

# LabBase: A Database to Manage Laboratory Data in a Large-Scale Genome-Mapping Project<sup>1 2</sup>

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

## Abstract

The central task of managing laboratory data is keeping track of laboratory samples, the experimental steps performed on them, and the results of these experiments. This task engenders several challenges, namely:

- The need to accommodate frequent changes to laboratory protocols.
- The need to provide data access to programs written in multiple languages and running on heterogeneous hardware.
- The need to represent unusual data types, such as DNA sequences, with specialized behavior.
- The need to view data in both static and historical perspectives. The static perspective deals with the current state of knowledge about a material such as the “sequence of a DNA fragment” or the “chromosome from which a DNA fragment was obtained”. The historical perspective deals with the history of experimental steps, such as “for what percentage of DNA fragments has the sequence of constituent bases been read more than once?” Such historical queries are crucial to understanding and refining laboratory workflow.

To meet these challenges, we designed and implemented LabBase, a functionally specialized DBMS for use in laboratory information systems.

---

<sup>1</sup>This work was supported by funds from the National Institutes of Health, National Center for Human Genome Research, grant number P50 HG00098 and from the U.S. Department of Energy under contract DE-FG02-95ER62101.

<sup>2</sup>To appear in IEEE Engineering in Medicine and Biology, Nov./Dec. 1995.

LabBase is freely available under a license that permits redistribution of source and object code.

## Introduction

The Whitehead/MIT Center for Genome Research (CGR) is engaged in several high-throughput genome-mapping projects, including physical mapping of the human genome and genetic-linkage mapping of the mouse genome. The scale and complexity of the laboratory workflows for these projects make a laboratory information system a necessity. For example, the human physical mapping project has performed over 1.2 million experimental steps to date, many of which involve numerous biochemical assays. The result is a physical map containing over 13 thousand sequence-tagged-site (STS) markers, with a near-term goal of 15 thousand markers. These markers will likely be used as a starting point for producing a higher resolution map that in turn will provide the raw material for sequencing the human genome.

LabBase is a database management system (DBMS) tailored to the needs of laboratory information systems that support such projects. It is designed to make it easy to keep track of laboratory samples, the experimental steps performed on them, and the results of these experiments. LabBase is *functionally specialized* because it provides special support for the requirements of managing laboratory data over and above what would be provided by a generic, off-the-shelf DBMS. To meet the challenges discussed in the abstract, LabBase provides special support

- I. for accommodating frequent changes to laboratory protocols,
- II. for providing data access to programs written in multiple languages and running on heterogeneous hardware,

III. for representing types—for example arrays and DNA sequences—and associated operations that are not supported in current relational DBMS, and

IV. for viewing data in both static and historical perspectives.

Laboratory information systems provide central repositories of mission-critical data that are updated concurrently. Therefore they require concurrency control and atomic recovery from software and hardware failures, and LabBase provides these services.

LabBase is designed to be only one component of a laboratory information system.

Other components include

- application programs for entering data,
- tools for processing data as part of the laboratory production line,
- status reports on laboratory activity, and
- forms-based query interfaces and an HTTP-based [1] browser.

These components are implemented in other programs that function as LabBase clients in a client/server architecture. LabBase communicates with its clients by means of a query language and application program interface. Programmers and database administrators also use the query language to pose occasional ad hoc data-mining queries.

The strategy of *componentry*—of assembling systems from relatively independent components—offers several salient advantages. In the context of a particular laboratory, developers can choose the most suitable implementation language and hardware platform for each client, and can replace or enhance individual client programs without disrupting either the central data store or other clients. In a broader context, componentry fosters software sharing and co-development, because a single component of an information system, for example a particular analysis program, user interface, or DBMS, is typically reusable, whereas an entire information system rarely is.

LabBase is used in laboratory information systems operated at CGR [2]. Other, similar systems used in genome research are described in [3, 4, 5]. In addition, laboratory information systems—dubbed *LIMS*, an acronym for “laboratory information management systems”—are used in analytical laboratories such as those used for environmental testing or quality control in pharmaceuticals manufacturing [6, 7]. We hypothesize that a key difference between information systems used in analytical laboratories and those used in genome laboratories is that the laboratory protocols used in genome laboratories are experimental, with the consequence that laboratory information systems for genome research must be extremely flexible in adapting to changes in the laboratory protocols. Another difference is that genome laboratories must often record results of interactions between samples. At a more abstract level, however, information systems in both genome-research and in analytical laboratories share a focus on tracking samples and recording the results of experiments performed on them.

LabBase’s design focuses on the essentials of DBMS support for laboratory information systems, as outlined in items I through IV above, and reflects experience garnered during three years of operating a predecessor, MapBase, at CGR [8, 9]. Guided by these essentials, and wishing to test and employ a minimal system in production as quickly as possible, we made pragmatic decisions about the focus of our implementation effort.<sup>3</sup> As a result, LabBase lacks a number of conveniences that we plan to add to it and that we discuss below as we present the existing LabBase design.

---

<sup>3</sup>Like Richardson, Eppig, and Nadeau [10], we have learned that we must plan for evolution and avoid over-design. Accommodating evolution of laboratory protocols was a primary impetus for the creation of LabBase: we decided that MapBase was too monolithic to evolve gracefully. In planning for evolution, componentry is a guiding principal, because it is far easier to changes a system built from components than a monolithic system. To avoid over-design, we have adopted a strategy of implementing software with minimal functionality in order to validate its usefulness to users. Componentry also helps us avoid over-design, because it lets us build simple components with circumscribed functionality; if in fact a component proves useful in practice, we can add to its functionality based on observed requirements.

The current implementation consists of approximately 11,000 lines of C++ code [11] and approximately 1000 lines of perl code [12]. Persistent storage is provided by, alternatively, ObjectStore [13] or the Texas Storage Manager [14].<sup>4</sup> Both storage managers, from the programmers point of view, are slight extensions to C++ that allow objects to be allocated on a *persistent heap* (via an overloaded `new` operator), so that they persist between program executions. Other than creation, all operations on persistent objects are identical to operations on ordinary non-persistent objects, so little user code needs to be aware of whether an object is persistent. Both storage managers provide an atomic checkpointing facility that guarantees that the persistent heap is left in a consistent state if a program crashes (even in the middle of a checkpoint).

The remainder of this paper focuses on an external view of LabBase: its data model, query language, and application program interface. LabBase's implementation is discussed in more detail in [15], and an effective workflow management scheme built on top of the LabBase application program interface (API) is described in [16].

## Running Example

We use a highly simplified form of the CGR's human physical-mapping production line as a running example to aid exposition. This example production line involves the following operations:

Op-1. Read the DNA sequence of a short fragment of human DNA (about 300 bases long).

(For those with a background in molecular biology, these small fragments are

---

<sup>4</sup>LabBase is available on two storage managers because, although we originally developed it on top of ObjectStore, we wanted to be able to provide it to potential users who could not afford ObjectStore's licensing fees. We continue to operate LabBase on top of ObjectStore because we have only a few month's experience with the Texas version, and because ObjectStore provides facilities for improving run-time locality of reference, which is a critical determinant of performance when databases become significantly larger than main memory.

small-insert clones.)

- Op-2.** Using the computer program BLAST [17], check whether the DNA sequence of the short fragment is similar (in technical terms, “homologous”) to any in the GenBank worldwide sequence database [18]. The short fragment will be discarded if it is found to be part of a sequence that occurs in multiple parts of the human genome. Otherwise, information about similarity to a known sequence might be of intrinsic interest; for example, the short fragment might be similar to a gene known from another organism.
- Op-3.** Choose *primer pairs*—two DNA sequences (each about 20 bases in length) that are contained in the short DNA fragment, that are separated by a minimum number of bases, and that meet certain other conditions. Relatively large quantities of these short primer sequences can be synthesized and used to prime a chemical reaction called polymerase chain reaction (PCR) that allows us to detect whether an arbitrary piece of DNA contains the primer pair.
- Op-4.** Check for presence of primer pairs in a long fragment of human DNA—a yeast artificial chromosome (YAC)—about  $10^6$  bases in length. (Such fragments are far too long for existing technology to read their DNA sequence directly.)

These operations would then be followed by an analysis to determine a partial order among the primer pairs based on the overlap of long fragments that share primer pairs. This information basically constitutes a physical map, and can be used, for example, as a starting point for the search for a gene. In reality, there are a number of additional operations between operations **Op-1** and **Op-4**, and operation **Op-4** itself actually involves many sub-operations.

## Data Model

The LabBase data model is designed to support the focus on sample tracking and recording of experimental results that is the hallmark of laboratory databases. There are two key notions in the LabBase data model:

*Materials* represent laboratory samples such as short and long DNA fragments (to use materials from our running example). More generally, materials play the role that “entities” or “objects” (two roughly equivalent terms) play in semantic data models [19]. Materials are grouped into *material kinds*. For example, materials that represent the short DNA fragment would have kind `short_fragment`, and those that represent the long DNA fragments would have kind `long_fragment`. When we speak of a material, we mean a material instance.

*Steps* represent operations that are performed on one or more materials and that generate experimental results. Steps would represent each of the operations `Op-1` through `Op-4` in the running example. Examples of experimental results would include the DNA sequence produced by `Op-1` or the GenBank DNA sequences found in `Op-2`. Analogously to materials, steps are grouped into *step kinds*. For example, instances of operation `Op-1` would be represented by steps of kind `read_sequence_step`, instances of operation `Op-2` by steps of kind `blast_step`, and so on. When we speak of a step, we mean a step instance.

Formally, a step is a set of pairs of *tags* and *values*, both of which we discuss in detail below. No tag can appear more than once in a step. Material kinds resemble “entity sets” or “object types” in semantic data models, and tags resemble attributes. There are, however, two important differences:

1. Tags are parts of steps rather than parts of materials. The static properties of a material are associated with the step representing its creation (and cannot be associated with any other step).

2. A material or step kind does not specify the set of tags that must or must not be associated with a material or step of that kind. This lack of constraints on the information that LabBase can record about materials and steps is motivated by the need to accommodate schema changes induced by frequent reengineering of laboratory workflows. It is worth observing that ACEDB [20], the most widely used system for storing and viewing genomic information on particular organisms, employs a related approach to accommodate schema evolution: ACEDB object types restrict only the tags that are allowed; any tag can be absent.

We plan to augment LabBase with an optional capability of requiring or forbidding a step or material to have certain tags.

## Tags and Values

As mentioned above, the role of tags is analogous to the role of attribute labels in semantic data models, and a step is a set of tag-and-value pairs, in which each tag can appear at most once. Each tag is represented by an alphanumeric identifier, and has a *type* that describes the values that can be legally associated with that type. For example, a step of kind `blast_step` (recording the results of executing `Op-2`) might consist of the following tag-and-value pairs:

```
who           'lou'
when          May 23, 1994 at 10:24
tested_short_fragment  the short DNA fragment with id PB223
blast_hits    {'V00748', 'Mouse pseudogene ...', 5.4e-07},
              ['J00405', 'Mouse MHC class I...', 5.4e-07],
              ['X03822', 'HSAG-1 middle rep...', 2.6e-06]}
```

All steps are required to have tags `who` and `when`, which respectively record the worker taking responsibility for the step and the date and time at which the operation was performed. The type of tag `who` is `STRING` and the type of tag `when` is `DATE`.

Tag `tested_short_fragment` has type `MATERIAL`, and the associated value is an object reference (i.e. a pointer) to a particular `short_fragment` instance. Each material has a *step history*. A material’s step history contains a step exactly when the material is the value of a tag-and-value pair in that step. Step histories confer two benefits. At a physical level they act as indexes that allow the LabBase query processor to quickly retrieve all the steps corresponding to experiments performed on a particular material. At a logical level the concept of step histories underlies query-language constructs that make it easy to retrieve experimental results related to a particular material without having to resort to the numerous joins that would likely be necessary in a relational system representing the same information. We discuss these query constructs in section *Query Language*, below.

The remaining tag, `blast_hits` has type `SET(TUPLE(String, String, Float))`. The associated value is a set of triples, each consisting of a GenBank identifier, a GenBank description, and the probability that this similarity occurred by chance (so a low number means a better match). If the sequence of a `tested_short_fragment` is not found in GenBank, the value associated with `blast_hits` will be the empty set.

The *tag types*—types that can be associated with a tag in LabBase—are defined inductively as follows:

- Various *atomic types* are tag types: `STRING`, `INTEGER`, `FLOAT`, `DATE`, `DNA_SEQUENCE`, `BOOLEAN`. The sorts of values described by these types should be self-explanatory.

The atomic type `MATERIAL` describes values that are references to materials in a LabBase database. `MATERIAL` values are not directly denotable in the LabBase query language—there is no way to write a `MATERIAL` the way one can write an `INTEGER`. (To use the terminology of [19], `MATERIAL` is an “abstract” type.) However, LabBase requires that each material be assigned a user-accessible identifier that is unique within the material’s kind. Conceptually, `MATERIAL` is the supertype of all material kinds. The current implementation of LabBase does not support the use of material

kinds in tag types, but this support would be a useful enhancement.

- *Type constructors* take additional types as arguments. For example, in

```
LIST(TUPLE(FLOAT, SET(INTEGER)))
```

there are three type constructors: `LIST`, `TUPLE`, and `SET`. The argument of `LIST` is `TUPLE(FLOAT, SET(INTEGER))`, the arguments of `TUPLE` are `FLOAT` and `SET(INTEGER)`, and the argument of `SET` is `INTEGER`. There are no restrictions on how types can be composed in LabBase.

The type constructors `LIST` and `SET` describe lists and sets of values. The primary operations on `LIST` values are indexing (getting the *i*th value), retrieving elements in order, and testing for membership. There is also a type constructor, `SLIST`, that provides a space-saving representation for sparse lists and that provides the same operations as `LIST`. Although many semantic data models provide no list type constructor, it has been an extremely useful modeling construct at CGR. For example, reaction results produced in wells of a microtitre plate are naturally represented as a `LIST(INTEGER)`, in which the signal produced in each well is an integer.

The type constructor `TUPLE` describes fixed-length tuples of values of possibly heterogeneous types. For example, `TUPLE(FLOAT, SET(INTEGER))` describes pairs whose first element is a float, and whose second element is a set of integers.

LabBase does not currently provide a union type constructor or a facility for naming new types. As a result it cannot describe recursive types. We are considering adding these facilities to future implementations of the system, though so far their absence has not been problematic.

The LabBase *query language* (see section *Query Language*) recognizes, in addition to tag types, types constructed using the atomic types `STEP`—the type of LabBase steps—and

TERM—the type of terms in the LabBase query language.

**Steps with Multiple Materials** As another example, a step of kind `test_long_fragment_step` (recording results of Op-4) might consist of the following tag-and-value pairs:

<code>who</code>	<code>'sam'</code>
<code>when</code>	June 14, 1994 at 23:06
<code>tested_short_fragment</code>	the short DNA fragment with id PB223
<code>tested_long_fragment</code>	the long DNA fragment with id X0_A_246
<code>score</code>	2

By the rule presented above, this step appears on the histories of both the short and long fragment materials.

**Type Extension** LabBase is designed to make it relatively easy for a programmer to add new built-in atomic types or type constructors, which then have the same status as those discussed above. For example, the type constructor `SLIST` is a recent addition that took about two day's work. LabBase is designed with extension in mind because of the requirement that it represent unusual data types, and the assumption that we will not be able to foresee all the types and operations that will be needed by some laboratory in the future.

To add a new built-in type it is mainly necessary to create a new C++ class in the appropriate class hierarchy. A detailed discussion of particulars for the current implementation of LabBase appears in [21]. The operations on a new built-in type that will by default be accessible from the query language will depend on the new type's position in the C++ class hierarchy. Section *Query Language* describes how one can explicitly make additional operations available in the query language.

**Id Tags** Certain tags are *id tags*. When an id tag appears in the step history of a material, the associated value (which must be a `STRING` in the current implementation) becomes a unique identifier (within the set of values associated with that tag) for that material. When a material with material kind *kind* is created, the step that creates it (always of step kind `create`) must have the id tag *kind\_id*. For example the creation of a `long_fragment` material might involve simply the assignment of an id to the long fragment and a comment about its source:

```
who           'sue'
when          January 10, 1993 at 23:06
created_material  the long DNA fragment with id X0_A_246
long_fragment_id  'X0_A_246'
long_fragment_source  'Mix-N-Match Recombinant DNA, Inc.'
```

## Value Sets

In addition to steps and materials, LabBase supports top-level *value sets*, which at CGR are used either to store intermediate results in complex, multi-statement queries, or to group materials depending on their status in the laboratory production line. For example, a value set named `'waiting for blast'` might contain all the short fragments that require the `blast_step` operation (`Op-2`). Thus, value sets provide direct support for the workflow representation discussed in [16].

## Query Language

The LabBase query language is based on well-known constructs from logic programming languages such as Prolog [22]. Essentially, the LabBase query language is non-recursive datalog without rules. (Datalog has more restricted functionality than Prolog. See Ullman [23] for an overview of datalog and other logic-programming approaches to querying databases.)

Our choice of datalog as a query language was motivated by several factors:

- Non-recursive datalog has an expressiveness similar to that of SQL, but with the added capability of gracefully handling compound values such as LabBase’s lists and sets. (Because LabBase lacks term matching and unification, it uses built-in accessor predicates to decompose compound values. See, for example, the `ith` predicate discussed in section *Built-In Predicates*.)
- The language itself has a simple grammar, making a parser easy to implement.
- Evaluation techniques for datalog are well-studied, and a simple bottom-up evaluator seemed adequate for our immediate needs.
- Ease of use by end users was not a design requirement. In any case, it is not clear that SQL-like languages are easier to use than datalog.
- It is possible to extend the language to a full-featured logic programming language with recursive rules.

Further discussion of this choice appears in [24]. Term matching would make some queries more succinct, and rules, even non-recursive ones, would make many queries more readable. These capabilities are on our list of planned enhancements to LabBase.

A LabBase query consists of a list of comma-separated *terms* and ends with a period. Each term is the application of a *predicate* to zero or more arguments. For example,

```
short_fragment(M) .
```

is a query consisting of a single term with a single argument. Each argument can be a *variable* or a *literal*. Following common Prolog conventions, variables begin with an upper case letter, and literals are either numeric literals, string literals enclosed in single quotes, or alphanumeric identifiers beginning with a lower-case letter. The argument, `M`, of the predicate `short_fragment` in the query above is a variable. The predicate in this query is

a material kind; such a predicate takes a single argument and is true only for materials of that kind.

The set of *bindings* returned by a query is the set of all possible assignments of values to variables in the query that make all the predicates in the query true. For example, to find all possible `short_fragment` and `long_fragment` pairs, one could use the query

```
short_fragment(M1),long_fragment(M2).
```

This query produces a number of bindings that is equal to the number of short fragments in the database times the number of long fragments in the database. The output (depending on the contents of the database) would look like:

```
M1=short_fragment(PB234),M2=long_fragment(X_23_45)
M1=short_fragment(PB234),M2=long_fragment(Y_0_78)
:
M1=short_fragment(PB234),M2=long_fragment(A_8_08)
M1=short_fragment(UT89),M2=long_fragment(X_23_45)
M1=short_fragment(UT89),M2=long_fragment(Y_0_78)
:
M1=short_fragment(UT89),M2=long_fragment(A_8_08)
:
```

LabBase uses datalog predicates to provide both static and historical perspectives on laboratory data as described next.

## Static Perspective

Queries can obtain a static perspective on materials by using predicates that are tag names, which act as accessors of the most recent value associated with a tag. For example, to retrieve the most recent set of `blasts_hits` for the `short_fragment` with `short_fragment_id` PB223 one would pose the query

```
short_fragment_id(F,'PB223'),blast_hits(F,Hits).
```

This query would bind `F` to the `short_fragment` with `short_fragment_id` `PB223`, then find the most recent occurrence of the tag `blast_hits` in `F`'s step history, and bind `Hits` to the associated value. If tag `blast_hits` is absent from `F`'s history, then the `blast_hits` predicate is false. To generalize, any tag identifier can act as a binary predicate that binds its second argument to the most recent value associated with that tag in the step history of its first argument. We call a predicate that is the same as a tag identifier a *tag predicate*. The predicate `short_fragment_id` is also a tag predicate. Because `short_fragment_id` is an id tag, it can be used to find the `MATERIAL` associated with a particular id string (in this case `'PB223'`).

LabBase tag predicates are general enough to provide access to results produced by experiments on several materials. For example, to find out the most recent `scores` obtained by testing all long fragments against all short fragments we would write

```
short_fragment(SF),long_fragment(LF),score(SF,LF,S).
```

When a tag predicate contains more than two arguments, all but the last are understood to be materials, and the last argument is bound to the value associated with the most recent occurrence of the tag in any step that is in the history of each of the materials.

## Historical Perspective

Queries can obtain a historical perspective on materials by using the `all_steps` predicate: `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 the steps in the history of the short fragment `PB223` we could use the query

```
short_fragment_id(M,'PB223'),all_steps(M,S).
```

Like materials, steps are not denotable. Other predicates provide information about the steps. For example, if we are interested in only steps of kind `create` and `blast_step` we could pose the query

```
short_fragment_id(M,'PB233'),all_steps(M,S),
  or(create(S),blast_step(S)).
```

This query uses step kinds `create` and `blast_step` as unary predicates, which are true only if their argument is a step of that kind. The `or` predicate is true when either of its arguments is true.

LabBase can also use a tag predicate to retrieve the value associated with that tag in a particular step. For example

```
short_fragment_id(M,'PB233'),all_steps(M,S),
  tested_long_fragment(S,L),score(S,Score).
```

will bind `L` to all the `tested_long_fragment` values and `Score` to the `score` values found (on the same step) in PB233's step history. The query is true only for steps, `S`, that have *both* tags.

## Built-In Predicates

Because LabBase has no rules, built-in predicates are especially important. LabBase has a small, but ever-growing set of built-in predicates, such as those that give the cardinality of a `SET` or that match regular expressions against strings. There are also predicates that call other predicates. For example,

```
insist(short_fragment_id(S,'PB233')),not(score(S,X)).
```

will raise an error if there is no `short_fragment` with id PB233, and will be true only if PB233 exists and does *not* have any `score` tag in its history. (The predicate `not` behaves as in Prolog.) The remainder of this section discusses some of the other built-in predicates in LabBase and updates. Exhaustive specifics can be found in [21].

**Compound Values** LabBase uses predicates to break apart (and reassemble) compound values. For example, to find short fragments with (most recent) blast hits whose probability is below  $10^{-6}$  one could pose the query

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

In this query, `blast_hits` is true when `Hits` is bound to the value associated with the most recent occurrence of the tag `blast_hits` in `S`'s history. The term `element(Hits, Triple)` is true for every element, `Triple`, in `Hits`. The term `ith(Triple, 2, P)` is true if `P` is the last element (i.e. element 2 using 0-based indexing) in `Triple`. The final term, `P < 1.0e-06`, tests that the probability is low enough; in this term the predicate (`<`) appears as an infix operator, in customary Prolog fashion.

**Aggregates** LabBase departs from Prolog custom in its handling of *aggregate predicates*, which relate a set or multiset of bindings to some value. A simple aggregate predicate is `count`. For example, the query

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

finds those short fragments that in their most recent `blast_hit` have more than one element with probability (`P`) less than  $10^{-6}$ . The `count` predicate is true if its last argument is the number of times bindings were found that made its previous arguments true when interpreted as a query.

**Updates** LabBase adopts the Prolog approach to updates: it provides side-effecting predicates that actually perform the update. For example, to create (insert) a new material of kind `long_fragment` one could execute the query

```
insert(long_fragment(long_fragment_id='ZZZ_45',
    who=tom,when=1994:07:08:09:15:27)).
```

The tag-and-value pairs for the new material's `create` step are connected by the infix operator `=`. The date is specified in a syntax designed for easy machine handling. Adding a

new step is similar. For example, to add a `blast_step` to `short_fragment X2385` one could execute the query

```
short_fragment_id(S,'X2385'),
  insert(blast_step(
    tested_short_fragment=S,blast_hits={},
    who=steve,when=1994:08:01:15:56:23)).
```

(There are no similar sequences in GenBank, so the value associated with the `blast_hits` tag is the empty set.)

**Adding New Built-In Predicates** There is a simple interface for adding new built-in predicates to LabBase. (In many cases these would correspond to so-called “evaluable” predicates in Prolog.) The full set of LabBase built-in predicates is described in [21], which also describes how to add new ones.

## Application Program Interface

The LabBase API consists of a library of perl subroutines built around a subroutine called `send_query` that sends a query to LabBase and returns an array of bindings. Each element of the array can be transformed into an associative array that maps variable names to their values in the binding. For example, to find the average number of steps on all `short_fragments`, one could execute the following

```
@r = lb::send_query($database,
                    "short_fragment(M),count(all_steps(M,S),C)",
                    *errors);

die join("\n",@errors) if @errors;

$length = 0;

for $r (@r) { # For each row returned ...
  $count++;
```

```

    %x = split $;,$r; # Transform a row into an associative array.
    $length += $x{C}; # Increment $length by the value bound to C.
}

print "$count short_fragments, avg. history size is ",
      $length/$count, "\n";

```

In fact, the query above could have been computed directly in LabBase:

```

count(short_fragment(M),Count),
count(short_fragment(M),all_steps(M,S),Length),
Avg is /(Length,*(1.0,Count)).

```

Nevertheless, there are many cases where it is convenient or indeed necessary to read the data into a perl program and process it there. Besides `send_query`, there are a number of routines that help programmers construct common queries.

LabBase currently offers an API only for perl, because this is essentially the only language used as system-integration “glue” at CGR. A set-at-time API (such as the one described) is suitable for non-interactive LabBase clients, which predominate at CGR. Eventually we plan to provide a tuple-at-a-time (i.e. binding-at-a-time) interface for interactive clients that should show the query’s first results without waiting for it to finish.

## Conclusion and Future Work

LabBase is a clear improvement over its predecessor, MapBase [8, 9], which modeled material kinds (actually, a single material kind) and step kinds directly as C++ classes. In designing and constructing LabBase our objective was to use the lessons we learned from MapBase to provide a robust implementation of a system that would meet the essential DBMS requirements of a laboratory information system. In order to confirm the utility of these essentials in practice (and to meet our immediate data-management needs) the current implementation of LabBase provides little beyond them.

Another system that represents laboratory steps explicitly is the Object Protocol Model (OPM) [25], which has been used to model laboratory steps in a large-scale sequencing project. OPM is implemented by translation to an underlying relational DBMS, and offers no query language of its own at present. In contrast to LabBase’s minimalist representation of laboratory steps, OPM explicitly records a schema for the laboratory production line workflow, including a specification of the inputs and outputs of each kind of step. OPM also offers a full-fledged static semantic data model.

We see the approaches taken by OPM and LabBase as complementary: OPM focuses on *modeling* of protocols, functionality that we believed was not absolutely essential, and that we therefore did not provide in the first version of LabBase. Nonetheless, we believe that adding features of a full-fledged semantic data model and the ability to represent explicit workflow schemas would enhance LabBase’s usefulness. We have added to LabBase a system for abstractly specifying laboratory workflow [16]. With this system, each step in the database records the execution of a workflow activity, although this system does not store as much information about the types of workflow activities in the database as OPM would. On the other hand, this system provides more support for the execution of workflows than OPM currently provides.

A list of the enhancements that we would like to add to LabBase includes:

- True multiuser concurrency control; in the current implementation all clients communicate with a unique single-threaded database server, with the result is that short queries sometimes have to wait for another client’s long-running query to finish.
- Static schemas in a semantic data model including is-a relationships among material kinds and among step kinds.
- Support for workflow schemas similar to those in the Object Protocol Model, so that it will be easier to track the relationships among step instances.

- Rules, term-matching, and unification for the query language, so that complex queries can be constructed in manageable pieces.

LabBase and its API are in production at CGR and are freely available from the authors at <ftp://genome.wi.mit.edu/pub/software/labbase/>. The LabBase database for the human physical-mapping project contains over 1.2 million steps for over 2 hundred thousand materials. The size of the database is approximately .57 GigaBytes.

## References

- [1] Internet Engineering Task Force: The http protocol. Internet Draft available at <http://info.cern.ch/hypertext/WWW/Protocols/HTTP/HTTP2.html>.
- [2] Stein L, Marquis A, Dredge E, Reeve MP, Daly M, *et al*: Splicing UNIX into a genome mapping laboratory. In: *USENIX Summer 1994 Technical Conference*. pp 221–229, June 1994. Available at [ftp://genome.wi.mit.edu/pub/papers/Y1994/Summer94\\_Usenix.ps.Z](ftp://genome.wi.mit.edu/pub/papers/Y1994/Summer94_Usenix.ps.Z).
- [3] Kerlavage AR, Adams MD, Kelly JC, Dubnick M, Powell J, *et al*: Analysis and management of data from high-throughput expressed sequence tag projects. In: Mudge TN, Milutinovic V, Hunter L (Eds): *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*. Vol 1, IEEE Computer Society Press, pp 585–594, Jan. 1993.
- [4] Kerlavage AR, FitzHugh W, Glodek A, Kelley J, Scott J, *et al*: Data management and analysis for high-throughput DNA sequencing projects. *IEEE Engineering in Medicine and Biology*, 1995. In press.

- [5] Clark SP, Evans GA, Garner HR: Informatics and automation used in physical mapping of the genome. In: Smith DW (Ed): *Biocomputing Informatics and Genome Projects*. Academic Press, Inc., pp 13–49, 1994.
- [6] Mahaffey RR: *LIMS Applied Information Technology for the Laboratory*. Van Nostrand Reinhold, 1990.
- [7] Nakagawa AS: *LIMS: Implementation and Management*. Thomas Graham House, The Science Park, Cambridge CB4 4WF, England, Royal Society of Chemistry, 1994.
- [8] Goodman N, Rozen S, Stein L: Building a laboratory information system around a C++-based object-oriented DBMS. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Sept. 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/building.ps.Z`.
- [9] Goodman N: An object oriented DBMS war story: Developing a genome mapping database in C++. In: Kim W (Ed): *Modern Database Management: Object-Oriented and Multidatabase Technologies*. ACM Press, 1994.
- [10] Richardson JE, Eppig JT, Nadeau JH: A view from the front line: Building an integrated mouse genome database. *IEEE Engineering in Medicine and Biology*, 1995. In press.
- [11] Stroustrup B: *The C++ Programming Language*. Addison-Wesley, 2nd ed., 1991.
- [12] Wall L Schwartz RL: *Programming perl*. O'Reilly & Associates, Inc., 1990.
- [13] Lamb C, Landis G, Orenstein J, Weinreb D: The ObjectStore database system. *Communications of the ACM*, 34(10)50–63, 1991.
- [14] Singhal V, Kakkad SV, Wilson PR: Texas: an efficient, portable persistent store. In: *Proceedings of Fifth International Workshop on Persistent Object Systems (POS-V)*

- (San Minato, Italy). Sept. 1992. Available at `ftp://cs.utexas.edu/pub/garbage/texasstore.ps`.
- [15] Rozen S, Stein L, Goodman N: Constructing a domain-specific DBMS using a persistent object system. In: Atkinson M, Benzaken V, Maier D (Eds): *Persistent Object Systems*. Springer-Verlag and British Computer Society, Workshops in Computing Series 1995. Presented at POS-VI, Sept. 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-design.ps.Z`.
- [16] Stein L, Rozen S, Goodman N: Managing laboratory workflow with LabBase. In: *Proceedings of the 1994 Conference on Computers in Medicine (CompMed94)*. World Scientific Publishing Company, 1995. In press. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1995/workflow.ps.Z`.
- [17] Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: Basic local alignment search tool. *J. Mol. Biol. (England)*, 215(3):403–410, 1990.
- [18] Burks C, Cassidy M, Cinkosky MJ, Cumella KE, Gilna P, *et al*: GenBank. *Nucleic Acids Research* 19:2221–2225, 1991.
- [19] Hull R, King R: Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
- [20] Durbin R, Thierry-Mieg J: 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`.
- [21] Rozen S, Stein L, Goodman N: *LabBase User Manual*, 1995. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-manual.ps`.
- [22] Clocksin WF, Mellish CS: *Programming in Prolog*. Springer-Verlag, 1987.

- [23] Ullman JD: *Principles of Database and Knowledge-Base Systems*: vols. 1,2. Computer Science Press, 1988, 1989.
- [24] Goodman N, Rozen S, Stein L: Requirements for a deductive query language in the MapBase genome-mapping database. In: Ramakrishnan R (Ed): *Applications of Logic Databases*. pp 259–278, Kluwer, 1994. Available at <ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps>.
- [25] Chen IMA, Markowitz VM: The Object-Protocol Model, version 3.0. Tech. Rep. LBL-32738: Lawrence Berkeley Laboratory: 1 Cyclotron Road, Berkeley, CA, 94720, USA, Dec. 1994. This document and others on OPM available via [http://gizmo.lbl.gov/DM\\_TOOLS/OPM/opm.html](http://gizmo.lbl.gov/DM_TOOLS/OPM/opm.html).

## Acknowledgments

Andre Marquis contributed several built-in predicates to LabBase. Mary-Pat Reeve was the first non-developer user, and a principal designer of the human physical-mapping schema. Robert Nahf, another non-developer user has provided much patient and useful feedback. Tony Bonner provided a number useful suggestions that were incorporated into the LabBase query language. Barbara Levy provided editorial assistance. We thank Eric S. Lander for his support and encouragement.