critical data, and must integrate the operations of diverse software and human agents. A term commonly used in conjunction with the database component of laboratory information systems

is "automated laboratory notebook". The database is notebook-like because all experimental results are recorded in it, along with information about the experimental step itself, such as when it was performed and by whom. LabBase can be thought of as a generic automated laboratory notebook.

We emphasize that many genomic applications are not laboratory information systems, but rather are charged with integrating and publishing results from numerous independent laboratories. For example, both the worldwide sequence databases (e.g. GenBank [7]) and the various integrated organism-specific databases (e.g. the *Caenorhabditis elegans* database [8]) have very different requirements from laboratory information systems.

Laboratory information systems such as those used at the Whitehead/MIT Genome Center (Genome Center) typically involve several components in addition to a database component:

- External compute servers, for example servers to look for possible genes in DNA sequences.
- External data sources, such as the worldwide DNA sequence databases.
- Data analysis programs, such as programs to construct genetic maps from raw data on co-inheritance of particular DNA sequences. Often these programs are written at another laboratory or as part of a separate project in the same laboratory, and consequently they cannot be closely tied to the data representation in the database.
- Human data entry programs, often consisting of customized standard software. For example, at the Genome Center, much data entry is done on Excel spreadsheets, sometimes connected to a digitizing tablet.
- Interfaces to laboratory machinery, such as pipetting robots.
- Periodic progress reports and status reports that allow scientists to monitor production-line activity and detect possible problems with it.
- Interfaces—such as e-mail and WorldWide Web [9] servers—that publish released data over the internet.

It is the heterogeneity of these components that gives rise to the requirement of multilingual access to the database. For example, at the Genome Center, many of the status report programs are written in perl [10], as are many small "glue" programs that connect various independently developed components (e.g. customized Excel spreadsheets and external compute servers) to the database.

1.2 Why ObjectStore?

Our design of LabBase was heavily influenced by our experience developing and operating a previous laboratory database, MapBase [11, 12]. Like LabBase, MapBase is constructed in ObjectStore. However, unlike LabBase, MapBase models each experimental step as a separate C++ class. Because of the requirement for multilingual access, MapBase (like LabBase) offers a query language.

However, the LabBase query language (a non-recursive datalog) is much more general than MapBase's.[1]

We expect LabBase to replace MapBase as the database component of the Genome Center's mouse genetic-mapping and human physical-mapping production lines. A LabBase database is currently "shadowing" the MapBase database for the human physical-mapping project as a first step toward putting LabBase into production.

We chose to use ObjectStore for MapBase's successor for the following reasons:

- We have almost three year's experience operating ObjectStore applications and have confidence in its robustness.
- We have garnered considerable expertise in ObjectStore development and operation.
- We know the limitations of the current release of ObjectStore with respect to our applications, and we have techniques (developed for MapBase) for working around them.
- We know, based on our experience with MapBase, that we can obtain the necessary performance from ObjectStore.

We originally used a persistent object system for MapBase because we believed that that would be the best way to get the modeling expressiveness we would require. Among the object systems that were available in early 1991, we focused on the C++-based systems because it seemed that C++ was becoming the ascendant object-oriented language, and that consequently at least some of the C++-based systems would survive in the marketplace. Furthermore, there were at least three commercial C++-based systems available: ONTOS [13], VERSANT [14], and ObjectStore. Therefore we felt that if necessary we could port from one C++-based system to another with relatively little difficulty should one system fail in the marketplace or prove technically unsuitable. (We have ported a subset of MapBase to VERSANT with relatively little difficulty, which supports this rationale for choosing a C++-based system.)

Among ONTOS, VERSANT, ObjectStore, we chose ObjectStore as the most robust in early 1991.

2 Design and Rationale

Figure 1 represents the architecture of LabBase and its clients. The two processes labeled lbserv and lbback, together with the ObjectStore database, constitute LabBase, and the various clients are to the left. We first briefly describe LabBase's major components and their roles, and then discuss the rational for this design below.

The lbserv process is written in perl, and manages connections from multiple clients. It can buffer partial queries from several clients. Once a complete query

[1] The ObjectStore query language described in [2] and referred to as ObjectStore DML in [3] is a preprocessor extension to C++. Because queries have to be compiled into the ObjectStore client, this query language is not workable as a multilingual application program interface.

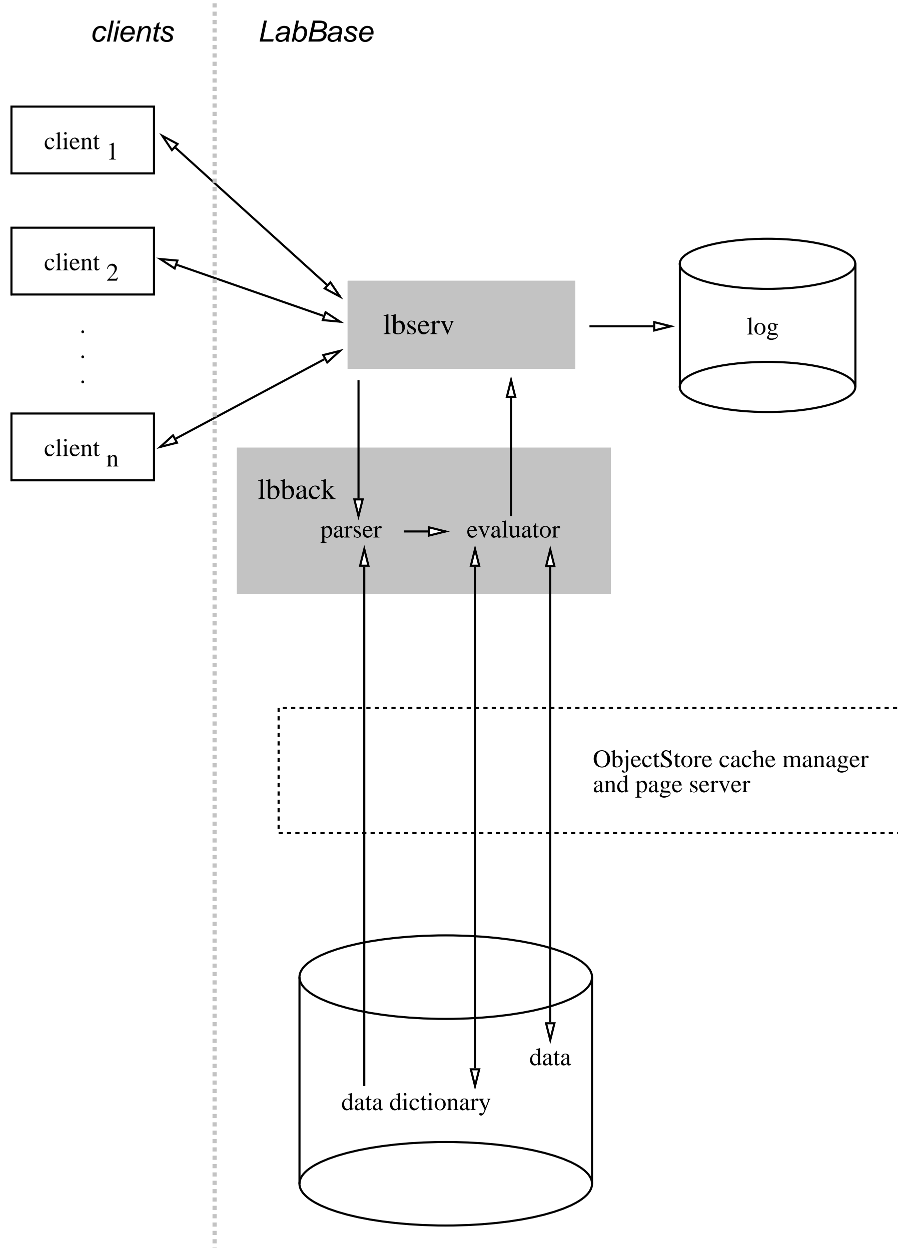

Figure 1: LabBase Architecture.

is available, lbserv forwards it to lbback, which executes it. lbserv also logs all transactions to a logical archive log. (By "logical" we mean that the log contains statements in the LabBase query language rather than page images, and by "archive" we mean that the log can be used to reconstruct the state of the database in the event of a disk-media failure or a dirty software failure that corrupts the database.)

The lbback process executes as an ObjectStore client written in C++ that implements the LabBase data definition facility and query language (discussed in more detail in section 2.1). Queries received from lbserv are first parsed (with the help of schema information stored in the data dictionary), and then executed by the evaluator. The evaluator (which requires information from the data dictionary) stores and retrieves database data, and can also update the data dictionary.

The parser and evaluator retrieve database objects from 3 persistent roots, which constitute the persistent part of the data dictionary. (The data dictionary also has a transient part, not shown in figure 1, which records the addresses of functions that implement built-in predicates.)

2.1 Query Language and Data Model

Recall that an important requirement for the database component of a laboratory information system is access from programs written in many languages (and running on diverse platforms—all of the laboratory information systems discussed in [4, 5, 6] use both Macintoshes and Unix machines).

In the absence of interlingual object standards, a straightforward way to provide multilingual access is to offer a query language, and this is the approach that we took in both LabBase and its predecessor, MapBase. The MapBase query language was an ad hoc construction, in which the "from" clause was always the set of all markers—short DNA sequences used for genome mapping—in the database, or certain extensionally defined subsets of these.

The MapBase query language was satisfactory as long as we only wanted to store information about markers, but it broke down rapidly when we had to record information about additional laboratory materials (other than markers) used in the Genome Center's human physical-mapping production line.

In order for LabBase to provide a more general query language than MapBase's we chose a non-recursive datalog. We selected a logic programming language for LabBase for the following reasons:

- The language is intended for use as an application program interface and for occasional data-dredging queries [15] posed by programmers. End-user queries are rare or non-existent, so we do not require the language to be end-user-friendly.

- We are not primarily language designers. Adapting a well-understood, flexible, and general language seemed preferable to expending effort inventing some new language.

- Datalog is syntactically and semantically simpler than other standard database languages that we might have selected, especially given LabBase's need to store historical data.

- An easy-to-implement, bottom-up evaluation of datalog queries seemed adequate to our needs for the near future.
- We hope to enhance the LabBase query language with rules and recursion, or replace it with a full-scale logic-programming implementation such as CORAL [16] or LDL [17].
- Even if, in the future, we use a language-independent object system, we would nevertheless still want associative queries.

Besides providing multilingual access, a query language such as LabBase's provides two interrelated advantages:

1. One can often express all or part of ad hoc data-dredging queries in an associative language that is closer to the application domain than C++. Furthermore, errors in ad hoc queries are trapped by the query language parser and evaluator, thereby increasing the robustness of the lbback process. Even though C++'s static type checking makes new C++ code more likely to be correct than the corresponding "traditional" C code, it is still difficult to avoid errors in new C++ code, especially in the management of heap-allocated storage.
2. The presence of a query language promotes a division of labor between the few developers that work on the implementation of lbback, and the relatively many that work on applications. The data definition facilities and query language present a more tractable interface than numerous C++ classes, and less chance for an application programmer to do harm—an important consideration when many applications are written by students.

These latter two considerations also convinced us to create our own modeling facilities and data dictionary as a source of meta data for parsing and evaluating queries. Thus LabBase provides a domain-specific data model.

LabBase's data model is based on the notions of *materials* and *steps*. All data about a material is entered in the context of a series of experimental steps, the material's *step history*. Each step is a mapping from *tags* (which are essentially attribute names) to *values*. Each tag is associated with a unique type. Only values that conform to the type of a tag can be associated with it in the database. (The query language itself performs only dynamic type checking.)

In the current implementation the atomic types are `BOOLEAN`, `INTEGER`, `DATE`, `FLOAT`, `STRING`, and `DNA_SEQUENCE`. For any types, $\alpha_1, \ldots, \alpha_n$, $n \geq 1$ whether atomic or complex, `LIST`($\alpha_1, \ldots, \alpha_n$) and `SET`($\alpha_1, \ldots, \alpha_n$) are complex types. These types characterize lists or sets (respectively) of tuples whose ith element has type α_i, $1 \leq i \leq n$. Elements of a `LIST` or `SET` can be retrieved with the `element` predicate. The constituent tuples have integer attribute labels, and tuples are decomposed with the subscripting predicate `ith`. For example, the query

```
element({[a,1],[b,2]},E),ith(E,1,I).
```

causes `I` to be bound successively to 1 and 2—though not necessarily in that order. (Curved braces enclose sets, and square braces enclose lists and tuples.) The following table shows some values and the types that describe them.

value	type
3.4	FLOAT
'D1Mit44'	STRING
1994:09:02:00:00:00	DATE
[1, 2, 5]	LIST(INTEGER)
{[1.0, 6] [3.4, 9]}	SET(FLOAT,INTEGER)
[[3],[],[5,7,9]]	LIST(LIST(INTEGER))

The `DATE` type is really a date-and-time type.

In addition to the atomic types and the type constructors (`LIST` and `SET`) already enumerated, there is also a type that describes materials, which can be used like an atomic type. Materials are not denotable in the query language, but they can be retrieved by their id. For example,

```
marker_id(M,'D1118').
```

is true if there is a marker with id `D1118`, in which case `M` is bound to that marker.

Data can be viewed as a static attribute of the material by requesting the value associated with the most recent instance of a particular tag in the history of a material. Data can also be viewed in the historical context of previous and subsequent experimental steps by requesting all the steps associated with one or more materials. Queries about step histories are useful in refining experimental protocols. For example, one could pose a query to find out how often some expensive operation is being performed on erroneous data (by detecting a correction step after the expensive operation in the step history). Queries to the system can freely intermingle both static and historical aspects.

Materials and steps have *kinds*, and these kinds, together with tags, can be used as predicates when querying LabBase. Additional built-in predicates provide negation, disjunction, some aggregates, boolean comparison operators, and so forth.

An example of a query that focuses on the most recent results relating to materials is the following, which finds the most recent DNA sequence associated with each marker:

```
marker(M),dna_sequence(M,S).
```

In evaluating this query, `M` is bound to each marker in the database, and, for each marker, m, `S` is bound to m's most recently entered sequence. (More than one DNA sequence could be associated with m if its DNA sequence is re-read because experimental results suggest that the initial DNA sequence is incorrect.) The predicate `marker` is, in addition, a material kind, and the predicate `dna_sequence` is a tag. For any material kind, κ, $\kappa(X)$ is true of all materials, X, of kind κ. Similarly, for a tag, τ, and a material, μ, the predicate $\tau(\mu,V)$ is true for that value, V, that is the most recent value associated with τ in μ's step history.

An example of a query that focuses on the history of steps associated with a material is the following, which finds all the DNA sequence steps (the steps that read the marker's DNA sequence) associated with marker `D1118`:

```
marker_id(M,'D1118'),all_steps(M,S),
dna_sequence_step(S).
```

In this query, M becomes bound to the marker with id D1118, S is successively bound to each step in D1118's step history, and dna_sequence_step restricts the results of the query to steps that are of kind dna_sequence_step.

Our guiding philosophy in implementing the LabBase query language has been to keep it, as much as *reasonably* possible, a subset of Prolog. In particular, negation is provided by a not predicate, which, as in Prolog, succeeds exactly when its argument (a term) fails.

Updates are carried out by predicates with side effects, much like Prolog's assertz. To insert a new material or a new experimental step one uses the predicate insert. The query

```
insert(marker(marker_id='D1118',
              who=steve,
              when=1994:04:01:16:45:11)).
```

creates a new marker with id D1118. Even the creating of a material must be annotated with

- the user (steve) responsible for the creation and
- a timestamp (1994:04:01:16:45:11) representing the valid time for the creation—the time at which the material should be considered in reality to have been created.

The query

```
marker_id(M,'D1118'),
insert(dna_sequence_step(material=M,
                         dna_sequence='CTGACCTG···GGTTA',
                         who=steve,
                         when=1994:04:01:16:48:23)).
```

inserts a dna_sequence_step onto the history of marker D1118. LabBase currently does not support deletes or updates of materials or steps.

A notable area in which LabBase is not a subset of Prolog is provision for aggregates. Our approach here is exemplified by the predicate gather_in_set:

gather_in_set($term_1 \ldots term_n$,V,Set)

is true if Set contains exactly those bindings of V that are among the bindings that make the query $term_1 \ldots term_n$ true.

As discussed in section 4, LabBase does not currently support rules, but the addition of rules would probably be the most beneficial enhancement we could make to LabBase, since it would increase the readability of long queries (some of which have more than 20 lines). More details on LabBase's data model and more example queries are provided in [18].

2.2 System Architecture and Implementation

The system architecture represented in figure 1 is partly dictated by the technical characteristics of ObjectStore. In particular, the choice of single process (lbback) mediating all access to the ObjectStore database is independent of the idea of creating a domain-specific DBMS. We previously used this design

in MapBase to work around the concurrency characteristics of ObjectStore Release 1. A notable deficiency of this design is that one client's long-running query excludes other queries for the entire duration of its execution. This exclusion results from the fact that lbback executes each query completely until it is done, and from the fact that there is a single lbback server for each database. ObjectStore's developers claim improvements in the area of concurrent performance for Release 3 [19], which we are now testing. Therefore we intend to reexamine the possibility of using multiple lbback servers interleaving their query executions by relying on ObjectStore's concurrency-control mechanism.

The roll-forward log (shown at the right in figure 1) to which lbserv logs all queries is another work-around, which is also independent of the idea of a domain-specific DBMS. ObjectStore offers no mechanism by which a periodic snapshot can be brought up-to-date by reading a log of transactions that followed the backup. ObjectStore's developers plan to provide this database amenity in its next major release [19], calling it "archive logging". At that time we might revise LabBase's implementation to rely ObjectStore-provided archive logging.

The implementation of lbserv in perl seems to impose some performance burden. But this choice of implementation language seems justified for the time being by the fact that we hope to replace much of lbserv with improved ObjectStore facilities in the future. The implementation of lbserv requires fewer than 1000 lines of code.

The implementation of lbback requires around 9,000 lines of C++ code, with about 50 classes. The parser and lexer are implemented in 500 lines of lex and yacc code [20].

We have taken some care to make it relatively easy to add new built-in predicates to lbback, and it should also be possible to add new types that can be associated with tags. We have not yet attempted any extensive performance tuning on LabBase, but its performance seems similar to MapBase's when they are both managing the same amount of data (around 60 Megabytes for MapBase and 80 Megabytes for LabBase). Our strategy with ObjectStore so far has been to make sure that the ObjectStore cache file (i.e., a paging file for lbback that contains database pages mapped into lbback's address space) is large enough to store the entire database. Making the cache file large is important because inserting database pages into the cache file seems to be a slow operation relative to paging by lbback against the cache file. It is also desirable to avoid paging out of the cache file, so we plan to buy additional physical memory as the database size grows. We expect eventually to have databases containing around 400 Megabytes of data.

We have also arranged for clustering of data in a few main segments: one for materials and their identifiers, one for step histories and steps exclusive of their associated values, and one for the values associated with steps.

Many of the clients communicate with LabBase using a perl library developed for the purpose. The library presents a set-at-a-time interface to perl clients, with all rows being stored in a single perl array. Each row can be transformed into an associative array (i.e. a hash table) implementing a partial map from variable identifier to binding.

3 Related Work

The idea of creating a domain-specific DBMS for the Genome Center's laboratory information systems came from two sources.

One source is the description of the target applications for ObjectStore. These are referred to as *CAx applications* in [2]. Example CAx applications include CAD (computer-aided design), CASE (computer-aided software engineering), CAP (computer-added publishing), and GIS (geographic information systems). It seemed that, just as one would not require the user of a CAD system to code a C++ class in order to model, say, a new kind of screw, perhaps one should not require the user of a laboratory information system database to code a C++ class in order to model a new kind of laboratory step. In other words, a CAD system provides an abstraction that is closer to the user's view of the world than C++, and it makes sense for our laboratory database to do the same.

The second source of the idea of a domain-specific DBMS is the extensive literature on extensible databases and database tool kits. Extensible databases (e.g. Starburst [21], Montage [22], GENESIS [23, 24]) are predicated on the notion of customizability for particular application domains, though the customizations are conceived primarily in terms of providing new storage structures, new query execution strategies, and new types and operations, rather than in terms of fundamental changes to the data model. For example [25] discusses using GENESIS to create "special-purpose database management systems" for applications such as CAD and textual databases, but the GENESIS data definition language is fixed.

The implementation of LabBase is in broad outline extremely similar to the implementation of a DAPLEX [26] DBMS in PS-algol [27] described in [28]. In this implementation, as in LabBase, an interpreted query language and data model were implemented in a persistent programming language.

4 Evaluation and Conclusion

Our experience with LabBase demonstrates some strengths of using a persistent C++ to create a domain-specific DBMS:

- A domain-specific DBMS can provide a query language. Having a query language seems to be a inevitable requirement for some application domains, including that of laboratory information systems. We did not find implementing a simple query language burdensome, though we hope in the future to replace the LabBase query language with a more general one. In particular, it would be useful to be able to define new predicates within the query language, rather than having to add new built-in predicates in the C++ code.

  Because we use perl so heavily, we do not suffer from impedance mismatch due to a set-at-a-time versus a tuple-at-a-time view: we retrieve entire query results into single perl arrays. We do lose type information (see below), though perl is so weakly typed that this loss is not an *added* disadvantage.

- As compared to using a generic DBMS, one can tune the data modeling facilities to the target application domain. For example, LabBase provides special support for storing histories of experimental steps, and for retrieving the most recent result of a particular sort.

  As compared to modeling an application directly in C++, the additional layer of abstraction presented by a domain-specific DBMS factors out commonality among applications in that domain, simplifying the data modeling task. This factoring can also reduce code size. LabBase offers more functionality than MapBase, with fewer than 1/3 as many lines of code (including lbserv). The factoring out of commonality among applications in a domain is challenging, but apparently well worth the effort.

- Application programs do not have access to the address space of the database, as they would if they were coded in C++ using ObjectStore's persistence mechanisms directly. This simplifies their view of the database contents, and excludes the possibility of database corruption due to low-level errors such as including dangling pointers in a database object.

- Because the domain-specific DBMS is coded in C++, one can extend its types and representations by adding new C++ classes. For example, since DNA sequences can be represented using 3 bits per base, one could design a representation that uses less memory than if one were to store them as strings of characters.

- One can also add new built-in predicates. For example, LabBase provides *reverse complementation*, a common operation on DNA sequences. (To take the reverse complement of a DNA sequence it is reversed, and the bases are substituted by the Watson-Crick paring rules: G↔C, A↔T. E.g., the reverse complement of CCATG is CATGG.)

Our experience with LabBase also reveals some weaknesses of using a persistent C++ to create a domain-specific DBMS:

- There can be a loss of type information at the interface between the query language and the application program. In particular, it is not clear how to move objects or object references from the DBMS to an application program.

- To the extent that the data model is less expressive than that of C++, there will be a loss of modeling ability. This loss is mitigated by the fact that a DBMS extender could add new types to the domain-specific DBMS.

- Even though it is possible to tune representations (such as those of DNA sequences), it is not easy to figure out how to provide tuning *options* for different uses of the domain specific DBMS. For example, we have not been able to decide what clustering options LabBase should provide.

- Even though it is not burdensome to implement a simple query language like LabBase's non-recursive datalog, implementing a full-scale query language, with recursion, unification, and optimization, is a different matter.

- There is a danger of devising idiosyncratic data models and query languages.

In addition, a domain-specific DBMS implemented in a persistent C++ derives many of its operational characteristics from the underlying persistent object system. For LabBase this has meant good performance provided we keep the entire database in lbback's virtual memory, but it has also meant that we had to give up native ObjectStore concurrency and implement our own roll-forward archive logging. Presumably using another persistent object systems would provide a different constellation of characteristics, dictating a system architecture different from LabBase's.

Persistent object systems are much more diverse than relational DBMSs in terms of data modeling approach, query language, and operational characteristics. Therefore we can only speculate about how we would construct a system like LabBase using a language-independent persistent object system such as O_2 [29], Gemstone [30], or Thor [31, 32]. We are currently attempting a prototype reimplementation of LabBase in Thor, partly with the goal of answering this question.

We think that creating generic abstractions for materials, histories of experimental steps, and for retrieving the most recent experimental result of a particular sort would still be valuable. In the most extreme case, one could do away with the LabBase query language in favor of the query language of the object system (if it provides one). In this case, LabBase would resemble so-called "third-party" packages (e.g. accounting packages) that are sometimes built on top of a relational DBMS. Of course, unlike the case with third-party packages built on top of today's relational DBMSs, considerable customization or extension of the object system would be feasible, and one could use this to create the generic abstractions discussed above.

Another possibility would be to keep the LabBase query language, but to take advantage of an object system's ability to create new classes from running applications and use the object system's run-time type data ("metadata") as the data dictionary. Thus, for example, each LabBase material kind could be represented as a subclass of a `Material` class.

There are a number of enhancements we would like to see for LabBase:

- Additional formal analysis of the notions of materials and steps, and how they interact, together with some methodologies for developing LabBase schemas.

- Support for constraints on step orders, to flag unusual orders as possible errors.

- A full-fledged logic programming language as a query language. This would reduce the need to code new built-in predicates in C++, and allow some applications to be coded entirely in the query language.

LabBase appears to be a success within the Genome Center. Our hope is that it will also be useful to other laboratories.

Acknowledgments Barbara Levy provided editorial assistance. Andre Marquis has helped in the implementation of LabBase. Mary-Pat Reeve, the first

non-developer user, offered useful suggestions that will be incorporated into LabBase. Tony Bonner provided valuable input on the design of the LabBase query language. We have incorporated numerous useful suggestions offered by the anonymous referees.

References

[1] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore database system," *Communications of the ACM*, vol. 34, pp. 50–63, Oct. 1991.

[2] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara, "Query processing in the ObjectStore database system," in *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (M. Stonebraker, ed.), pp. 403–412, June 1992.

[3] Object Design, Inc., 25 Burlington Mall Rd., Burlington MA 01803-4194, USA, Manual set for ObjectStore Release 3.0 for UNIX Systems, Dec. 1993.

[4] A. R. Kerlavage, M. D. Adams, J. C. Kelly, *et al.*, "Analysis and management of data from high-throughput expressed sequence tag projects," in *Proceedings of the 26th Annual Hawaii International Conference on System Sciences* (T. N.Mudge, V. Milutinovic, and L. Hunter, eds.), vol. 1, pp. 585–594, IEEE Computer Society Press, Jan. 1993.

[5] S. P. Clark, G. A. Evans, and H. R. Garner, "Informatics and automation used in physical mapping of the genome," in *Biocomputing Informatics and Genome Projects* (D. W. Smith, ed.), pp. 13–49, Academic Press, Inc., 1994.

[6] L. Stein, A. Marquis, E. Dredge, *et al.*, "Splicing UNIX into a genome mapping laboratory," in *USENIX Summer 1994 Technical Conference*, pp. 221–229, June 1994.

[7] C. Burks, M. Cassidy, M. J. Cinkosky, *et al.*, "GenBank," *Nucleic Acids Research*, pp. 2221–2225, 1991.

[8] R. Durbin and J. Thierry-Mieg, "A *C. elegans* database," 1991. Documentation, code and data available from anonymous `ftp` servers at `lirmm.lirmm.fr`, `cele.mrc-lmb.cam.ac.uk` and `ncbi.nlm.nih.gov`.

[9] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann, "World-Wide Web: The information universe," *Electronic Networking: Research, Applications and Policy*, vol. 2, no. 1, pp. 52–58, 1992. Also available at `http://info.cern.ch/hypertext/WWW/Bibliography/Papers.html.`

[10] L. Wall and R. L. Schwartz, *Programming perl.* O'Reilly & Associates, Inc., 1990.

[11] N. Goodman, "An object oriented DBMS war story: Developing a genome mapping database in C++," in *Modern Database Management: Object-Oriented and Multidatabase Technologies* (W. Kim, ed.), ACM Press, 1994.

[12] N. Goodman, S. Rozen, and L. Stein, "Requirements for a deductive query language in the MapBase genome-mapping database," in *Applications of Deductive Databases (tentative)* (R. Ramakrishnan, ed.), Kluwer, 1994. In press. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps`.

[13] ONTOS, Inc., Three Burlington Woods, Burlington MA 01803, USA, *ONTOS DB 2.2 Developer's Guide*, Feb. 1992.

[14] Versant Object Technology Corporation, 4500 Bohannon Drive, Menlo Park CA 94025, USA, *VERSANT Object Database Management System Release 2 System Manual*, July 1993. Part Number 1003-0793.

[15] S. Tsur, "Data dredging," *Data Engineering*, vol. 13, Dec. 1990.

[16] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, "CORAL–control, relations and logic," in *Proceedings of the 18th International Conference on Very Large Data Bases* (L.-Y. Yuan, ed.), pp. 238–250, Aug. 1992. A longer version is available from the authors.

[17] S. Tsur and C. Zaniolo, "LDL: a logic-based data language," in *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 33–41, Aug. 1986.

[18] S. Rozen, L. Stein, and N. Goodman, *LabBase User Manual.* Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-manual.ps`.

[19] P. O'Brien, "R3 & R4 product directions overview," Mar. 1994. Talk presented at ObjectStore Northeast users group meeting.

[20] T. Mason and D. Brown, *lex & yac.* O'Reilly & Associates, 1991.

[21] L. Haas *et al.*, "Starburst mid-flight: As the dust clears," *IEEE Trans. on Knowledge and Data Eng.*, vol. 2, Mar. 1990.

[22] M. Stonebraker, "Object-relational data base systems," 1993. Distributed with Montage (now called Illustra) sales literature.

[23] D. S. Batory, J. R. Barnett, J. F. Garza, *et al.*, "GENESIS: An extensible database management system," *IEEE Trans. on Software Eng.*, vol. 14, pp. 1711–1730, Nov. 1988.

[24] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. on Software Engineering*, vol. 1, Oct. 1992.

[25] D. S. Batory, T. Y. Leung, and T. E. Wise, "Implementation concepts for an extensible data model and data language," *ACM Trans. on Database Syst.*, vol. 13, pp. 231–262, Sept. 1988.

[26] D. W. Shipman, "The functional data model and the data language DAPLEX," *ACM Trans. on Database Syst.*, vol. 6, no. 1, pp. 140–173, 1981.

[27] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison, "An approach to persistent programming," *The Computer Journal*, vol. 26, no. 4, pp. 360–365, 1983.

[28] K. G. Kulkarni and M. P. Atkinson, "Implementing an extended functional data model using PS-algol," *Software—Practice and Experience*, vol. 17, pp. 171–183, Mar. 1987.

[29] O. Deux *et al.*, "The O_2 system," *Communications of the ACM*, vol. 34, pp. 34–48, Oct. 1991.

[30] P. Butterworth, A. Otis, and J. Stein, "The GemStone object database management system," *Communications of the ACM*, vol. 34, pp. 65–77, Oct. 1991.

[31] B. Liskov, M. Day, and L. Shrira, "Distributed object management in Thor," in *Distributed Object Management* (T. Ozsu, U. Dayal, and P. Valduriez, eds.), Morgan Kaufmann, 1994.

[32] M. Day, R. Gruber, B. Liskov, and A. C. Myers, "Abstraction mechanisms in Theta," 1994. Submitted for publication.