

Chapter _

REQUIREMENTS FOR A DEDUCTIVE QUERY LANGUAGE IN THE MAPBASE GENOME-MAPPING DATABASE *

Nathan Goodman, Steve Rozen and
Lincoln Stein

{nat,steve,lstein}@genome.wi.mit.edu
Whitehead Institute for Biomedical Research
One Kendall Square, Building 300
Cambridge MA 02139
USA

ABSTRACT

MapBase is a database that stores information about short DNA sequences called markers and about the positions of markers along the chromosomes of an organism. It also stores information about experimental steps carried out on the markers. For modeling and performance reasons, we built MapBase on top of a C++-based object-oriented database management system. We describe MapBase's design, and examine the requirements that, together with the underlying database management system, have shaped it. Our experiences using MapBase show that its most pressing needs are for a more powerful ad hoc query facility and for more flexibility in schema evolution. In particular, we believe that a deductive query language will provide the support we need and improve the integration between MapBase and its clients.

1 OVERVIEW AND REQUIREMENTS

MapBase is a critical component of the genome-mapping efforts at the Whitehead Institute/MIT Center for Genome Research (the Genome Center). These

*To appear in R. Ramakrishnan, ed., *Applications of Deductive Databases* (tentative), Kluwer, 1994. This version available at <ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps>. An earlier version appeared in *Proceedings of the Workshop on Programming with Logic Databases In Conjunction with ILPS, Vancouver, B.C.* (R. Ramakrishnan, ed.), pp. 18–32, Oct. 1993.

efforts will require completion of over 2.5 million experiments, each of which requires several steps. Broadly speaking, the purpose of these experiments is to find short DNA sequences called *markers* and to determine their locations on the chromosomes of an organism, thereby generating a genome map. MapBase records both the experimental steps and the conclusions about markers derived from these experiments. For more details on MapBase's role in genome mapping please see [GRS92, SMD⁺94].

Several critical requirements have driven MapBase's design:

- R-1.* The need to model complex data types, for example *DNA sequences* and *genome maps*, without sacrificing performance. DNA sequences are like strings, but they have a restricted alphabet, and require special operations, such as reverse complementation (see below), that can't easily be provided in today's relational systems. A genome map is, abstractly, a sequence of groups of markers with inter-group distances. The map must also record multiple possible positions of markers for which experimental data is inconclusive.
- R-2.* The need to accommodate frequent changes in experimental protocols as genome-mapping technology evolves. In essence, the Genome Center is engaged in continual process re-engineering. As a result, a schema change is required at least every two months, and the frequency of schema changes is, if anything, increasing. In this respect, MapBase's requirements differ from those of some other (deductive) molecular biology databases (e.g. [SGTss, GST93, HMOP93]), which are primarily concerned with querying experimental results.
- R-3.* The need for a client/server architecture with the ability to supply data to clients written in many different languages. In addition to the MapBase database, the system includes a map construction program called MAP-MAKER [LGA⁺87], a program called PRIMER [LDL91] for analyzing markers, numerous small application programs, user interface programs, and programs to control laboratory machines. For pragmatic and historical reasons, these programs have been written in a variety of programming languages: C, C++, Smalltalk, Perl, and Lisp. The Macintosh is the desktop platform of choice among our users, who often enter MapBase data by means of Excel spreadsheets created on these machines. Most data analysis programs run on Unix workstations, although a few programs run only on Macintoshes or PCs.
- R-4.* The need for concurrent multi-user access. Different users must be able to retrieve data from MapBase and update it concurrently.

- R-5.* The need for reliability. MapBase must be able to survive both soft failures and disk failures without loss of data. It is also necessary to perform consistency checking to ensure that the data in MapBase make sense. This checking will be particularly important as the number of different client applications updating MapBase grows.

In addition to the requirements above, we have one meta-requirement. We wish to avoid becoming locked into any one object-oriented database management system (OODBMS). OODBMSs are new technology, and we want to be able to take advantage of future improvements regardless of the system that offers them.

It is worth mentioning at this point why deductive query languages are important for MapBase's application. As we discuss below, MapBase's design falls short of fully satisfying our requirements for schema flexibility (*R-2*) and a client/server architecture (*R-3*). A deductive query language would let us pose many queries that we now would have to code imperatively and compile into the database. Although most MapBase queries are not recursive, some are, and it is possible that if we had a deductive query language we would pose more recursive queries. Furthermore, if the deductive query language were extensible and supported user-defined functions and abstract data types coded in C++, we would have the option of prototyping queries in the deductive language, and then, if performance requirements so dictate, re-coding performance-critical predicates in C++.

The following section discusses how MapBase's design tries to satisfy requirements *R-1-R-5* using a C++-based OODBMS. The remaining sections discuss the kinds of queries that must be expressed, why a deductive query language makes sense, and how we provide one in MapBase's successor.

2 MAPBASE DESIGN

We decided to build MapBase on top of an OODBMS because of our requirement for data-modeling expressiveness without sacrificing performance (*R-1*). We also considered a relational database management system, Sybase [MD92], which is the most widely used relational system in molecular biology research. We decided, however, that, in the long term, it would be too hard to satisfy our modeling requirement, *R-1*, with Sybase. We chose ObjectStore [LLOW91] because we judged it to be the most mature of the OODBMSs at the time

the choice was made. We also considered ONTOS [ONT92], O₂ [O⁺91], VERSANT [Ver93], and GemStone [BOS91].

Because ObjectStore offers the ability to store instances of any C++ class, we were able to model complex data types, as required by *R-1*. In addition, ObjectStore provides excellent single-user performance, provided the database fits in a main memory cache.

Having selected ObjectStore, we faced a number of further systems challenges:

- C-1.* We needed to improve multi-user performance. We found it difficult to avoid concurrency hot spots in storage allocation and other low-level areas. We experimented with ObjectStore's versioning capability, but in our tests this imposed too great a performance overhead on both queries and updates.
- C-2.* We needed to provide roll-forward recovery.¹ Without roll-forward recovery a disk crash or a dirty crash of the database management system could destroy all updates since the last database backup.
- C-3.* We needed a way to support schema evolution, which was not fully supported in the first version of ObjectStore that we used: a schema change could invalidate the database and make it inaccessible. Since ObjectStore provided no way to reload the database when this happened, a schema change had the same effect as a disk crash.²
- C-4.* We needed to reduce the amount of time spent in transaction commits.
- C-5.* We needed to provide a language-independent application program interface (API) and interpreted query language.³

The keystone of our solution to these challenges was to implement MapBase as a multi-connection server that mediates between client requests and the underlying database. This server, from the point of view of the underlying database,

¹Roll-forward recovery from "archive" logs is planned for the next major release of ObjectStore [O'B94].

²ObjectStore Release 2.0 and above supports schema evolution. See [Obj93], *User Guide: DML*, pages 305–363. We have not used the ObjectStore schema-evolution facility, which involves writing a C++ program to modify the data in the database to fit the new schema. We believe our current solution (discussed below in bullet *C'-3*) to be easier and less error prone for our requirements, though it involved considerable initial development.

³The elegant ObjectStore query expressions described in [OHMS92] (and referred to as ObjectStore DML in [Obj93]) must be embedded in C++ and compiled.

is a single user. It therefore provides high throughput because of ObjectStore's good single-user performance. To avoid the need to implement our own locking and transaction roll-back, the server offers only single-statement transactions to its clients. It must also complete one transaction before starting another; it is single threaded. This characteristic sometimes severely increases response time, when the server is executing a long-running query. Increased response time is usually not a problem, however, because most of MapBase's clients run unattended. A single-threaded server offering single-statement transactions is not ideal, of course, but we deem it to be an acceptable tradeoff at present; naturally we would prefer both high throughput *and* multi-statement transactions.

To satisfy our requirement *R-3*, the need for a client/server architecture, the interface between application programs and the MapBase server is entirely text based. All commands and data are represented as character strings. Commands can be updates, retrievals, or control messages (such as set-date or commit-transaction), and usually take parameters. To put data into the database, an application program converts the data to text, and sends it as parameters to appropriate MapBase update commands. Retrievals are accomplished in two ways:

1. The MapBase server provides a simple, home-brew, interpretive query language that allows select operations, count aggregates and sorting.
2. Retrievals that cannot be expressed in this language are either coded in C++ and compiled into the MapBase server or performed by clients on data dumped from MapBase.

In either case, the server responds with a stream of textual results, which the client program converts to the desired internal representation.

With a single MapBase server interpreting text commands we were able to address challenges *C-1-C-5*:

- C'-1*. Multi-user performance is improved because there is no lock contention: the MapBase server is the only process using the underlying database.
- C'-2*. The server supplies roll-forward recovery by logging all updates to a separate disk. The log is a logical log; it contains the update commands in their text representation.

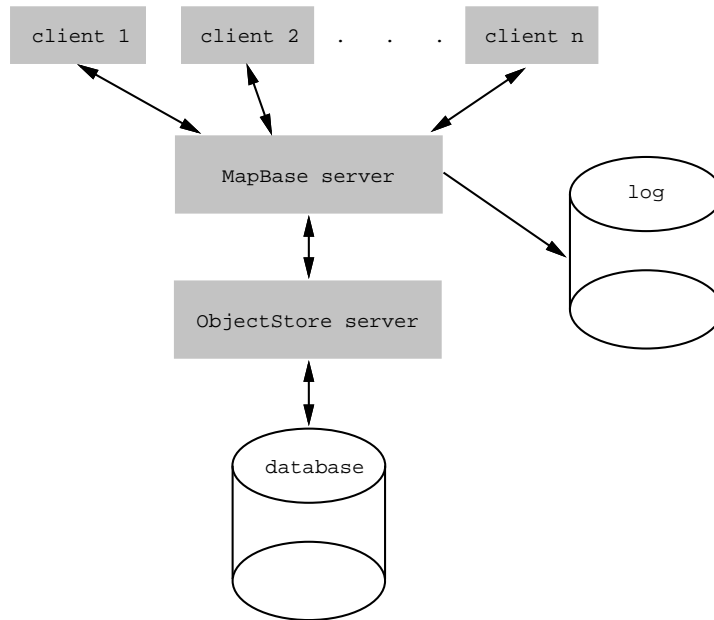


Figure 1 Architecture of the MapBase System.

- C'*-3. We evolve the schema by re-running all the roll-forward logs. If the schema change has invalidated any update statements in the logs, we write a script to appropriately revise the logs.
- C'*-4. The server amortizes the cost of commits by piggybacking them. The server performs a commit at approximately 10-minute intervals.⁴ Uncommitted updates can be recovered from the roll-forward log.
- C'*-5. The text commands and queries understood by the server constitute an API suitable for multi-lingual clients. Simple selection queries and count aggregates can be ad hoc. Other queries must be compiled into the MapBase server.

Figure 1 is a diagrammatic representation of MapBase's architecture.

⁴It is not clear why commits are expensive. Possibly the problem is that committed pages are forced back to the ObjectStore server on commit; moving data between the ObjectStore server and the MapBase server appears to be expensive.

Creating a server allowed us to satisfy most of our key requirements. However, the limitations of MapBase’s home-brew, ad hoc query facility remained a problem, and we expected them to become more of a problem as we move into a new phase of our laboratory experimentation. These limitations cramped the design of the MapBase schema because we tended to make only those schema changes that could be easily accommodated by MapBase’s query language. In addition, these limitations raised administrative difficulties, when a query that could not be expressed in MapBase’s language required the server to be recompiled.

3 WHAT WE WANT FROM A QUERY FACILITY

In light of the continuing problems posed by limitations of MapBase’s ad hoc query facility, we decided to seek a more powerful one that could be adapted to MapBase. It would also be desirable if such a facility reduced the need to evolve the schema at the ObjectStore level. To understand what capabilities we want in a query facility and why a deductive query language makes sense, in the remainder of this section we consider, as examples, some queries that MapBase must answer.

3.1 DNA Sequence Operations

Recall that DNA molecules are chains of *bases*, conventionally denoted by the letters G, A, T, and C.⁵ Each potential marker in MapBase is associated with a primary sequence of DNA bases. MapBase also keeps track of many *subsequences* of its primary DNA sequences, which it does by recording the length and starting position of the subsequence.⁶ We would not want to explicitly store these subsequences because of the possibility of introducing inconsistencies, and because we want to conserve space to keep the database in physical memory. Therefore, a ubiquitous operation on DNA sequences is to generate a subsequence of a DNA sequence.

Another common operation on DNA sequences is called *reverse complementation*. In a double-stranded DNA molecule, one strand is conventionally referred to as the *forward* strand, while the other is referred to as the *reverse*

⁵For more details on the biology please refer to e.g. [Lew94]

⁶For technical details please see [GRS92, Goo94, SMD⁺94].

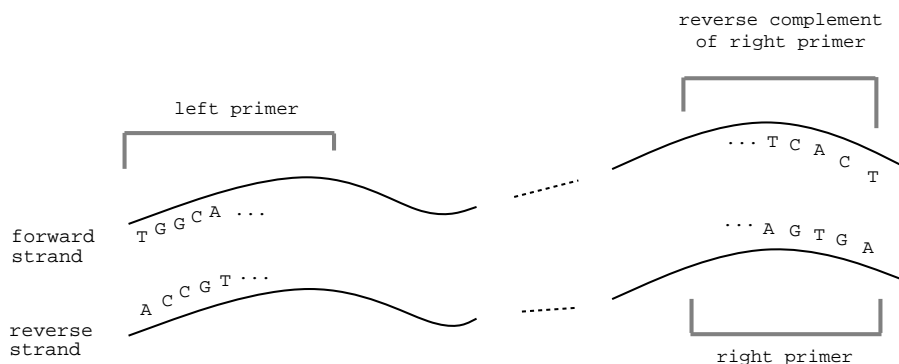


Figure 2 Left and Right Primers and Reverse Complementation. The forward sequence is read left-to-right, and the reverse sequence is read right-to-left. Only the forward DNA sequence is stored in the database.

strand. (The reasons are not important here.) Given a forward sequence, the reverse complementation operator produces the matching reverse sequence, and vice versa. To do this, reverse complementation reverses the order of the sequence, and then substitutes the bases according to the Watson-Crick base-pairing rules: $G \leftrightarrow C$ and $T \leftrightarrow A$. For example, the reverse complement of GATCCGGG is CCCGGAATC. Naturally, when MapBase stores a primary DNA sequence, it stores only one strand.

MapBase queries often require the reverse complement of a sequence. For example, in the laboratory mapping protocol, short DNA sequences are tagged at either end with even shorter sequences called *primers*. The left primer is part of the forward sequence and the right primer is part of the reverse sequence. To find the left primer we need its starting position and length on this sequence; to get the right primer we use the starting position and length of the matching nucleotide sequence on the forward sequence, and compute the reverse complement of this subsequence (see figure 2). For efficiency reasons, it is important that we be able to provide built-in predicates for ubiquitous operations like substring and reverse complementation.

3.2 Ready-For

The representation of a marker in MapBase has a large number of boolean “ready-for” methods that keep track of its progress through the laboratory

protocol. These methods are coded in C++ and linked into the MapBase server. For example, the steps for the early part of the analysis of a potential marker sequence are as follows:

1. Enter the sequence, S into MapBase.
2. Use a C-language client to duplicate-check S by comparing it to all sequences known to MapBase.
3. Use a (different) C-language client to check to see if any part of S closely resembles some sequence published in the national GenBank [BCC⁺91] sequence database, in which case S might be associated with a known gene.

⋮
(many other steps)
⋮

The C-language client in step 3 invokes a program called BLAST, which runs on a National Center for Biotechnology Information computer [AGM⁺90]. One of the ready-for conditions is `ready_for_BLAST`. Each night the client mentioned in step 3, above, queries MapBase for all markers where `ready_for_BLAST` is true; for each such marker it retrieves from MapBase the DNA sequence needed as input by BLAST. `Ready_for_BLAST` is true for a marker, m , if all the following criteria are true:

- m 's DNA sequence has been read,
- m has been duplicate checked against previous sequences and no duplicates were found (i.e. at step 2), and
- either (i) m has never been examined by BLAST or (ii) m was previously examined by BLAST, but at an earlier date than the last duplicate-checker run (perhaps because the DNA sequence was revised by re-doing step 1).

This example illustrates both the pluses and minuses of MapBase's design. On the plus side, ready-for relationships are easily represented in C++ as computed marker attributes. But, on the minus side, this representation means that part of the laboratory protocol is hard-coded into the database server. Even a trivial change to the protocol, such as doing the BLAST search before the duplicate check, forces us to recompile and relink the server.

3.3 Genetic-Linkage Mapping

The Genome Center carries out two kinds of genome mapping, *genetic-linkage mapping* and *physical mapping*. In genetic-linkage mapping (see [Lew94]), markers are first assigned to a chromosome and then ordered relative to one another based on the patterns by which they are co-inherited. Section 3.4 discusses physical mapping in detail.

The Genome Center's genetic-linkage mapping strategy generates two types of orderings. The "framework" ordering involves markers that map unambiguously to a unique position on a chromosome, and is represented as an ordered list of markers with inter-marker distances. The "placement" ordering involves markers whose position is less certain. These markers are placed on the map by giving their distance relative to the nearest framework marker, and they can have multiple placements. Map-position queries involve calculating a transitive closure over the framework and placement orderings. The query `?signed_distance("MPC101", "MPC3003", D)` (bind `D` to the signed distance or distances between these two markers) must do the following:

1. If `MPC101` and `MPC3003` are on different chromosomes, or if one or the other is not yet mapped, then `signed_distance` is false.
2. If they are both framework markers, find all intervening markers and add up the distances between them. If `MPC101` is above `MPC3003` on the chromosome, the sign of the result is positive; if `MPC3003` is above `MPC101` the sign is negative.
3. If one or both of the markers is a placed marker, find the nearest framework marker(s), calculate the distance between these two framework markers using rule (2), and adjust the distance by the placement-to-framework distance(s).

For a `signed_distance` involving a placed marker, one might be interested in maximum and minimum distances between the two markers, given all their reasonably likely positions. MapBase gives only the distance between most-likely positions, and handles this type of query by a C++ method (of the marker class) that implements the rules above. This method is in turn used to answer queries involving the relative positioning of markers, the most frequent of which is to request all markers that map between a given pair of markers.

3.4 Physical Mapping

Physical mapping uses methods different from those used in the genetic-linkage mapping discussed in section 3.3. For the kind of physical mapping carried out at the Genome Center, one must determine which members of a set of approximately 25,000 large DNA segments contain the much smaller fragments we use as markers.⁷ These larger fragments, which are called *YACs* (for reasons not important here) are about 10^6 bases long, while the markers are only about 300 bases long. We cannot simply determine the DNA sequence of YACs because they are far too large to be handled by current DNA sequence-reading technology. Instead we use a chemical test to determine if a YAC contains the marker sequence (i.e. if the marker sequence is a subsequence of the YAC sequence).

Pool Address

To reduce the number of individual experiments performed for each marker, small wells containing the YACs are conceptually arrayed in a three dimensional grid, and fractions of the wells with the same X , Y , or Z index are physically mixed to form *pools* for each of the X , Y , and Z dimensions—one pool for each index in each dimension, as shown in figure 3. For example, fractions of the fluid from each well with Z index 5 are taken and mixed to form the $Z = 5$ pool. Each marker is then tested against each of the pools. In a completely successful experiment, the marker is found in a single pool for each of the three dimensions, providing an (X, Y, Z) address that identifies a unique YAC that contains the marker. An example would be the address $(5, 8, 3)$, shown in figure 3 as a point at the intersection of three pools. However, due to experimental error, results can be incomplete or contradictory. If a marker is not found in some pool for one dimension, then the result is an incomplete address. An example would be $(5, 8, ?)$ —shown as a line in figure 3—which indicates that the YAC could be any YAC present in both the $X = 5$ pool and the $Y = 8$ pool. There can also be multiple positives in a dimension, which indicate that the marker is contained in more than one YAC. (Often, multiple positives are caused by unintentional mixing of YACs as they are handled in the laboratory.)

In cases of incomplete results, the laboratory sometimes repeats the experiment. Sometimes this repeat yields a unique address, but in other cases a different incomplete result is obtained. Suppose, for example, that the first time an experiment is run the address $(5, 8, ?)$ is determined, while on the repeat experiment, the address is $(5, ?, 3)$. These two incomplete results can be merged to obtain a complete address, namely $(5, 8, 3)$; MapBase handles incom-

⁷There are many other kinds of physical mapping.

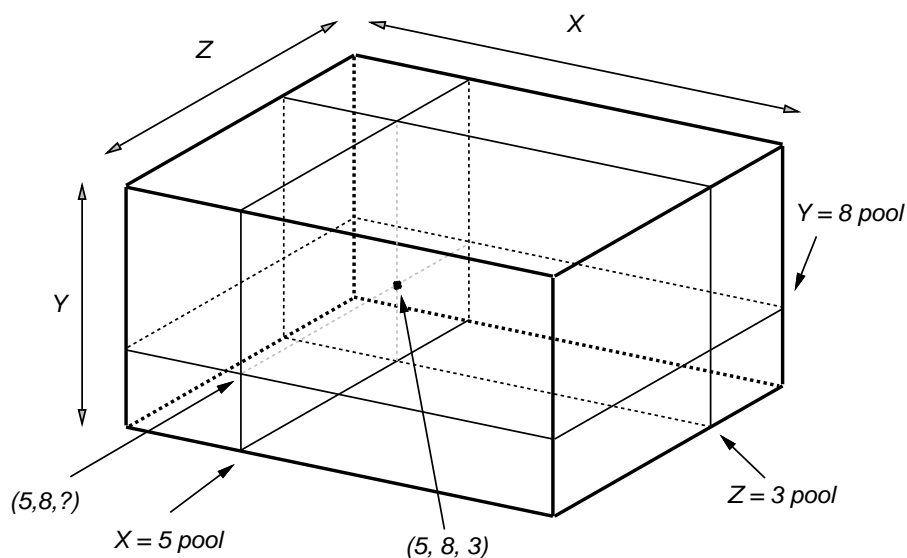


Figure 3 3-Dimensional Pooling Scheme. Conceptually, there is a small well of fluid (containing a YAC), at each (X, Y, Z) point for integer X , Y , and Z in some small range for each dimension.

plete addresses by calculating their intersection. This facility is implemented as a special-case query written in C++, since it cannot be handled in Map-Base's query language, though it could be easily expressed in a deductive query language.

Contig Assembly

Once pool addresses have been obtained by laboratory experimentation, they can be used to figure out which YACs contain the same markers and therefore overlap. The task of grouping overlapping YACs (and estimating a likely order for the markers) is one form of *contig assembly*.⁸

A number of problems can arise, however, in assembling YAC contigs. One significant problem is *chimerism* of YACs: a YAC can be composed of two or more fragments of DNA that are not contiguous in the genome of the organism being studied. A second significant problem is that of incomplete addresses,

⁸Another form of contig assembly groups and orders fragments of DNA according to overlapping sequences.

discussed above. Because of chimerism, there is a strong possibility that, even if two markers are found in the same YAC, they are not close together in the genome. Therefore we require *double linkage*: two markers must both be contained in two different YACs before we consider them to be (probably) close to each other.

The problem of finding markers that are probably close together can be expressed as a logic-programming query (or even an SQL query). Suppose we have the extensional relations

- $\text{hit}(\text{Marker}, \text{YAC})$, meaning *Marker* definitely hits (i.e. is contained in) *YAC* as determined by a complete pool address.
- $\text{amb_hit}(\text{Marker}, \text{YAC_set})$, meaning that *Marker* “ambiguously” hits each YAC in *YAC_set*, where the *YAC_set* is determined by a pool address with missing data in one dimension.

Two markers, p and q , are probably close together if either

- p and q both (unambiguously) hit two different YACs, Y and Z , i.e.

$$\text{hit}(p, Y), \text{hit}(q, Y), \text{hit}(p, Z), Y \neq Z, \text{hit}(q, Z)$$

or

- p and q both hit one YAC, Y , and p hits a second YAC, Z , which is contained in a YAC set that q ambiguously hits, i.e.

$$\text{hit}(p, Y), \text{hit}(q, Y), \text{hit}(p, Z), Y \neq Z, \text{amb_hit}(q, S), Z \in S$$

For other purposes, however, a biologist might not want to use the incomplete addresses. For example, one might be interested in a minimal set of YACs that connect two markers via a path of markers in which two markers are deemed to be close only if both unambiguously hit two different YACs. [HB94] reports on the use of CORAL and Hy+ [Men93] to explore raw contig data of the sort discussed in this section.

3.5 Workflow Analysis

Because the Genome Center seeks to improve throughput and reduce the cost of determining the location of a marker on a genome map, we regularly pose

queries about the workflow itself. For example, the last step we perform on a marker in genetic-linkage mapping is to run the map-construction program, MAPMAKER, to generate the probable location of the marker on a chromosome. The input to this program is the *genotyping* data obtained by observing the DNA of each mouse in a set of mice, and determining if that mouse inherited the marker from its grandmother or grandfather. (The mice in the set have only two grandparents, which they all share.)

Running MAPMAKER is a computationally expensive step, but we found that we often detected errors in the genotyping data only by running MAPMAKER. To estimate whether it would be worthwhile to create a step to check genotyping data *prior* to MAPMAKER runs, we wanted to find out how often the step for a MAPMAKER run is followed by a genotyping step (indicating that the MAPMAKER run revealed errors in the genotype data).

If we stored data about genotyping in an extensional relation,

```
genotyping_step(Marker, Typing_results, Timestamp),
```

and stored data about MAPMAKER runs in another extensional relation,

```
mapmaker_step(Marker, Map_location, Timestamp),
```

we could pose the query

```
?mapmaker_step(M,M1,T2),genotyping_step(M,Tr,T1),T2 < T1.
```

to find the genotyping steps that follow MAPMAKER steps. It would not be much harder to find only those pairs of genotyping and MAPMAKER steps between which there is no intervening genotyping or MAPMAKER step.

3.6 Other Criteria

Expressiveness is not our only criterion for an ad hoc query facility. Architectural issues, such as ease of adaptability to ObjectStore or another OODBMS, and the ease with which client programs can form queries and receive data

are also important. And, of course, we must still satisfy our original requirements $R-1$ – $R-5$. Among these, the requirement for schema evolution is still only partially satisfied in MapBase. As mentioned above, MapBase’s schema must continually evolve as the Genome Center revises its experimental workflow. For example, we order for each marker a quantity of the short DNA sequences called “primers” (discussed in 3.1). We receive 96 primers (left and right primers for 48 markers) in a box. A box has become a unit of work for a lab technician, so we recently started to record the association between marker and box.

In a deductive database, this change could be easily accommodated by adding a new base predicate `box(Marker_id, Box_id, Date, User)`, indicating that we received the primers for *Marker_id* in *Box_id*, with this information recorded on *Date* by *User*. Lest this seem trivial, consider that when we made this change in MapBase it required 108 lines of C++ code and required us to re-link the MapBase server.

4 A DEDUCTIVE SOLUTION

MapBase, though successful enough to be used in production for three years, is hampered by the limitations of its home-brew, ad hoc query language and by the difficulty of modifying its compiled-in schema to accommodate frequent changes in laboratory protocols. Several possible paths to improving this situation presented themselves:

- Adapting an object SQL language such as CQL++ [DGJ92], OQL[C++] [Bla94], or OQL [CAD⁺94] to MapBase. However, none of these languages has a DBMS-independent, interpreted implementation at present.⁹ We would like a DBMS-independent language so that we could use it with our existing DBMS, which we are comfortable with in terms of performance, robustness, and administration. The main reason that ad hoc queries require interpreted query execution rather than run-time compilation and loading is that an error in a dynamically loaded `.o` file could crash the MapBase server. It is unacceptable that a coding error in a single, ad hoc query make the database unavailable to all clients during recovery. One commercial company, Dharma, is trying to provide a portable,

⁹OQL[C++], however, is extremely similar to the interpreted object SQL provided by VERSANT.

interpreted, object SQL [Dha], but it seemed that a port to ObjectStore would be too much work for us.

- Moving to another OODBMS, such as VERSANT or O₂, that would provide fully general ad hoc queries and more flexibility in schema evolution.
- Adapting a deductive, possibly object-oriented query facility to our existing database. For example, CORAL++ [SRSS93] and LDL++ [AOTZ94] are designed in a way that might make it feasible for us to do this. Primary considerations would be performance and the ability to add abstract data types and user-defined functions to the system (basically our requirement R-1 above).
- Moving to a deductive database management system such as Aditi [VRK⁺90, HR93], ConceptBase [Con], EKS-V1 [VBKL90], or GlueNail [DMP93] (or CORAL or LDL). Primary issues here would be, again, performance and the ability to add abstract data types and user-defined functions to the system. Additional considerations would be the robustness and data-administration facilities of the underlying storage manager.

Based on a review of available technology for the options above, we decided to stay with ObjectStore and to generalize MapBase by adding a data dictionary and a simple declarative query processor. The result is MapBase's successor, which we call LabBase. LabBase is designed primarily to manage laboratory workflow, as opposed to performing analysis on laboratory results. The query language is a non-recursive datalog without (even non-recursive) rules, but with sets and lists as values, and with some aggregates. LabBase evaluates queries in a left-to-right, bottom-up fashion, without optimization.

LabBase's data model is based on the *general* notions of laboratory materials (as opposed to markers in MapBase) and histories of experimental steps. In MapBase it was necessary to write C++ code to add a new kind of experimental step, and adding a new kind of material was so difficult that we never attempted it. By contrast, in LabBase a data-dictionary update suffices to add either a new kind of experimental step or a new kind experimental step. LabBase offers a simple interface for adding built-in predicates, and it is also possible to add new built-in types. Such additions require re-compilation of the server, but such additions are much less frequent than the addition of a new kind of experimental step. (LabBase is also implemented in approximately 10,000 lines of code as opposed to 40,000 lines for MapBase.) Among the built-in predicates currently supported by LabBase are regular-expression matching and the reverse complementation operation discussed in section 3.1. We recently

added a new built-in type that provides memory-saving representations for sparse lists (which in turn permit more of the database to be cached in main memory).

The choice of a non-recursive datalog as the query language is a pragmatic compromise between implementation resources and requirements. Although term matching and rules would be extremely useful in making queries more concise and comprehensible, we chose to limit our implementation investment in the query language in the hope that, in the future, we will be able to adapt an existing deductive query language with optimization and more sophisticated query-evaluation strategies. Within the existing LabBase query language we can pose the ad hoc workflow-analysis queries and pool-address queries of sections 3.5 and 3.4. We can also pose various “ready-for” queries like those discussed in section 3.2, though we usually find it more convenient for client programs to maintain extensional sets of materials (e.g. markers) that are ready for particular steps. We have deployed LabBase for the Genome Center’s physical-mapping project, and detailed discussions are available in [RSG94] and [RSG]. MapBase is still used in several other projects.

Acknowledgements

This work was supported by funds from the National Institutes of Health, National Center for Human Genome Research, grant number P50 HG00098. We thank Mary-Pat Reeve, Andre Marquis, and the anonymous referees for their patient help in reviewing this paper. We thank Eric S. Lander for his support and encouragement.

REFERENCES

- [AGM⁺90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol. (England)*, 215(3):403–410, October 1990.
- [AOTZ94] Natraj Arni, Kayliang Ong, Shalom Tsur, and Carlo Zaniolo. The LDL++ system: Rationale, technology and applications. In *International Logic Programming Symposium*, November 1994. In press.

- [BCC⁺91] C. Burks, M. Cassidy, M. J. Cinkosky, K. E. Cumella, P. Gilna, J. E.-D. Hayden, G. M. Keen, T. A. Kelley, M. Kelly, D. Krsitofferson, and J. Ryals. GenBank. *Nucleic Acids Research*, pages 2221–2225, 1991.
- [Bla94] José A. Blakeley. OQL[C++]: Extending C++ with an object query capability. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press/Addison-Wesley, 1994.
- [BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):65–77, October 1991.
- [CAD⁺94] R.G.G. Cattell, Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis, and Drew Wade. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.
- [Con] ConceptBase Team, RWTH Aachen - Informatik V, Lehrstuhl Prof. Dr. Matthias Jarke, Ahornstr. 55 - 52056 Aachen, Germany. *ConceptBase V3.2 User Manual*. available by anonymous ftp from ftp.informatik.rwth-aachen.de (137.226.112.172), directory pub/CB/CB.3.2.
- [DGJ92] S. Dar, N. H. Gehani, and H. V. Jagadish. A SQL for a C++ based object-oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, March 1992.
- [Dha] Opening up proprietary databases, A white paper. Dharma Systems Inc., 15 Trafalger Square, Nashua NH 03063, USA.
- [DMP93] Marcia A. Derr, Sinichi Morishita, and Geoffrey Phipps. Design and implementation of the Glue-Nail database system. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, May 1993.
- [Goo94] Nathan Goodman. An object oriented DBMS war story: Developing a genome mapping database in C++. In Won Kim, editor, *Modern Database Management: Object-Oriented and Multidatabase Technologies*. ACM Press, 1994.
- [GRS92] Nathan Goodman, Mary-Pat Reeve, and Lincoln Stein. The design of MapBase: An object oriented database for genome mapping, December 1992. Whitehead Institute for Biomedical Research, technical report.
- [GST93] Susumu Goto, Norihiro Sakamoto, and Toshihisa Takagi. Object-oriented database with rule-based query interface for genomic computation. In *Proceedings of the Third International Symposium on Database Systems for Advanced Applications (Taejon, Korea)*, pages 65–72, 1993.

- [HB94] Eric Harley and Anthony Bonner. Genome map assembly using the Hy+ visualization system. In *International Conference on Intelligent Systems in Molecular Biology*, August 1994.
- [HMOP93] Ray Hagstrom, George S. Michaels, Ross Overbeek, and Morgan Price. Overview of GRACE—a database system for analysis of multiple genomes. In Trevor N. Mudge, Veljko Milutinovic, and Lawrence Hunter, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 1, pages 574–584. IEEE Computer Society Press, January 1993.
- [HR93] James Harland and Kotagiri Ramamohanarao. An Aditi implementation of a flights database. In Raghuram Ramakrishnan, editor, *Proceedings of the Workshop on Programming with Logic Databases*, pages 6–17, October 1993.
- [LDL91] S. E. Lincoln, M. J. Daly, and E. S. Lander. PRIMER: a computer program for automatically selecting PCR primers, May 1991. Whitehead Institute for Biomedical Research.
- [Lew94] Benjamin Lewin. *Genes V*. Oxford University Press, 1994.
- [LGA⁺87] E. S. Lander, P. Green, J. Abrahamson, A. Barlow, M. J. Daly, S. E. Lincoln, and L. Newberg. MAPMAKER: an interactive computer package for constructing genetic linkage maps. *Genomics*, 1(1):174–181, October 1987.
- [LOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.
- [MD92] D. McGoveran and C. J. Date. *A Guide to SYBASE and SQL Server*. Addison-Wesley, 1992.
- [Men93] A. O. Mendelzon. Declarative database visualization: Recent papers from the Hy+/GraphLog project. Technical Report CSRI-285, The University of Toronto, Computer Systems Research Institute, 6 King’s College Rd, Toronto, Ont, Canada M5S 1A1, 1993.
- [O⁺91] O. Deux et al. The O₂ system. *Communications of the ACM*, 34(10):34–48, October 1991.
- [O’B94] Patrick O’Brien. R3 & R4 product directions overview, March 1994. Talk presented at ObjectStore Northeast users group meeting.

- [Obj93] Object Design, Inc., 25 Burlington Mall Rd., Burlington MA 01803-4194, USA. Manual set for ObjectStore Release 3.0 for UNIX Systems, December 1993.
- [OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 403–412, June 1992.
- [ONT92] ONTOS, Inc., Three Burlington Woods, Burlington MA 01803, USA. *ONTOS DB 2.2 Developer's Guide*, February 1992.
- [RSG] Steve Rozen, Lincoln Stein, and Nathan Goodman. *Lab-Base User Manual*. Available at URL <ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-manual.ps>.
- [RSG94] Steve Rozen, Lincoln Stein, and Nathan Goodman. Constructing a domain-specific DBMS using a persistent object system. In *Sixth International Workshop on Persistent Object Systems*, September 1994. In press.
- [SGTss] Norihiro Sakamoto, Susumu Goto, and Toshihisa Takagi. A deductive database system for analyzing human nucleotide sequence data. *The International Journal of Bio-Medical Computing*, In press.
- [SMD⁺94] Lincoln Stein, Andre Marquis, Ert Dredge, Mary Pat Reeve, Mark Daly, Steve Rozen, and Nathan Goodman. Splicing UNIX into a genome mapping laboratory. In *USENIX Summer 1994 Technical Conference*, pages 221–229, June 1994.
- [SRSS93] Divesh Srivastava, Raghu Ramakrishnan, Praveen Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the 19th International Conference on Very Large Data Bases*, 1993.
- [VBKL90] Laurent Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI Workshop on Knowledge Base Management Systems*, 1990.
- [Ver93] Versant Object Technology Corporation, 4500 Bohannon Drive, Menlo Park CA 94025, USA. *VERSANT Object Database Management System Release 2 System Manual*, July 1993. Part Number 1003-0793.
- [VRK⁺90] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. The Aditi declarative database system. In J. Chomicki, editor, *Proceedings of the NACL'90 Workshop on Deductive Databases*, 1990.