

The LabBase system for data management in large scale biology research laboratories

Nathan Goodman¹, Steve Rozen², Lincoln D. Stein³ and Alex G. Smith¹

¹The Jackson Laboratory, 600 Main Street, Bar Harbor, ME 04609-1500, USA,

²Whitehead Institute for Biomedical Research, Cambridge, MA, USA and ³Cold Spring Harbor Laboratory, Cold Spring, NY, USA

Received on February 2, 1998; revised on April 2, 1998; accepted on April 7, 1998

Abstract

Motivation: The development of laboratory information management systems (LIMSs) for large scale biology research projects can be a challenging problem. Many such projects generate complex datasets via complex procedures that undergo continuous refinement. A key software challenge is to simplify the database-development task so that databases can be built and modified quickly enough to keep pace with changing project-requirements.

Results: LabBase extends the facilities offered by relational database systems to simplify the task of creating databases for large scale biology research projects. LabBase provides a structural object data model, similar to ACEDB, and adds to this the concepts of Materials, Steps, and States: Materials are objects representing the identifiable things that participate in a laboratory protocol; Steps are objects reporting the results of a laboratory or analytical procedure; and States are objects denoting places in a laboratory protocol. The system provides a data definition language for succinctly defining laboratory databases, and operations for conveniently storing and retrieving data in such databases. The system also provides support for workflow management. LabBase is implemented in Perl5 and provides a natural interface for laboratory application programs written in Perl.

Availability: The software is freely available. Contact the authors.

Contact: nat@jax.org

Introduction

The requirements for laboratory databases vary considerably from project to project (Kerlavage *et al.*, 1993, 1995; Clark *et al.*, 1994; Sargent *et al.*, 1996). The most basic role of a laboratory database is to store the data and analyses generated in the course of the project; such datasets vary over a wide range of difficulty, from simple lists of independent observations to highly interdependent networks of complex ob-

jects. Many projects also use databases to monitor the progress of work flowing through the project, a task often called *sample tracking* when simple and *workflow management* when complex; laboratory processes may be as simple as a single procedure or as complex as an arbitrary flowchart with branch-points and cycles, and the difficulty of the data management task varies concomitantly. Another consideration is the rate of change of the laboratory process and the impact of such change on the database software; some projects are stable and change little over their lifetimes, while others undergo continuous change as laboratory personnel refine old techniques, invent new ones, and incorporate methods developed elsewhere in the ongoing struggle to remain competitive. Operational factors relating to database size, the volume of update and retrieval activity, availability, security, and multi-user access shape the requirements further.

No single technology can satisfy this full range of requirements. In simple cases, a spreadsheet running on a personal computer (or even a laboratory notebook) may suffice. For projects that are more complex, the appropriate technology depends on which aspects are most pressing; projects with stringent operational requirements will tend toward robust commercial technology, such as relational databases (Date, 1995); projects with complex datasets may prefer ACEDB (Durbin and Mieg, 1991), OPM (Chen and Markowitz, 1995), or commercial object-oriented databases (Cattell, 1994); projects involving complex, multi-step processes may favor workflow management systems (WfMC, 1996); projects facing rapid change should focus on approaches that enable rapid software development. For projects that are highly complex along multiple dimensions, there may be no good choice, and the development effort will be challenging. At the extreme of complexity along all dimensions, technical feasibility may be in doubt.

LabBase addresses a constellation of requirements that we have seen in large scale projects at the Whitehead Institute Center for Genome Research and elsewhere. These projects are complex along three dimensions. First, they give rise to complex datasets. Second, the laboratory processes are com-

plex, and workflow management is an essential aspect of the problem. Third, and perhaps most striking, these projects experience a very rapid rate of change; these projects seek to be world leaders in the rapidly moving field of genomics, a position that can only be sustained through constant innovation. On the other hand, these projects have modest operational requirements: the databases are not too big; database activity is moderate; it is not a disaster for the system to crash occasionally; security is not a major concern.

The key challenge in this environment is to simplify the database-development task so that databases can be built and modified quickly enough to keep pace with changing project-requirements. As a general rule, complexity and change are opposing forces. To construct a complex system takes time and effort. As requirements change, the system becomes ill-suited to the demands of the project, and to bring the system in line with the new requirements yet more time and effort must be expended. If requirements change too quickly, the software will constantly lag and never serve the project effectively. In extreme cases, the system may be obsolete before it is finished.

LabBase attacks this challenge by implementing generic mechanisms that solve certain fundamental problems of laboratory data management and that can be quickly customized for the specific purpose at hand. The system provides a data definition language for succinctly defining laboratory databases, and operations for conveniently storing and retrieving data in such databases. LabBase provides support for workflow management and is designed to work with the LabFlow workflow management system (Goodman *et al.*, 1998).

The software described in this article is the latest in a series of systems we have built to tackle these problems (Goodman *et al.*, 1994; Stein *et al.*, 1994a, b; Rozen *et al.*, 1995). We and our colleagues have used the predecessor systems in genetic and physical mapping of mouse (Dietrich *et al.*, 1996; WICGR, 1997a), genetic mapping of rat (Steen, 1997), physical mapping of human (Hudson *et al.*, 1995), and genomic sequencing of human and mouse (WICGR, 1997b). The current system is just now entering service.

LabBase is implemented in Perl5 (Stein, 1996; Wall *et al.*, 1996) on top of commercial relational database systems. The system is designed to be conveniently used by Perl5 programs. The software is freely available and redistributable (see <http://goodman.jax.org> for details). The system was first developed for Sybase (McGoveran and Date, 1992), then ported to ORACLE (Date and White, 1991). (As we write this, the ORACLE version has atrophied, but it could be re-suscitated with little effort.) Previous incarnations of our software were implemented on ObjectStore (Lamb *et al.*, 1991), an object-oriented database system.

This article presents a detailed look at the design and implementation of LabBase. Section 2 defines the LabBase data model, and Section 3 describes the implementation.

LabBase is research software and is incomplete in many ways. As we go, we will endeavor to point out the major areas that need further work. A longer version of this paper with further detail, and a LabBase Reference Manual can be found on the authors' Web site (<http://goodman/jax.org/>).

LabBase Data Model

Overview

Logically, the data model has two layers. The bottom layer offers a structural object model similar to ACEDB (Durbin and Mieg, 1991), OPM (Chen and Markowitz, 1995), lore (McHugh *et al.*, 1997), UnQL (Buneman *et al.*, 1996), and many other systems; this layer supports complex data, but is not specific to laboratory databases per se. On top of this, LabBase adds the concepts of Materials, Steps, and States that are tailored for laboratory data management. *Materials* are objects that represent the identifiable things that participate in a laboratory protocol, such as clones and sequences. *Steps* are objects reporting the results of a laboratory or analytical procedure, such as sequencing a clone, or running BLAST (Altschul *et al.*, 1990) on a sequence. *States* are objects that represent places in a laboratory protocol, e.g. 'ready for sequencing' or 'ready for BLAST analysis'. We use the term *Object* (upper-case) to refer to an object which is not a Material, Step, or State.

LabBase provides built-in support for two relationships among Materials, Steps, and States that lie at the core of many laboratory databases. One is a relationship connecting Steps to the Materials upon which they operate. When a Step is stored in the database, LabBase automatically links the Step to its operand Materials in a chronological history and provides a means to access Step-specific data directly from these Materials; for example, one can retrieve a clone's sequence or a sequence's BLAST analysis by querying the Materials that were affected by the sequencing or BLAST analysis Steps. The second built-in relationship connects Materials to States. When a Material is created, LabBase provides a means to place the Material in an initial State; then as Steps operating on the Material are created, the system provides a means to move the Material to the appropriate next State thereby tracking its progress through the protocol. Both of these relationships are many-to-many.

It would be useful to extend the system to handle commonly occurring relationships involving Materials. There are many situations in which one Material is *derived from* another (e.g. when a sequence is derived from a clone), or when a set of Materials are grouped together to form a new Material (e.g. when a set of samples are arrayed in a plate or on a chip). Part/whole relationships are common also. It would be useful for the system to 'understand' such relationships as much as it currently understands history and state relationships.

LabBase is far from a full object-oriented database system (Cattell, 1994). It lacks, among things, support for inheritance or class hierarchies in any general sense, and database support for methods (i.e. functions) applied to objects. In practice, these limitations haven't yet presented major hindrances to the design of working laboratory management systems.

Data structures

A LabBase object is a collection of named fields. The values that may be stored in fields are drawn from a small universe of defined data types which includes

- booleans, numbers (integers and floats), and time-stamps,
- strings, nucleic acid sequences (`nt_sequences`) and protein sequences (`aa_sequences`) of arbitrary length,
- strings of limited length (`short_strings`), up to 255 characters in the current implementation,
- sub-objects (called structs),
- lists,
- pointers to other objects (object-references).

Objects, Materials, and Steps may not be used as values of fields, although references to them may be used. Figure 1 illustrates the schema of an example LabBase database.

Every LabBase object has a unique *internal object-identifier* used by the system to locate objects in the database and to connect objects together. A LabBase object is essentially a hierarchical record which is 'object-like' in that it has a unique identifier. A LabBase object may also have an *external object-identifier* which is visible to a program using the database. An external object-identifier is similar to a primary key in a relational database. LabBase notes the time each object was created and who did the creation and stores this information in special fields of the object.

LabBase objects are strongly typed: the system knows the type of every object, the fields that may appear in the object, and the data types of the values that may be stored in each field. Since Perl is non-typed, LabBase consults the database schema to check the type of an object when a Perl program creates or updates an object in the database.

A field may be *mandatory* or *optional* for a given type of object or struct. Mandatory means the field must be present with a non-NULL value in all objects or structs of the type; optional means the field may be omitted or have a NULL value in some objects or structs. A field whose value is NULL is equivalent in all respects to a field which is omitted from the object or struct.

Field names are global. This means if several types of objects or structs contain fields with the same field name, the fields must have the same type throughout. (Though, it may be mandatory in some types of objects or structs and optional in others.) This is unlike most relational databases, where

```

person      = Object (-id, name, address, dependents)
name        = struct (first, middle, last)
address     = struct (street, zip)
dependents = list (person)
first       = string
middle     = string
last       = string
street     = string

zip         = Object (-id, city, state)
city        = string
state       = string

library     = Material (-id, vector)
vector      = string

clone      = Material (-id, library)

pick_clone = Step (library, clone)

read_seq   = Step (clone, sequence)
sequence   = nt_sequence

blast_seq  = Step (clone, results)
results    = list (struct (p_value, score))
p_value    = float
score      = integer

```

Fig. 1. Example LabBase Schema. This figure shows the definitions of two types of Objects (`person` and `zip`), two types of Materials (`library` and `clone`), and three types of Steps (`pick_clone`, `read_seq`, and `blast_seq`). A `person` Object has fields `-id`, `name`, `address`, and `dependents`. As explained in the text, `-id` is a special field that hold an object's external object identifier. The other fields are defined in the subsequent lines of the figure: `name` is a struct (sub-object) with fields `first`, `middle`, and `last`; `address` is a struct with fields `street` and `zip`; `dependents` is a list of `person` Objects, i.e. a list of references to `person` Objects; `first`, `middle`, `last`, and `street` are strings; when used as a field, `zip` is a reference to a `zip` Object, whose definition follows.

column-names are local to each table. LabBase adopts this unusual property to support the ability to access Step data via Materials as mentioned in the preceding section.

Fields are single-valued, meaning that each field contains a single value of its allowed type (or perhaps no value if the field is optional). This is in contrast to the multi-valued fields of ACEDB. Lists may be used to emulate multi-valued fields.

Certain fields are pre-defined and used by the system for special purposes. In all cases, system-defined tags are prefixed by '-'. The built-in fields are listed below. Note that some of these are virtual fields computed by the software on demand.

- `-database_id`. Contains the object's internal object-identifier.
- `-id`. Contains the object's external object-identifier if the object-type has such.
- `-kind`. Contains the object's type.
- `-who`. Contains the object's creator.

- `-when`. Contains the object's creation time.
- `-history_list`. For Materials only, contains a list of all Steps that have operated on the Material.
- `-state_list`. For Materials only, contains a list of all States in which the Material currently resides.

A struct is a sub-object that exists as part of a surrounding 'top-level' object. Like the objects we have already discussed, a struct is a collection of named fields. A struct may only exist as the value of a field in an object or other struct, or as an element of a list (which itself may only exist as the value of a field or an element of a list). It is not possible to store an object-reference to a struct as the value of a field; this ensures that the only way to access a struct is via its enclosing top-level object, and that the structure of an object is acyclic. Like an object, a struct has `-database_id` and `-kind` fields, but it does not have `-id`, `-when`, or `-who` fields.

A list is an ordered collection of elements. Lists are *homogenous*, meaning that all elements of a list must be of the same type. The only exception are history-lists, which may contain multiple types of Steps. Lists may be defined over any data types that may appear as values of fields (i.e. everything except Objects, Materials, and Steps, although one may define lists of references to such elements). A list may contain NULL elements. A list may only exist as the value of a field in an object or struct or as an element of another list.

A State is a LabBase object which can be used to store status information associated with a Material, most typically to track the progress of a Material through a laboratory process. A State is, in most respects, an ordinary LabBase object. LabBase provides a predefined object-type, called `-state`, and a State is simply an object which is an instance of that type.

Materials and Steps are objects with some additional properties. First, they participate in the special Material/Step and Material/State relationships described in the previous section. Second, various LabBase operations have special behavior, described in the next section, when applied to these elements. Third, a Material is required to have an external object-identifier represented (as for all objects) by a `-id` tag.

Technically, Object, Material, and Step are *type constructors*, not types. One uses these constructs to define types of objects, such as 'clone'. Having done so, one can then create actual objects (instances) of the types, e.g. an object representing a specific clone. While we say colloquially, 'a clone is a Material', it is more precise to say, 'a clone-object is an instance of the clone-object-type, and that the clone-object-type is a type of Material'. By contrast, a State is an actual object.

The database schema contains the information known to LabBase about basic and constructed data types. For each type of object or struct, the schema tells the name of the type, the names of the fields that may appear in objects or structs of this type, whether the field is optional or mandatory, and

whether the type has an external object-identifier (in which case, it is mandatory and unique, meaning that all objects of the type must have unique, non-NULL identifiers). For each field, the schema also tells the data type of the values that may be stored in that field. LabBase stores the schema in the database as a collection of LabBase objects in much the same way as a relational database stores its schema (or *catalog*) in system-tables. A *data definition language* is provided for users to define schemas. The data definition language is illustrated in Figure 1.

It is natural to draw a LabBase object as a data structure diagram (directed graph), as in Figure 2, with one node representing the 'top-level' of the object and additional nodes for each struct and list (recursively) contained in the object, and with edges indicating which elements directly contain which others. We have already noted, but wish to restate for emphasis, that LabBase objects are strictly hierarchical. This means that the diagram of any individual LabBase object is always a tree. We can depict a network of objects by using other edges (drawn as dashed lines in the figure) to represent object-references. Networks of objects may, of course, be cyclic, as illustrated in the figure.

Operations

LabBase supports a 'fetch-and-store' database interface (Orfali *et al.*, 1996), also known as a 'two-level store', similar to most relational databases. Objects must be explicitly stored in the database to create them and explicitly fetched from the database to retrieve an up-to-date copy. Updates to an object must also be explicitly stored back in the database.

LabBase provides the following operations to manipulate objects:

- `put` creates objects (i.e. 'puts' them in the database).
- `get` retrieves entire objects or selected fields from objects satisfying a condition; `get` can retrieve entire structs and lists, but cannot retrieve selected fields from a struct nor selected elements from a list.
- `count` returns the number of objects satisfying a condition.
- `delete` removes from the database a set of objects satisfying a condition.
- `update` changes the values of selected fields in a set of objects identified by their object-identifiers; this operation can also be applied to structs.
- `set_states` changes the States of one or more Materials.

The 'conditions' that can be specified in these operations are far simpler than in many databases: an operation can manipulate all objects of a given type, or all Materials of a given type in a given State, or a single object denoted by its internal or external object-identifier; `set_states` can also operate on

```

person      = Object (-id, name, address, dependents)
name        = struct (first, middle, last)
address     = struct (street, zip)
dependents  = list (person)
zip         = Object (-id, city, state)
    
```

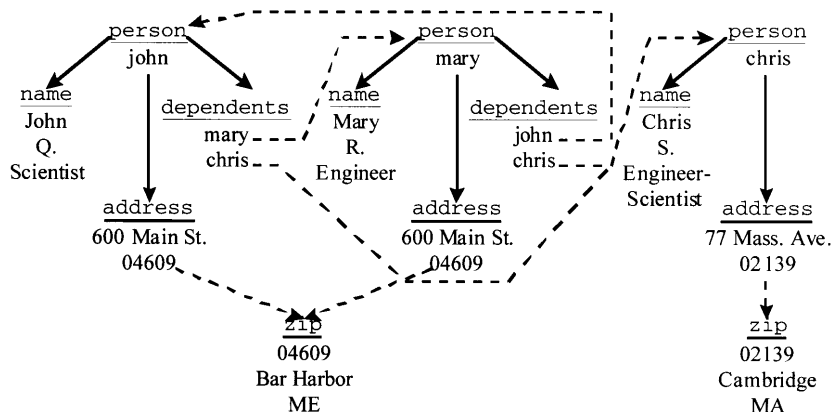


Fig. 2. Data Structure Diagram. This diagram shows three person Objects and two zip Objects.

all Materials in a given State. Beyond these conditions, there is no general query mechanism that allows one to manipulate the objects based on arbitrary fields. Previous versions of LabBase supported a general query mechanism based on datalog (Ullman, 1988; Ramakrishnan and Ullman, 1995); we have not carried this forward to the current version, but rather expect programmers to code complex data retrievals directly in Perl5. If this proves unworkable, we may implement a more traditional query facility in a future release.

When retrieving Materials, the `get` operation treats all user-defined fields of all Steps on the Material's history as virtual fields of the Material itself and allows these fields to be retrieved exactly as if they were real fields of the Material. Figure 3 illustrates this phenomenon. Note that fields of the same name may appear in many Steps, and may even appear in the Material itself. By default, `get` retrieves the *most-recent* instance of the field. Since field names are global, we are assured that all instances of the field have the same type, thus preserving the strong typing of the retrieved object. In general, different fields for the same Material will get their most-recent values from different objects. In addition, when operating on Materials, `get` can retrieve two other virtual fields maintained by the system: the list of all Steps that have operated on the Material (called its *history-list*), and the list of all States in which the Material currently resides (called its *state-list*).

When retrieving Materials, `get` can move each Material to a new State. This is a convenient way to indicate that the Materials are actively being processed by the client program and to avoid anomalies that can result from two program processing the same Material at the same time; this technique is sometimes called 'application-level locking'.

When creating Materials, `put` can place each Material into one or more initial States.

When creating Steps, `put` can be provided with a list of *associated* Materials. The system links the Step to each associated Material in a chronological history, and can be instructed to move each Material from one State to another.

A key issue with `delete` is referential integrity, viz. what to do with references pointing to the deleted object. Our approach is simple: instead of tracking down all references to the deleted object, LabBase never re-uses an object-identifier and guarantees that any attempt to follow a reference to a deleted object will yield NULL. The main consequence is that programmers must be aware that an object-reference retrieved from the database may be 'stale', and that an attempt to `get` the referent object may yield an empty result.

Every LabBase operation executes as an atomic transaction. For applications requiring more control over transaction boundaries, the system provides the ability to execute a block of code, generally involving many LabBase operations, as a single transaction.

LabBase implementation

Data structures

The implementation includes a small number of built-in tables which are independent of the particular LabBase schema, and a generally larger number of tables whose existence and definition are driven by the LabBase schema. Figure 4 illustrates the representation of an example database. There are also a handful of low level 'utility tables', e.g. to generate unique database-ids in Sybase, which we shall not discuss further.

```

clone      = Material (-id, library)
pick_clone = Step (library, clone)
read_seq   = Step (clone, sequence)
sequence   = nt_sequence
import_seq = Step (clone, sequence)

```

<u>clone</u>	<u>clone</u>	<u>clone</u>	<u>clone</u>
P123	P456	P789	P246
<u>pick_clone</u> 00:01 23 Dec 97	<u>pick_clone</u> 00:03 23 Dec 97	<u>pick_clone</u> 00:05 23 Dec 97	<u>pick_clone</u> 00:07 23 Dec 97
<u>read_seq</u> ACGT... 00:01 25 Dec 97	<u>import_seq</u> CGTA... 00:02 25 Dec 97	<u>read_seq</u> GTAC... 00:03 25 Dec 97	<u>read_seq</u> TACG... 00:04 25 Dec 97
		<u>import_seq</u> CATG... 00:05 26 Dec 97	<u>read_seq</u> ATGC... 00:06 26 Dec 97

Fig. 3. Database Illustrating Access of Step-data via Materials. This figure extends the schema of Figure 1 with an additional Step, import_seq. The intent is read_seq and import_seq are two different ways of obtaining the sequence of a clone. The figure illustrates four clone Materials and their history-lists. In all cases, a get operation that retrieves the sequence field of a clone will retrieve the sequence data from the last (i.e. most recent) read_seq or import_seq Step in the clone's history-list. The program doing the get does not have to know which type of Step this is.

Given a LabBase schema, we create a relational table for each type of object and struct defined in the schema; we call these *type-tables*. The name of the type-table is the same as the name of the type. The type-table contains a `-database_id` column which contains its internal identifier and is the primary key of the table. An index is always created on this column. If the object has a `-id` field (external database identifier), the type-table includes a column of the same name, and an index is created on this column. For each other field in the object or struct, the type-table will contain one or more columns depending on the data type of the field (see discussion below). For types that can be represented as a single column, the column name is the same as the field name. For types requiring multiple columns, the column names are generated by adding suitable one-letter suffixes to the field name.

In addition to the type-tables, there is a single table, called `-object`, which stores the `-who`, `-when`, and `-kind` fields for all objects (not structs). This table also has a `-database_id` column which is its primary key and which links the rows of `-object` with their correspondents in the type-tables. The `-kind` information is redundant in most cases, because of LabBase's strong typing; it is only used when retrieving the Steps contained on a Material's history-list. This table is not truly necessary. It would work just as well to store `-who` and `-when` in the type-table for each object, and to push the type information into the `-history` table (see below). The current design is more normalized hence better by conventional criteria. We have some concern that `-object` will prove to be a concurrency hot spot; if

experience bears this out, the suggested denormalizations are the clear solution.

We considered the option of generating synthetic names for type-tables and columns rather than using the names as given in the LabBase schema. The advantage of our current approach is that the relational schema is reasonably intelligible. The disadvantage is that names used in LabBase schemas must conform to constraints imposed by the relational database; in particular, names are limited to about 32 characters and may not contain white space or punctuation. On balance, it would probably be better to switch to synthetic names.

Lists are represented by rows in a table called `-list`. Each row in `-list` represents one list element. To tie the elements of a given list together, the table includes a column which contains an identifier for the list, and another column which indicates the ordinal position of each element in the list. Since lists can be defined over any data type, the table contains columns for each different data type. We discuss the representation of the types below. In any given row, most of these columns are NULL.

We felt it important to optimize the case of lists containing a single element. We expect that some applications will use lists to emulate multi-valued fields in the style of ACEDB. In this setting, most fields are defined as 'multi-valued' (indeed, this is the default in ACEDB), although in most data instances, most fields contain a single value. In our terminology, this means that most fields will contain lists of one element. To handle this case efficiently, we store the first list element in the enclosing table, and resort to the `-list` table

person					
-database_id	-id	name	address	dependents	dependentsL
123	john	1	1	124	1
124	mary	2	2	123	2
125	chris	3	3	NULL	NULL

name						
-database_id	first0	first	middle0	middle	last0	last
1	John	NULL	Q.	NULL	Scientist	NULL
2	Mary	NULL	R.	NULL	Engineer	NULL
3	Chris	NULL	S.	NULL	Engineer- Scientist	NULL

address			
-database_id	street0	street	zip
1	600 Main St.	NULL	126
2	600 Main St.	NULL	126
3	77 Mass. Ave.	NULL	127

zip					
-database_id	-id	city0	city	state0	state
126	04609	Bar Harbor	NULL	ME	NULL
127	02139	Cambridge	NULL	MA	NULL

-object			
-database_id	-kind	-who	-when
123	person	nat	00:01 01 Jan 98
124	person	nat	00:02 01 Jan 98
125	person	nat	00:03 01 Jan 98
126	zip	nat	00:03 01 Jan 98
127	zip	nat	00:04 01 Jan 98

-list			
list	position	val_object	...
1	1	125	NULL
2	1	125	NULL

Fig. 4. Implementation of Objects from Figure 2. This figure illustrates the database representation of three person and two zip objects from Figure 2. The -list table contains additional columns, depicted by ..., for other data types not used in the example. The -string table is omitted since none of the strings in the example are long enough to overflow there. For string-valued fields, the column with suffix '0' hold the first portion of the string, and the column without the suffix holds a link to the rest; these are all NULL in the example. For list-valued fields, the column without the suffix holds the first element of the list, and the column with suffix 'L' holds a link to the rest.

only for subsequent elements. There are some exceptions to this general rule discussed later in the section.

Strings pose a complication in the design. In standard relational databases, the basic string data types, char and varchar, are limited to a maximum of 255 characters. Most relational database products support data types for longer strings, e.g. TEXT or BLOB, but different products impose different restrictions on these types, and in some products, there is considerable overhead in using these mechanisms.

For these reasons, we initially chose to implement strings in a list-like manner. In this initial implementation, we represented a string by rows in a -string table. Each row of this table contained a substring of up to 255 characters. To tie the

substrings of a given string together, the table included a column which contained an identifier for the string, and another column which indicated the ordinal position of each substring in the whole. As with lists, we optimized the (very common) case of strings shorter than 255 characters by storing the first substring in the enclosing table.

Experience quickly showed that this representation was dreadfully slow for strings greater than about 1000 characters, a case which arises frequently in projects dealing with genomic sequences or full-length cDNAs. We soon changed the implementation so that the -string table contains seven varchar columns, each capable of holding 255 characters, plus one TEXT field capable of storing an arbitrary

number of characters. The choice of seven varchar columns was driven by limitations on the maximum row size imposed by the underlying database system. As in the first design, we optimized short strings by storing the first 255 characters in the enclosing table. This design provides three tiers of efficiency: short strings (up to 255 characters) are most efficient, because the entire string is stored in the enclosing table; strings containing 256–2040 characters are somewhat less efficient, because some characters are stored in varchar fields of the `-string` table; long strings (more than 2040 characters) are least efficient, because they use the heavyweight TEXT mechanism.

One final annoyance in the string implementation is that varchar and TEXT fields truncate trailing white space! Since we spread a single LabBase string across multiple varchar fields, white space in the middle of a LabBase string can coincidentally land on the end of a varchar. To consider an extreme case: given a string containing an ‘a’ followed by 2039 spaces, then a ‘b’, the varchar implementation would collapse the entire string to ‘ab’! Our solution is to use binary instead of character data types: viz. varbinary instead of varchar, and IMAGE instead of TEXT. Even this is imperfect, because binary data types truncate trailing binary 0’s. Still, it works well enough for our purposes.

The row representing an object or struct must be linked to the rows representing the structs, lists, and strings contained therein. We have chosen as a general design principle to orient links ‘downward’, so that each type-table contains identifiers pointing to its constituents. The alternative design, in which links point upward is possible as well. The advantage of the downward choice is that we can tell when a link is NULL without having to access the table representing the constituent; this is an important benefit since we expect many such links to be NULL, especially for strings and lists.

The two built-in relationships that LabBase supports over Materials, States, and Steps are each implemented by a linking table in the usual way. We have a table called `-state_rel` with two columns, one containing internal identifiers of Materials and the other containing internal identifiers of States. A row in this table indicates that a given Material is in a given State. We have a similar table called `-history` with one column containing internal identifiers of Materials and a second containing internal identifiers of Steps. A row in this table indicates that a given Material is associated with a given Step. The `-history` table does not directly represent the chronological order of the history-list; to sort the history, one must join `-history` with `-object` to get the `-when` field of each Step.

To support access to Step-data via Materials, we store Step-data redundantly in a table called `-transpose`. Each row of `-transpose` contains a field-name, the value of the field and the internal identifier of the Step from whence this information comes. The name ‘`-transpose`’ reflects

the fact that this table is a transposed representation of the data contained in a Step; in a ‘normal’ representation, the columns of the type-table represent the fields of the Step, while in this ‘transposed’ representation, the rows represent the fields. Since fields can take values from any data type, this table (like `-list`) contains columns for each different data type. In any given row, most of these columns are NULL.

As we have seen, there are several situations in which a row must contain a representation of one or more LabBase data types. In particular, each row of `-list` must contain the representation of a list element, each row of `-transpose` must contain the representation of a fields’s value, and each row of each type-table must contain the representations for all of its fields’ values. We use the same scheme to represent data types in all these situations.

Most data types can be naturally represented as a single column. These include boolean, integer, float, timestamp, `short_string`, object-reference, and struct. The first several are obvious. An object-reference is represented by the internal identifier of the referent object; there is no need to store the type of the referent object, because LabBase is strongly typed and the software can infer the type from the schema. For structs, the body of the struct is stored in its own type-table; the enclosing table merely stores an internal identifier pointing to the struct; again, there is no need to store the type of the struct since this can be inferred from the schema.

Strings require two columns: one is a varbinary (255) that contains the first (up to) 255 characters of the string; the other contains an identifier pointing to the rest of the string in the `-string` table. In many cases, the second column is NULL. The `nt_sequence` and `aa_sequence` data types use the string implementation.

For lists, the enclosing table contains one or two columns to represent the first element of the list, plus an additional column that contains an identifier pointing to the rest of the list in the `-list` table. In particular, a list of strings requires three columns: one for the first 255 characters of the first element of the list, a second for the remaining characters of the first element of the list, and a third for the rest of the list. Lists of lists are an exception to the general rule, because in a deeply nested list, the first element is itself a list of lists. (For example in a list of lists of integers, the first element is a list of integers.) We felt it unwarranted to invest effort in handling this special case efficiently, and we simply store all elements in the `-list` table. Lists of structs containing two or more elements are also an exception: in this case we store all elements in `-list` so that the list and its elements can be retrieved in single SQL statement that joins `-list` with the struct’s type-table.

A limitation of our software is that we offer no means for the database designer to tune the database representation. For structs, it would be useful to let the designer choose an alter-

person						
-database_id	-id	first0	first	middle0	middle	last0
123	john	John	NULL	Q.	NULL	Scientist
124	mary	Mary	NULL	R.	NULL	Engineer
125	chris	Chris	NULL	S.	NULL	Engineer-Scientist

person (cont.)					
last	street0	street	zip	dependents	dependentsL
NULL	600 Main St.	NULL	126	124	1
NULL	600 Main St.	NULL	126	123	2
NULL	77 Mass. Ave.	NULL	127	NULL	NULL

Fig. 5. Alternative Implementation of `person` Objects from Figure 2. This figure illustrates an alternative implementation of objects in which structs are stored in the same table as the 'top-level' object.

native implementation in which the struct is 'expanded' into the type-table of the enclosing object or struct as illustrated in Figure 5; this would reduce the number of tables to be accessed when retrieving an object and would be especially useful if applied recursively to deeply nested structures. We did not adopt this as the default implementation, because it can lead to excessively wide tables (which may exceed the limits of the relational database system), and because it only works for hierarchical schemas.

It is worth noting that the transposed representation used for Steps is capable of representing all types of objects and offers some advantages over the conventional representation. It is arguably the most natural way to represent Perl5 objects (which are essentially key/value pairs). It allows new fields to be added to an object-instance or object-type at any time and easily supports non-typed fields. It also provides the ability to efficiently find all objects that contain a specific field, a feature one also finds in ACEDB. We chose not to adopt the transposed design for all objects, because we were concerned about the extra space needed to store objects, and the extra time needed to reconstruct them. In our current design, we only pay this price for the top level of Steps. If we were to fully adopt this design, we would have to pay the price for the entire database. Sadly, we never did any experiments to assess the tradeoff between the functional advantages of the transposed design vs. the performance penalty. This is not as easy as it might sound, because to do these experiments, we would have had to fully implement both designs. In retrospect, it might have been better to adopt the transposed design for all objects at the outset, with the option of adding the standard design later if warranted. This would have been less work (because we had to implement the transposed design for Steps anyway), and it would have yielded a system with more capabilities. But, given the pressure to produce software that could be used in real laboratory projects, we were unwilling to pursue such a radical course.

Operations

To put an object into the database, the software must create rows in several tables: `-object`, the type-table for the object and each struct it (recursively) contains, `-string` for any strings longer than 255 characters, `-list` for any list with more than one element and lists of lists, and `-transpose` and `-history` for Steps that are associated with Materials. In addition, for Materials and Steps, the operation may have to create, update, or delete rows in `-state_rel` if the operation involves any state changes. Figure 6 illustrates the procedure.

The first step is to obtain a unique internal identifier for the object and insert a row into `-object`. The internal identifier is simply a unique sequence number; in Sybase we obtain this number by using `identity` columns; in ORACLE, we use a trigger to peel successive integers from a sequence type.

The next, and more substantial task, is to construct the database representation of the value of each field in the object. For many data types, viz. boolean, integer, float, timestamp, and `short_string`, this is a straightforward format conversion task in which the Perl5 input format is translated into the corresponding database format; for example, for boolean data, Perl5 'true' values are translated to 1 and Perl5 'false' values to 0. For the remaining data types, viz. object-references, strings, lists, and structs, we generally need to access or update the database to construct the representation.

For object-references, the input format may contain either an external or internal object identifier. If it contains an internal identifier, we assume the identifier is valid and simply use it as the identifier of the referent object. This is obviously risky, but is justified on the grounds that programs are only supposed to obtain internal object identifiers from the database; so long as programs follow this rule, the identifier is guaranteed to be valid. If the reference does not contain an

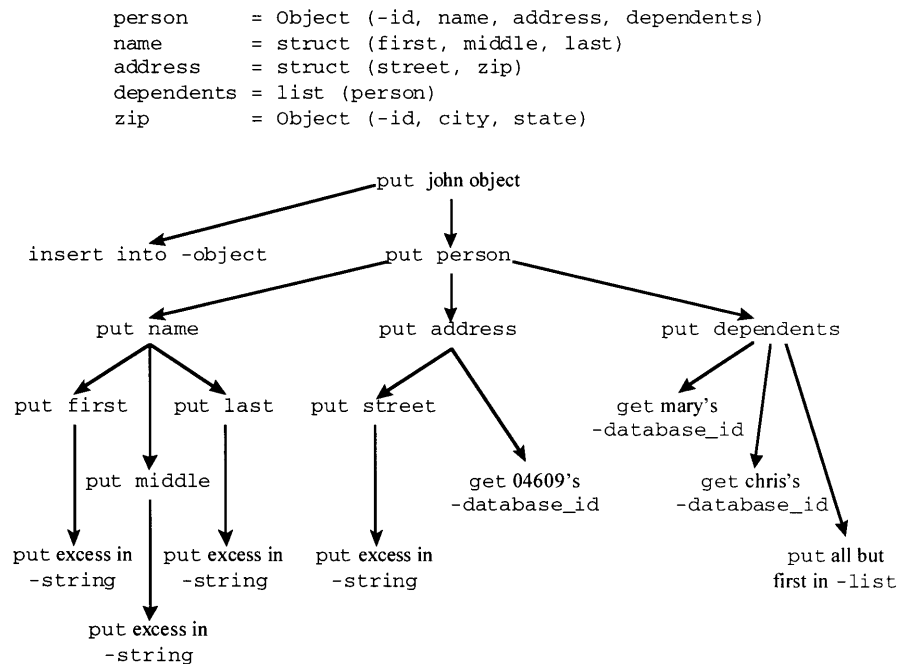


Fig. 6. Procedure for putting a person Object. This figure shows the series of database operations required to put the john person object of Figure 2 into the database. The figure assumes that all objects referenced by john are already in the database.

internal identifier, we access the database to lookup the internal identifier of the referent object given its external identifier.

For strings longer than 255 characters, we put the excess into the `-string` table. The function that puts data into `-string` returns the string identifier. The database representation of the string consists of two parts: a substring containing the first 255 characters, and the identifier pointing to the rest.

Lists are more complicated. First, we have to construct the database representation of each list element, and then construct the representation of the list itself. The former entails a recursive application of the procedure we are describing here for each list element. For example, to construct a list of strings, we have to take each string in turn and follow the procedure above; namely, if the string is longer than 255 characters, store the excess in `-string`, and represent the string as a whole by its initial substring together with the identifier pointing to the rest. Once the elements are constructed, if the list does not fit in the enclosing table (i.e. it has more than one element or is a lists of lists), we put the extra elements into the database as rows of the `-list` table. The function that puts data into `-list` returns the list identifier. The database representation of the list consists of two parts: the representation of its first element, and the identifier pointing to the rest.

For structs, we recursively apply the procedure described for objects, except that no row is inserted into `-object`. The function that performs this procedure returns the internal

identifier of the struct. For purposes of the row we are now constructing, the representation of the struct is simply its internal identifier.

After creating the representations for the values of each field, we combine these into a row, and insert the row into the type-table for the object. The net effect is to store the object bottom-up. The last row stored in the database is the one representing the top-level of the object.

When putting a Material in the database, if the put operation specifies one or more initial States for the Material, we must insert rows into `-state_rel` to place the Material in these States.

When putting a Step in the database, there are several further tasks. For each field in the Step, a row must be inserted into `-transpose`. For each Material associated with the Step, a row must be inserted into `-history` to connect the Material to the Step. Also, for each such Material, if the put operation specifies a state change, it is necessary to insert, delete, or modify rows of `-state_rel` to effect the change; we accomplish this by invoking the `set_states` operation.

Let us turn now to the `get` operation. We will only describe the case of getting all tags of an object given its internal object identifier; other cases are similar.

First, we access `-object` and the object's type-table to get the one row from each table corresponding to the object. This is accomplished by a single SQL statement that joins the two tables on `-database_id` and selects for the desired

material	-id	step	-kind	tag	-when	...
238	P789	239	pick_clone	library	00:05 23 Dec 97	...
238	P789	239	pick_clone	clone	00:05 23 Dec 97	...
238	P789	244	read-seq	clone	00:03 25 Dec 97	...
238	P789	244	read-seq	sequence	00:03 25 Dec 97	...
238	P789	246	import_seq	clone	00:05 26 Dec 97	...
238	P789	246	import_seq	sequence	00:05 26 Dec 97	...

Fig. 7. Retrieving Step-data for one Material from Figure 3. This figure shows the most relevant fields produced by joining the `clone`, `-history`, `-transpose`, and `-object` tables as needed to retrieve the most-recent value of each field for `clone` P789. From this data, we see that the most recent value of `library` comes from step 239, while the most recent values of `clone` and `sequence` come from step 246.

value of `-database_id`. The result is a row containing the database representation for each field of the object.

Next, we convert the database representation of each field into its Perl5 representation and combine these into a Perl HASH. For many data types, viz. boolean, integer, float, and short_string, the conversion is trivial because the database format is essentially identical to the Perl5 representation; for example, for boolean data, the database represents 'true' by 1 and 'false' by 0, which are valid representations of these values in Perl5. Timestamps and object-references are slightly harder. For timestamps, the value in the database must be fed into a Timestamp constructor (which is just a simple Perl5 function) to obtain a Perl5 Timestamp object. For object-references, the value in the database is the internal identifier of the referent object; the software must consult the schema to discover the type of the referent object, then construct a Perl HASH containing the type and the identifier.

For the remaining data types, viz. strings, lists, and structs, we generally need to retrieve more data from the database to construct the answer.

For strings longer than 255 characters, we must access `-string` to obtain the tail of the string, and concatenate the result with the head of the string stored in the top level row.

For lists that do not fit in the top level row, we have to access `-list` to get the remaining elements, combine these with the first element stored in the top level row, construct the Perl5 representation of each list element, and combine the results into a Perl5 ARRAY. The construction of each list element entails a recursive application of the procedure we are describing here and may involve more database accesses.

For structs, we recursively apply the procedure described for objects, except that no row is retrieved from `-object`.

The net effect is to retrieve the object top-down. The first row retrieved from the database is the one representing the top-level of the object. As the process recurses, rows are retrieved representing the structs and other elements that are more deeply embedded in the object. An alternative design would be to use outer-joins to retrieve the top-level row and all structs that it (recursively) contains in a single SQL statement. (Outer-join, rather than normal join, is needed to handle structs that are missing.) This could be made to work for

schemas that are acyclic and would probably improve performance. The same idea should not be used to retrieve all lists contained in an object, because the multiplicity of such elements can lead to a combinatorial explosion; for example, if an object contains two lists with 10 elements each, an outer-join that retrieves both lists will produce 100 (10×10) rows.

When getting a Material from the database, we must also get the most recent value of each field from its history-list; 'most-recent' is defined with respect to the `-when` field of the Step that created each field. The basic plan for doing this is to join `-history` (which links Materials and Steps) with `-transpose` (which links Steps to fields) with `-object` (which contains the `-when` field of each Step), and restrict by the internal identifier of the Material of interest. Figure 7 demonstrates the procedure. In principle, we could use SQL to select the most recent value of each field from this result, but the SQL statement for doing so is impressively opaque. We found it to be more effective to retrieve the entire result (i.e. all values for each field on the Material's history-list), and select the most-recent value in the LabBase software.

One consequence of this implementation is that it is equally efficient to retrieve the entire list of values for each field, rather than just its most recent value. We experimented with this capability and found that it was not terribly useful, because one usually wants to correlate values across fields; e.g. there is little point in getting lists of left-primers and right-primers for an STS without knowing which pairs of primers work together! More useful would be the ability to specify a set of fields and retrieve a list whose elements contain values extracted from the history-list for all specified fields.

We thought about extending the history-list idea to all objects (essentially doing away with the distinction between Materials and other types of objects). We chose not to allow this, because it complicates the process of getting field-values from an object's history-list. Suppose Steps were allowed to have histories, too; then when looking for the most-recent value of a field, we would have to look at fields on the Step's history-list, too *ad infinitum*. This seemed too complex.

When getting a Material, the `get` operation can also retrieve its history-list and state-list. The former is obtained by joining `-history` with `-object` and restricting by the internal identifier of the Material. The latter is obtained by joining `-state_rel` with `-state` and restricting by the internal identifier of the Material.

The `get` operation can also move the Material to a new State. This is accomplished by invoking the `set_states` operation.

The `count` operation is much simpler than `get`, because it cares about the existence of objects and not their content. We can, for example, count the number of Materials of a given type in a given state by joining the type-table for the desired type with `-state`, restricting by the internal identifier of the desired State, and counting the number of rows in the result. This can be accomplished by a single SQL statement. Since the conditions that can be specified in `count` are so limited, there is never a need to descend into the rows that represent the constituents of the object.

The `delete` operation retrieves the rows that represent the object in a top-down fashion, like `get`, then deletes these rows bottom-up, analogous to `put`. In addition, for Materials, all rows referring to the Material are deleted from `-state_rel` and `-history`. For Steps, all rows referring to the Step are deleted from `-history` and `-transpose`. The system guarantees that internal identifiers for deleted objects are never re-used.

The `update` operation is simpler, because it can only update fields directly contained in an object or struct. The operation gets the row representing the object or struct. Then, for each field being updated, if the old value is a string, list, or struct, the old value is deleted. Finally, for each field being updated, the new value is constructed as in `put`, and the ensemble submitted to the database as a SQL update statement.

The `set_states` operation can be used directly by applications, but is also used internally by `put` and `get` to effect the State changes they are responsible for. The operation can be used to move a Material from one State to another, or to remove the Material from a State without placing it in a new one, or to put the Material in a new State without removing it from the old. Additionally, `set_states` can be instructed to heed or ignore various anomalous conditions; for example, when moving a Material from one State to another, an anomalous condition arises if either State does not exist, or if the Material is not in the source State, or the Material is already in the target State. In some cases, `set_states` can accomplish the requested work through a single SQL update statement applied to `-state_rel`; often, however, the program must issue a series of deletes and inserts.

We execute each LabBase operation as a database transaction so that (i) no concurrent LabBase application can see the

database content midway through the process, and (ii) to ensure that if the system were to crash before the operation completes, the partial results would be rolled back.

Discussion

The facilities provided by LabBase make it easier to create a laboratory database that conform to our model by automating commonly occurring database design tasks. Since Steps are automatically connected to Materials, the database designer need not be concerned with how these links are maintained. Since Steps are automatically preserved in a chronological history, the designer need not be concerned with preventing updates from overwriting previous data. Since Step-data can be queried via Materials, the designer need not be concerned with storing laboratory results inside Materials, nor with providing views for this purpose. The structural objects of LabBase let the designer model complex data containing arbitrarily long strings, lists, and sub-objects without having to worry about the implementation details.

The software has numerous limitations. Most notably, the system is not easily extensible, it lacks many useful object-oriented features, and the query language is weak. We were reluctant to correct these flaws until the system went into production, and we could see which features were most important in real laboratories. Now that the system is entering service, we have started to extend the software in some of these areas.

A methodological concern is that the bulk of the implementation is devoted to fairly generic data management functions, such as support for structural objects, in contrast to features that are specific to laboratory data management, viz. the Material, Step, and State constructs. An alternative implementation strategy would have been to develop LabBase on top of an object-oriented database system. We pursued this approach in the past and believe it to be technically superior, but we were pushed down the relational path by the pressure of potential users. We found it difficult to convince potential users, other than our close collaborators, to try a non-relational solution, because projects whose database problems were hard enough to need LabBase were unwilling to entrust their data to a non-standard solution.

The LabBase approach is attractive from a functional standpoint. Given the current technology choices, LabBase offers a useful middle ground in which application developers gain some of the strengths of object-oriented data modeling while retaining the robust, efficient, and well-supported storage management facilities of today's relational products. It is reasonable to speculate that the newly emerging generation of object-relational database products will eventually gain widespread acceptance. If and when this

happens, it will time to consider a reimplementa- tion of Lab-Base using this technology.

References

- Altschul,S.F., Gish,W., Miller,W., Myers,E.W. and Lipman,D.J. (1990) Basic local alignment search tool. *J. Molec. Biol.*, **215**, 403–410.
- Buneman,P., Davidson,S.B., Hillebrand,G. and Suci,D. (1996) A query language and optimization techniques for unstructured data. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, Montreal, Quebec, June 1996.
- Cattell,R. (1994) *Object Data Management Revised Edition: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley.
- Chen,I.-M.A. and Markowitz,V.M. (1995) An overview of the Object-Protocol Model (OPM) and OPM data management tools. *Inf. Sys.*, **20**(5), 393–418.
- Clark,S.P., Evans,G.A. and Garner,H.R. (1994) Informatics and automation used in physical mapping of the genome. In Smith,D. (ed.), *Biocomputing: Informatics and Genome Projects*. Academic Press, pp. 13–49.
- Date,C.J. (1995) *An Introduction to Database Systems*. Addison-Wesley.
- Date,C.J. and White,C. (1991) *A Guide to Oracle*. Addison-Wesley.
- Dietrich,W.F., Miller,J., Steen,R., Merchant,M.A., Damron-Boles,D., Husain,Z., Dredge,R., Daly,M.J., Ingalls,K.A., O'Connor,T.J., Evans,C.A., DeAngelis,M.M., Levinson,D.M., Kruglyak,L., Goodman,N., Copeland,N.G., Jenkins,N.A., Hawkins,T.L., Stein,L.D., Page,D.C. and Lander,E.S. (1996) A comprehensive genetic map of the mouse genome. *Nature*, **380**, 149–152.
- Durbin,R. and Mieg,J.T. (1991) A *C. elegans* Database. Documenta- tion, code and data available from anonymous FTP servers at lirmm.lirmm.fr, cele.mrc-lmb.cam.ac.uk and ncbi.nlm.nih.gov.
- Goodman,N., Rozen,S. and Stein,L.D. (1994) Building a laboratory information system around a C++-based object-oriented DBMS. *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago de Chile, Chile, The Very Large Data Bases (VLDB) Endowment Inc.
- Goodman,N., Rozen,S. and Stein,L.D. (1998) The LabFlow system for workflow management in large scale biology research laboratories. *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, Montreal, Quebec, June 1998.
- Hudson,T.J., Stein,L.D., Gerety,S.S., Ma,J., Castle,A.B., Silva,J., Slonim,D.K., Baptista,R., Kruglyak,L., Xu,S.-H., Hu,X., Col- bert,A.M.E., Rosenberg,C., Reeve-Daly,M.P., Rozen,S., Hui,L., Wu,X., Vestergaard,C., Wilson,K.M., Bae,J.S., Maitra,S., Ganiat- sas,S., Evans,C.A., DeAngelis,M.M., Ingalls,K.A., Nahf,R.W., Lloyd,J., Horton,T., Anderson,M.O., Collymore,A.J., Ye,W., Kouy- oumjian,V., Zemsteva,I.S., Tam,J., Devine,R., Courtney,D.F., Ren- naud,M.T., Nguyen,H., O'Connor,T.J., Fizames,C., Faure,S., Gya- pay,G., Dib,C., Morissette,J., Orlin,J.B., Birren,B.W., Goodman,N., Weissenbach,J., Hawkins,T.L., Foote,S., Page,D.C. and Lan- der,E.S. (1995) An STS-based map of the human genome. *Science*, **270**, 1945–1955.
- Kerlavage,A.R., Adams,M.D., Kelley,J.C., Dubnick,M., Powell,J., Shanmugam,P., Venter,J.C. and Fields,C. (1993). Analysis and management of data from high throughput sequence tag projects. *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press.
- Kerlavage,A.R., FitzHugh,W., Glodek,A., Kelley,J.C., Scott,J., Shirley,R., Sutton,G., Wai-Chiu,M., White,O. and Adams,M.D. (1995) Data management and analysis for high-throughput DNA sequencing projects. *IEEE Eng. Med. Biol.*, **14**, 710–717.
- Lamb,C., Landis,G., Orenstein,J. and Weinreb,D. (1991) The Object-Store database system. *Commun. ACM*, **34**(10), 50–63.
- McGoveran,D. and Date,C.J. (1992) *A Guide to Sybase and SQL Server*. Addison-Wesley.
- McHugh,J., Abiteboul,S., Goldman,R., Quass,D. and Widom,J. (1997) Lore: a database management system for semistructured data. *SIGMOD Record*, **26**(3), 54–66.
- Orfali,R., Harkey,D. and Edwards,J. (1996) *The Essential Distributed Objects Survival Guide*. John Wiley & Sons.
- Ramakrishnan,R. and Ullman,J.D. (1995) A survey of research on deductive database systems. *J Logic Programm.*, **23**(2), 125–149.
- Rozen,S., Stein,L.D. and Goodman,N. (1995) LabBase: a database to manage laboratory data in a large-scale genome-mapping project. *IEEE Trans. Eng. Med. Biol.*, **14**, 702–709.
- Sargent,R., Fuhrman,D., Critchlow,T., Sera,T.D., Mecklenburg,R., Lindstrom,G. and Cartwright,P. (1996) The design and implementa- tion of a database for human genome research. *Proceedings of the Eighth International Conference on Scientific and Statistical Database Management*, Stockholm, Sweden, IEEE Computer Society Press, June 1996.
- Steen,R. (1997) Genetic maps of the rat genome. <http://www.genome.wi.mit.edu/rat/public/>. Whitehead Institute/MIT Center for Genome Research.
- Stein,L.D., Marquis,A., Dredge,E., Reeve,M.P., Daly,M., Rozen,S. and Goodman,N. (1994a) Splicing UNIX into a genome mapping laboratory. *Proceedings of the USENIX Summer 1994 Technical Conference*, USENIX.
- Stein,L.D., Rozen,S. and Goodman,N. (1994b) Managing laboratory workflow with LabBase. *Proceedings of the 1994 Conference on Computers in Medicine (CompMed94)*.
- Stein,L.D. (1996) How Perl saved the human genome project. *Perl J.*, **1**(2), 5–9.
- Ullman,J.D. (1988) *Principles of Database and Knowledge-Base Systems*. Computer Science Press.
- Wall,L., Christiansen,T. and Schwartz,R.L. (1996) *Programming Perl*. O'Reilly & Associates.
- WfMC (1996) Workflow Management Coalition. <http://www.aiim.org/wfmc/>. Workflow Management Coalition (WfMC), October 1996.
- WICGR (1997a) Genetic and physical maps of the mouse genome. <http://www.genome.wi.mit.edu/cgi-bin/mouse/index>. Whitehead Institute/MIT Center for Genome Research.
- WICGR (1997b) Welcome to the Whitehead Institute/MIT Genome Sequencing Project. <http://www-seq.wi.mit.edu/>. Whitehead Insti- tute/MIT Center for Genome Research.