LabFlow-1: a Database Benchmark for High-Throughput Workflow Management*

Anthony J. Bonner[1] bonner@db.toronto.edu

Adel Shrufi[1] shrufi@db.toronto.edu

Steve Rozen[2] steve@genome.wi.mit.edu

[1]University of Toronto
Department of Computer Science
10 King's College Rd
Toronto, ON, Canada
M5S 1A4

[2]Whitehead/MIT
Center for Genome Research
One Kendall Square
Building 300, Floor 5
Cambridge, MA 02139, USA

October 8, 1996

Abstract

Workflow management is a ubiquitous task faced by many organizations, and entails the coordination of various activities. This coordination is increasingly carried out by software systems called *workflow management systems* (WFMS). An important component of many WFMSs is a DBMS for keeping track of workflow activity. This DBMS maintains an audit trail, or event history, that records the results of each activity. Like other data, the event history can be indexed and queried, and views can be defined on top of it. In addition, a WFMS must accommodate frequent workflow changes, which result from a rapidly evolving business environment. Since the database schema depends on the workflow, the DBMS must support dynamic schema evolution. These requirements are especially challenging in high-throughput WFMSs—*i.e.*, systems for managing high-volume, mission-critical workflows. Unfortunately, existing database benchmarks do not capture the combination of flexibility and performance required by these systems. To address this issue, we have developed *LabFlow-1*, the first version of a benchmark that concisely captures the DBMS requirements of high-throughput WFMSs. LabFlow-1 is based on the data and workflow management needs of a large genome-mapping laboratory, and reflects their real-world experience. In addition, we use LabFlow-1 to test the usability and performance of two object storage managers. These tests revealed substantial differences between these two systems, and highlighted the critical importance of being able to control locality of reference to persistent data.

Portions of this report appear in:

- *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 1996.
- *The OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, 1995.
- *The NSF Workshop on Workflow and Process Automation in Information Systems*, 1996.

These papers and the entire report are available at the following web page:

`http://www.db.toronto.edu:8020/people/bonner/bonner.html`

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

Contents

1 Introduction

1.1 Overview

Workflow management is a ubiquitous task faced by many organizations in a wide range of industries, from banking and insurance, to telecommunications and manufacturing, to pharmaceuticals and health care [22, 50, 38, 19]. The task is to coordinate the various activities involved in running an enterprise. Increasingly, this coordination is carried out by a software system called a *workflow management system* (WFMS). The activities themselves may use a variety of software components, including files, databases, application programs, and legacy systems, which may run on a variety of hardware platforms and operating systems, which may be located at a variety of sites. For example, in a large genome laboratory, workflow management software knits together a complex web of manual and automated laboratory activities, including experiment scheduling and setup, robot control, raw-data capture, multiple stages of preliminary analysis and quality control, and release of finished results. Appropriate software is necessary to make coordination of these activities both intellectually manageable and operationally efficient, and is a prerequisite for high-throughput laboratories. This software includes a DBMS component for tracking and controlling workflow activity. This paper addresses the requirements of this DBMS.

LabFlow-1 is a benchmark that concisely describes the database requirements of a WFMS in a high-throughput genome laboratory. Although it is based on genome-laboratory workflow, we believe that LabFlow-1 captures the database requirements of a common class workflow management applications: those that require a *production workflow system* [29]. In production workflow, activities are organized into a kind of production line, involving a mix of human and computer activities. Examples in business include insurance-claim or loan-application processing. Production workflow systems are typically complex, high-volume, and central to the organizations that rely on them; certainly these characteristics apply to the laboratory workflow-management systems used in high-throughput genome laboratories. Many production workflows are organized around central materials of some kind, which the workflow activities operate on. Examples of central materials include insurance claims, loan applications, and laboratory samples. As a central material is processed, workflow activities gather information about it.

Production workflow systems include the class of *Laboratory Information Management Systems*, or LIMS [41, 3, 38]. LIMS are found in analytical laboratories in a wide range of industries, including pharmaceuticals, health care, environmental monitoring, food and drug testing, and water and soil management. In all cases, the laboratory receives a continual stream of samples, each of which is subjected to a battery of tests and analyses. Workflow management is needed to maintain throughput and control quality [37]. Data models and software environments have also been developed with laboratory workflow in mind [15, 39].

Much of the research on workflow management in computer science has focussed on developing extended transaction models for specifying dependencies between workflow activities, especially in a heterogeneous environment [22, 40, 18, 27, 26, 20, 48]. However, the *performance* of WFMSs has so far received little attention. The need to study performance arises because commercial products cannot support applications with high-throughput workflows. As stated in [22],

> Commercial workflow management systems typically support no more than a few hundred workflows a day. Some processes require handling a larger number of workflows; perhaps a number comparable to the number of transactions TP systems are capable of handling. For example, telecommunications companies currently need to process ten thousand service provisioning workflows a day, including a few thousand service provisioning workflows per

hour at peak hours. Commercial workflow management systems are currently not capable of handling such workloads.

Note that each workflow may involve many transactions.

High-throughput workflows are also characteristic of large genome laboratories, like the Whitehead Institute/MIT Center for Genome Research (hereafter called "the Genome Center"). Workflow management is needed to support the Genome Center's large-scale genome-mapping projects [17]. Because of automation in instrumentation, data capture and workflow management, transaction rates at the Genome Center have increased dramatically in the last three years, from processing under 1,000 queries and updates per day in 1992 [23], to over 15,000 on many days in 1995. Of course, peak rates can be much higher, with a rate of 22.5 updates and queries per second recently observed over a 5-minute period. These rates are expected to increase by another order of magnitude in the near future if the Genome Center begins large scale sequencing of the Human genome [17]. Moreover, unlike the simple banking debit/credit transactions of some TPC benchmarks [49], these transactions involve complex queries, and updates to complex objects, such as arrays, sequences, and nested sets.

In this paper, we take a first step towards measuring the performance of workflow management systems. We do not attempt to study all the components that affect WFMS performance, such as networks, hardware platforms, and operating systems. Instead, we focus on one dimension of the problem: the performance of DBMSs that control and track workflow. Like other components, the DBMS can become a workflow bottleneck, especially in high-throughput applications.

1.2 DBMS Requirements

Workflow management has numerous DBMS requirements. First, it requires standard database features, such as concurrency control, crash recovery, consistency maintenance, a high-level query language, and query optimization. Workflow for advanced applications (such as laboratory workflow) also requires complex data types, as are found in object-oriented databases. A DBMS supporting production workflow management must provide this functionality on a mixed workload of queries and updates. In addition, it must provide two other features, which are characteristic of production workflow systems:

Event Histories. The DBMS must maintain an audit trail, or event history, of all workflow activity. From this history, the DBMS must be able to quickly retrieve information about any material or activity, for day-to-day operations. The history is also used to explore the cause of unexpected workflow results, to generate reports on workflow activity, and to discover workflow bottlenecks during process re-engineering. The DBMS must therefore support queries and views on an historical database.[1] We note that many commercial laboratories are legally bound to record event histories, since "Accountability is critical in tracking who is responsible for data and its approval for release" [35]. Salient examples include clinical drug trials and environmental testing.

Dynamic Schema Evolution. A hallmark of modern workflow management is that workflows change frequently, in response to rapidly changing business needs and circumstances [19, 50, 22]. Typically, a workflow will acquire new activities and existing activities will evolve. In both cases, the changed workflow generates new kinds of information, which must be recorded in the database. This requires changes (usually additions) to the database schema, preferably while the workflow is in operation (which is called *dynamic workflow modification*).

[1] In this respect, the event history is similar to a sequence of transaction records as studied in [28]. The main difference is that workflow management requires greater functionality on a shorter history.

It is worth observing that because the database is historical and the schema is evolving, data at different points in the history will be stored under different schemas. Thus, an historical query or view may access data with many different schemas. This presents a challenge both to database design and to the formulation of queries and views. For instance, an application program may have to query an object's schema, as well as its value.

Existing database benchmarks do not capture the above requirements. This should not be surprising, as it has been observed by researchers working on OODBMS benchmarks that advanced applications are too complex and diverse to be captured by a single benchmark [12, 14]. A quick glance at several recent benchmarks illustrates their diversity of characteristics and requirements. For instance, the OO1, OO7 and HyperModel benchmarks [13, 12, 4] are concerned with the traversal of large graphs, which is a requirement of engineering and hypertext applications. In contrast, the SEQUOIA 2000 benchmark [54] is concerned with the manipulation of large sets of spatial and image data, such as those found in geographic information systems (GIS). The Set Query benchmark [43] is concerned with queries for decision support, including aggregation, multiple joins and report generation. (Such queries also arise in workflow management—for process re-engineering—but they are only part of the story.) Like these benchmarks, LabFlow-1 specifically targets a broad application area: workflow management. This application is characterized by a demand for *flexible* management of a stream of queries and updates, and of historical data and schema.

On the surface, LabFlow-1 might appear similar to some TPC benchmarks [49, 33], which are also based on a stream of transactions that construct a history. As discussed in Section 9, however, the way in which the stream is generated (the workload) is very different. Moreover, the history stored in TPC databases is not queried or indexed,[2] and there is no concern for flexibility and schema evolution. Finally, TPC databases are explicitly relational and contain no complex data types. The TPC benchmarks were not intended to capture the requirements of workflow management systems.

Unfortunately, many DBMSs do not yet fully support the requirements of production workflow systems as described above. Certainly, a surprising number of commercial products showed serious flaws in simple tests performed by the Genome Center in 1991. Fortunately, one can build a specialized DBMS that supports workflow on top of a storage manager that does not. This approach is taken at the Genome Center. Their specialized DBMS (called *LabBase* [46, 53, 45]) provides the needed support for event histories and schema evolution on top of an object storage manager. LabBase provides a historical query language, as well as structures for rapid access into history lists. It also transforms data from the user's database schema (which is dynamic) into the storage manager's schema (which is static). Besides providing support for workflow, this approach also provides portability, since different object storage managers can be "plugged into" the DBMS. In this way, we can test a wide range of existing storage managers. The benchmark implementation in this paper is based on this idea. We emphasize, however, that the LabFlow-1 benchmark does not depend on LabBase, which is an implementation detail. In the future, we hope to use our benchmark to test the support for workflow management in "off-the-shelf" DBMSs.

1.3 LabFlow-1

The LabFlow-1 benchmark is based on the data and workflow management needs of the Genome Center, and reflects their real-world experience. The benchmark has several goals. One goal is to provide a tool for the Genome Center to use in analyzing object storage managers for LabBase. Other goals are (*i*) to provide a general benchmark for databases supporting workflow-management systems, and (*ii*) to provide developers of next-generation DBMS technology with a set of requirements based on the

[2] In particular, the "`history`" relation never appears in the `from` clause of an SQL statement.

characteristics of a real-world application domain. Developers of new technology often have few if any realistic applications against which to measure their systems. As with any benchmark, the challenge in designing LabFlow-1 was to create a database schema and workload that are realistic enough to be representative of the target class of applications yet simple enough to be feasibly implemented on a variety of systems.

Although LabFlow-1 is intended to be a general benchmark for DBMSs, this paper uses it to compare storage managers only. This is achieved by running the benchmark on versions of LabBase implemented on top of different storage managers, as described above. This paper compares ObjectStore (version 3.0) [32, 42] and Texas (version 0.3) [51, 59]. Compared to relational systems, these storage managers have been used in few production applications, so this analysis is interesting in its own right. Since they are a relatively novel technology, we compare these storage managers not only in terms of performance, but also in terms of client interface, tuning options, and system-administration facilities. In a similar fashion, LabFlow-1 can be used to compare other DBMS components, such as query optimizers and historical access structures.

The benchmark presented in this paper is a preliminary version, intended for a single database user; hence the name LabFlow-1. This choice arises from the architecture of the Genome Center's production DBMS, and also from a belief that it is useful to understand single-user performance before attempting to understand multi-user performance. We plan to develop in the future (*i*) multi-user and client/server workflow benchmarks, and (*ii*) benchmarks to evaluate and compare deductive database systems as query processors for workflow databases.

In sum, LabFlow-1 is the first version of a benchmark for DBMSs that control and track workflow. It is designed for applications with the following characteristics and requirements:

- high volume, mission critical workflows;
- frequent workflow change and process re-engineering;
- an audit trail of workflow activity;
- complex-structured data.

Benchmark software is available at the following web site:

`ftp://db.toronto.edu/pub/bonner/papers/workflow/software/`

The paper is organized as follows. Section 2 provides an overview of workflow management at the Genome Center. Section 4 presents the database schema for the LabFlow-1 benchmark, and Section 5 outlines the benchmark implementation. Section 6 presents the workflow language in which benchmark queries and updates are expressed. Using the query language, Section 7 defines a view of the event history that augments the benchmark schema and is an important feature of LabFlow-1. Many queries in the benchmark operate this view. Section 8 provides detailed examples of queries expressed in this language that are typical of those in the benchmark. Section 9 describes the benchmark itself and its workload. Section 10 presents the results of running the benchmark with two different storage managers (ObjectStore and Texas), and discusses the differences in observed performance. Finally, Appendix A presents the detailed database schema for the benchmark, and Appendix B provides a detailed description of how the benchmark workload was generated.

2 Workflow at the Genome Center

As described above, LabFlow-1 is a benchmark that captures the DBMS requirements of high-throughput workflow management. Before specifying the benchmark and its workflow model, we give a concrete example of high-throughput workflow. The example should help to motivate and clarify the benchmark. Our workflow example comes from the Whitehead Institute/MIT Center for Genome Research (the "Genome Center"), which is engaged in several large-scale genome mapping projects [17]. Each project involves the completion of tens of millions of experiments, organized into a network of factory-like production lines. coordinating the flow of materials through the production lines, and recording (and querying) the history of steps and the experimental results they produce are the main data and workflow management requirements. These requirements are particularly challenging because genome-mapping technology is evolving rapidly, with the consequence that the Genome Center is engaged in continual process re-engineering.

2.1 Production Workflow

Workflow at the Genome Center is an example of *production workflow* [29]. In production workflows, activities are organized into production lines, with a mix of human and computer activities. Systems for managing production workflow are typically complex, high-volume, and central to the organizations that rely on them. Many production workflows are organized around central materials of some kind, which the workflow activities operate on. As a central material is processed, workflow activities gather information about it. Certainly, these characteristics are all true of the workflows at the Genome Center.

Each production line at the Genome Center is a collection of steps that process a single kind of material. For example, a production line might be set up to process strands of DNA (or clones). Each step on a production line takes place at a work station. A step might involve a scientist carrying out a experiment, or a robot assembling simple materials into a composite material, or a computer performing complex statistical calculations. In any case, a step produces information about a material, information that must be entered into the database. In effect, as a material moves through a production line, a data structure about the material is gradually assembled in the database. For each of the millions of materials in the laboratory, a data structure records what has so far been learned about that material (such as its length, its position on a genetic map, its DNA sequence, its similarity to known genes, its interactions with various chemicals, its enzyme cutting sites, etc.) There are numerous production lines in the Genome Center, each for a different kind of material, and each with different kinds of steps. A typical genome mapping project might involve 40 kinds of steps and a dozen kinds of materials. The corresponding database would contain on the order of 10^6 material instances, and on the order of 10^7 step instances.

The various production lines are not independent. Interaction comes about when different production lines share a step. Such steps involve more than one material, one from each production line. This can happen in several ways. For example, a step might test the interaction of two kinds of material; *e.g.*, an experiment might determine whether two kinds of DNA stick together, where the two kinds of DNA are prepared by different production lines.[3] Testing for interactions between materials is central to genome mapping and sequencing, and to scientific experiments generally. Production lines also share steps when one line prepares auxiliary materials used by another line. In this case, the two production lines might form a kind of T, where the final product of one line is used by a step in the middle of the other line. Two production lines may also share several steps. This can happen if the two lines are synchronized, interacting at several points, as with the two production lines in Appendix B. In sum, each production line may share steps with other lines, which in turn may share steps with other lines, thus giving rise to a network of production lines.

[3] The technology used to prepare and process DNA depends on the length of the DNA sequence. Differences in length of an order of magnitude generally imply different technologies.

The production lines in the Genome Center are flexible. For instance, a production line is not really a line at all, but a directed graph (called a *workflow graph*). In general, the next step to be performed on a material is not pre-determined, but depends on the outcome of previous steps. Often, if a step does not produce good results, earlier steps are repeated. In addition, workflow at the Genome Center changes constantly, as the center tries to increase efficiency and adopt new technologies. The most common change is for a step to record more information. Other changes include adding a new step to a production line, or refining a single step into a combination of separate steps. Thus, the kinds of steps, the kinds of data gathered, and the order of steps all tend to change. This flexible, changing workflow demands a database with a flexible, changing schema.

2.2 Workflow Management

Workflow management can be divided into three main responsibilities: *workflow modeling*, *workflow control* and *workflow tracking*. This section continues our example by describing how workflow is modelled and controlled at the Genome Center. The rest of the paper then focuses on workflow tracking, in which the DBMS play a central role.

2.2.1 Workflow Modeling

A workflow model specifies ordering relationships and data dependencies among workflow steps. This model may be implicit (embedded in application programs), or explicit (specified in a declarative formalism). At present, the Genome Center takes an intermediate position: the ordering of workflow steps is made explicit in workflow graphs, while data dependencies are implicit in application programs [53]. Workflow graphs are based on the idea that each material has a workflow *state*, and that as the material is processed, it moves from one state to another.

The state of a material determines where it is in a production line and what step to apply to it next, if any. For instance, the possible states of a strand of DNA might include the following:

```
waiting_for_sequencing  waiting_for_resequencing
   sequence_completed  unsequenceable
```

The first two states indicate that the DNA is waiting for some kind of sequencing step. The last two states are terminal states, indicating that the DNA has reached the end of a production line.

Each time a material is processed by a step, the material moves from one state to another. From each state, a material may move into one of several other states, depending on the outcome of the step. For instance, after sequencing, a strand of DNA might move from state `waiting_for_sequencing` to any one of the following steps, depending on the outcome of the sequencing step:

```
sequence_completed    unsequenceable    waiting_for_resequencing
```

At present, control over state transitions is embedded in application programs. However, the set of legal transitions is represented declaratively as a set of annotated, directed graphs, called *workflow graphs*. Each workflow graph corresponds to a single production line, and thus represents the processing of a single kind of material. A node in a workflow graph represents a material state, and an arc represents a legal state transition. Each arc has a label, which serves as a name for the transition. Each arc may also have a constraint associated with it. A material cannot make the transition across an arc unless the constraint is satisfied. As a material is processed, it follows a path through the graph from a source to a sink.

A workflow graph may have several sources and several sinks. Typically, one source is where completely unprocessed materials enter the workflow, while other sources are entry points for partially processed materials obtained from other labs. Likewise, one sink is usually where completely processed materials leave the workflow, while other sinks are exit points for materials that cannot be processed past a certain point. A workflow graph may also have cycles. A cycle allows materials to be returned to earlier workflow states for re-processing. This typically occurs when a workflow activity produces an unexpected or unsatisfactory result. Application programs must ensure that a material does not cycle through the graph indefinitely.

Often, there is an implicit correspondence between workflow steps and material states. For instance, all materials in the state `waiting_for_sequencing` are waiting for a sequencing step. In practice, many states have names of the form `waiting_for_X`, where `X` is a kind of step. However, the correspondence between states and steps is not one-to-one. For instance, a terminal state (*i.e.*, a sink in a workflow graph) does not correspond to any step. Moreover, a single step can correspond to two (or more) states. This can happen in two ways. (*i*) Materials in different states of the same production line can wait for the same step. *e.g.*, Strands of DNA in states `waiting_for_sequencing` and `waiting_for_resequencing` might all be waiting for the same sequencing step. (*ii*) Materials of different kinds in different production lines can wait for the same step. This happens when a step is shared by two production lines. Thus, the nodes of a workflow graph do not correspond exactly to the steps of a production line.

Workflow graphs are a preliminary workflow model, recently adopted by the Genome Center, that help to control workflow. However, there is much room for improvement. For instance, workflow graphs do not capture the data dependencies between workflow steps. Nor do they capture the interactions between production lines. Nor is there an explicit link between workflow graphs and the steps of a production line. At present, this information is all embedded in application programs, as described in the next subsection.

It is worth noting that the idea of assigning states to materials contrasts with transactional workflow, in which states are assigned to long-running activities (*e.g.*, [48, 1, 58, 21]). This difference might be resolved if each material were associated with a single long-running activity. This activity would exist as long as the material is being processed, and would correspond to the sequence of workflow steps that process the material. In this case, material state and activity state might be identified. However, unlike many extended transactions, workflow steps in the Genome Center are never rolled back or compensated, since this would amount to throwing away experimental results. Experimental results, even bad or sloppy results, must be permanently recorded in the database.

2.2.2 Workflow Control

Because of the flexibility of the production lines, special care must be taken to coordinate the millions of materials and experiments. At any given time, it should be easy to identify what materials are being processed by what steps, and what steps will be applied to them next. In addition, as each step is carried out, the right programs have to execute in the right order, user input has to be obtained at the right points, and so on. Thus, a key requirement at the Genome Center is to ensure that workflow steps are carried out in the right order,[4] and to control what happens as they are carried out.

Workflow at the Genome Center is controlled by a library of application programs, one program for each kind of workflow step. These programs are guided in part by the workflow graphs described above. Each program provides an interface between the database and the scientist (or robot or other entity) who carries out the step. The program handles the data analysis and database transactions required by the step. The data analysis is domain specific and may require computing such things as optimal PCR temperatures from DNA sequences. In general, each step will involve very different

[4]This order may be different for different materials, even within a single production line.

data analyses, which can be invoked at any time. In contrast, the database transactions are more structured and predictable. They can be divided into five categories, corresponding to five distinct phases of each workflow step:

1. Initially, a step queries the database to find out what materials are waiting to be processed by the step. This involves querying materials based on their state. For instance, to find out what materials are ready to be sequenced, a step might retrieve the names of all materials whose state is `waiting_for_sequencing`. Typically, the names of these materials are then printed out, along with an indication of where the materials are located (*e.g.*, at a central repository or at another workstation) so a scientist can retrieve them.

2. During a step, the results of other workflow steps may be needed. For instance, to compute the optimal temperature for a polymerase chain reaction (PCR) for a sample of DNA, the sequence of the DNA may be needed. This sequence will have been recorded by an earlier workflow step. The current step issues a database query to retrieve the information. Note that the recording and querying of workflow results represents a flow of data from one workflow step to another. This introduces data dependencies into the workflow, which often determine the order of steps. (*e.g.*, DNA sequencing must precede PCR.)

3. During a step, new materials may be created. For instance, a strand of DNA might be broken up into several smaller strands, or modified copies of the strand might be made, or several strands might be ligated into a single strand. The new materials may be used later in the production line, or they may form the input to some other production line (as in Appendix B). In all cases, the creation of the new materials is recorded in the database, and new database objects are constructed to represent them, as described in Section 8.3

4. When a step has been completed, a record of the step is inserted into the database. The record includes information about who carried out the step, when it was carried out, what the results were, and the conditions under which the step was carried out. The results include information gleaned about the material under study (such as its DNA sequence or its position on a genetic map), and an indication of whether the step was successful. In this way, an audit trail (or event history) of workflow activity is built up, one step at a time.

5. Finally, based on the results of the step, a transaction changes the state of the material. This transaction corresponds to an arc in the workflow graph. The transaction first verifies that the state of the material is the same as the initial state of the arc. It then checks any constraints associated with the arc. If these conditions are satisfied, then the transaction updates the state of the material, changing it to the terminal state of the arc. If the conditions are not satisfied, then the transaction logs an error, informing the system administrator of an illegal request by an application program.

For each step, these transactions are invoked at the appropriate times by the application program. These application programs are domain specific, with different user interfaces and data analyses at each step. Thus, from the user's perspective, each step and its application program is different. However, from a database perspective, all steps and their application programs are similar: each has five phases, defined by the five kinds of transactions above. Later in the paper, we give the composition of the LabFlow-1 benchmark in terms of these transaction categories (in Tables 2 and 5).

3 Workflow Tracking in LabFlow-1

An important component of many workflow management systems is a DBMS for tracking workflow activity. This DBMS maintains an audit trail, or event history, that records the results of each activity. Like other data, the event history can be indexed and queried, and views can be defined on top of it. In genome laboratories, the event history serves much the same function as a laboratory notebook: it records what was done, when it was done, who did it, and what the results were. It can also be used to analyze workflow activity, for example to find rate-limiting steps or to investigate anomalous final results. All laboratory information management systems produce an event history. Managing this event history is the primary function of the DBMS, and the main subject of this paper. This section provides an overview of data and workflow management in the LabFlow-1 benchmark from the perspective of the DBMS, along with pointers to the rest of the paper. To keep the discussion concrete, we frame it in terms of laboratory workflow and LabBase.

The database contains two main kinds of object: *materials* and *steps*. In object-oriented terms, the database has a material class and a step class, with subclasses representing different kinds of materials and steps. Step and material instances (objects) are created in two distinct situations. (*i*) As the laboratory receives or creates new materials, new material instances are created in the database to represent them. Each instance records the material's name; and if it is a compound material (*e.g.*, an array of other materials), then its structure is also recorded. (*ii*) Each time a material is processed by a workflow step, a new step instance is created in the database to represent the activity. The step instance records the results of the workflow step (*e.g.*, measurements of length and sequence) and the conditions under which the step was carried out (*e.g.*, temperature and salinity). Step and material classes are described in more detail in Section 4; and examples of step and material creation are given in Section 8.3.

As a material is processed by a workflow, more-and-more step instances are created in the database, all associated with the same material. These steps constitute the material's *event history*. Since steps may be repeated, the event history of a material may contain several different (though similar) versions of a step. Usually, the most recent version is of greatest interest to scientists, since it represents the most up-to-date results. The notion of most-recent results is thus a central feature of LabBase's query language, and is developed in Section 6.2. Internally, LabBase uses special indexes to rapidly retrieve most-recent results from the event history of each material, as discussed in Section 5.2.

Notice that in the above representation, the database schema depends on the workflow. For each kind of workflow step, there is a class in the database, and for each measurement made by the step, the class has an attribute. Consequently, workflow changes are reflected in the database as schema changes. In particular, as laboratory steps change, the corresponding classes also change. For instance, if scientists decide that a particular step should measure more properties, then attributes must be added to the corresponding class in the database. Likewise, if scientists add a new step to the workflow, then a new class is added to the database schema. If scientists split a complex step into a combination of smaller steps, then new classes are introduced into the schema to represent the smaller steps. These ideas are illustrated in Section 8.1.

Classes are never removed from the database schema, even if the corresponding steps are no longer carried out in the laboratory. This is because the database is historical; so old classes are needed to store and interpret old data, *i.e.*, data gathered under earlier workflows. Thus, the database schema expands (as classes are created and attributes are added to them), but never shrinks. In effect, as a step evolves, new versions of the step are created. Each step object is associated forever with the same version of a step class; so schema changes do not require data re-organization. A similar approach to schema evolution can be found in [52]. The main difference is that LabBase does not use an explicit versioning mechanism. Instead, it identifies versions of objects by their attribute set, as illustrated in

Section 8.1.

Because the database is historical and the schema is evolving, data is stored under many different schema at any given time. Thus, an historical query or view may access data with many different schema. This requirement reflects the flexibility demanded by workflow management, and presents a challenge to database design and to the formulation of database queries and views. These challenges were a major factor in the design of LabBase. The solutions are described in Sections 4, 5, 6, 7 and 8.1. The main issues are outlined below, and discussed in detail in later sections.

Notice that the data representation described above is *event oriented*. That is, information about a step is kept in one place, but information about a material is scattered among different steps. This provides a straightforward record of laboratory activity, but an awkward representation of materials. In particular, retrieving information about a material requires a detailed knowledge of laboratory workflow. For each property of a material, one must know what step(s) measured its value. Moreover, because workflows change constantly, a detailed knowledge of workflow changes and workflow history are also needed. This is a common occurrence in laboratory notebooks. When the notebook is stored in a database, the problem is compounded, since application programs may have to be changed each time the workflow changes.

To alleviate these problems, the database provides a view that is *material oriented*, *i.e.*, in which information is associated with materials, not steps. In the database, observed values (*e.g.*, sequence, length, and mass) are attributes of *steps*; while in the view, they are attributes of *materials*. Using the view, an application programmer can retrieve the length of a DNA molecule without knowing what step measured the length. In this way, the view isolates the application programmer from the details of workflow and workflow change. The view may be materialized, but LabBase does not currently do this. Instead, it uses special indexes (described in Section 5.2) to quickly compute elements of the view on demand. The idea of material-oriented views is developed in detail in Section 7.

Defining this view is not a straightforward matter. For one thing, the view definition depends on the workflow and its history. For another, different instances of the *same* kind of material can have *different* attributes in the view. For instance, segments of DNA may or may not have a `length` attribute, depending on whether or not they were processed by a step that measured their length. Thus, the attributes (or type) of a material depend on the *history* of the material, as well as its class. This rather dynamic feature reflects the flexibility demanded by workflow management, and is closely related to research on time and action in artificial intelligence and logic programming [31, 30, 36, 44]. These issues are addressed in detail in Section 7, where we introduce structures that allow the view to be defined *independently* of the workflow, so that the view definition does not have to be changed each time the workflow changes.

In addition to views, the database must support historical queries. Most of these queries can be divided into four categories:

- Queries that look-up a particular experimental step. For instance, "Retrieve the most recent hybridization step involving clones `C123` and `TC456`". Like material-oriented views, these queries are used in the day-to-day operation of the lab and are very common.

- Queries that examine the workflow history of a particular material. For instance, "Retrieve all steps involving clone `C123`". These queries are used to explore anomalous experimental results, often via an interactive browser.

- Data-dredging queries. These queries are aimed at identifying bottlenecks in a workflow, and typically involve aggregation. For instance, "How often was step 1 repeated after step 3".

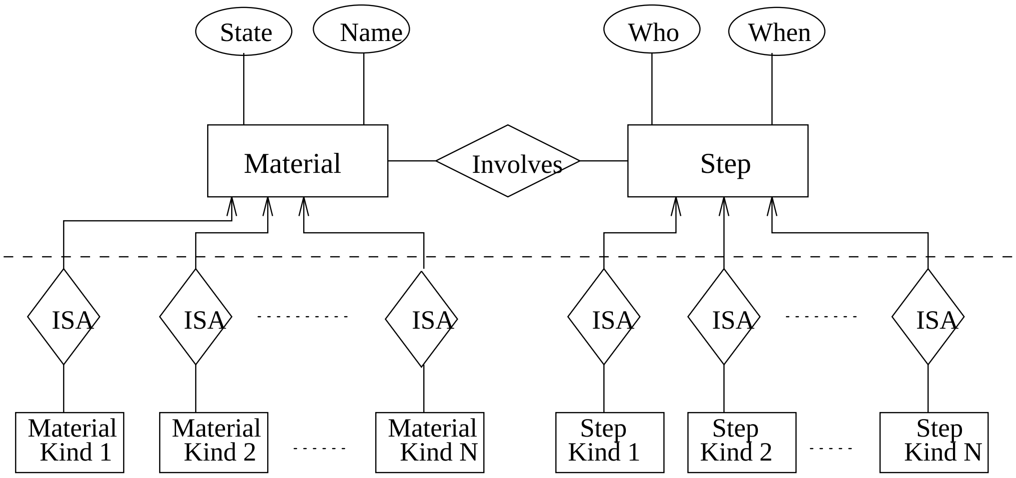

Figure 1: An EER diagram for the benchmark schema

- Report-generation queries. These queries scour the event history to produce summaries of laboratory activity. The summaries are hierarchically structured, so the query answer is typically a nested set or a nested list (*i.e.*, a list of lists of lists).

Detailed examples of queries and updates are given in Section 8.

The functions described above come largely under the heading of workflow *tracking*. As described in Section 2.2, a workflow management system must also provide a means of workflow *modeling* [22]. A workflow model specifies the dependencies among workflow steps. The Genome Center represents the most important dependencies as a *workflow graph* [53]. Workflow graphs are based on the idea that each material has a workflow *state*, and as the material is processed, it moves from one state to another. The workflow graph largely determines the workload for the DBMS. Appendix B gives an example of a workflow graph, one that forms the basis of the workload for the LabFlow-1 benchmark.

4 Benchmark Database Schema

Like the OO1 benchmark [13] and the Sequoia 2000 benchmark [54], our benchmark is independent of the data model provided by the DBMS. We therefore describe the database schema abstractly, using an extended entity relationship (EER) diagram [57, 34]. An EER diagram is an ER diagram extended with `is-a` links. The database itself can be implemented in any number of ways, *e.g.*, as a relational database or as an object-oriented database.

As shown in Figure 1, the EER diagram has two levels (separated by a dashed line). The top level is abstract, and is a partial specification of an event-history model.[5] The second level is concrete, and represents the workflow in particular laboratory projects. The top-level is the same for all laboratory projects, while the bottom level changes from project to project. In fact, because of schema evolution, the bottom level also changes during the lifetime of a single project, possibly many times.

In the laboratory, materials are processed by experimental steps. The top level of the EER diagram thus has two entities: `material` and `step`, as shown in the figure. `material` has two attributes: `name`, a human-recognizable name; and `state`, a workflow state.[6] `Step` has two attributes: `who`, the name

[5] The model is more completely specified in Sections 6 and 7.

[6] Associating each material with a single state is an oversimplification, though it is the technique actually used at the Genome Center. For purposes of quality control, one would sometimes like to concurrently execute the same workflow

of the user (or program) that carried out the step; and `when`, the time and date at which the step was completed. Each laboratory step involves at least one material, and each material is processed by at least one step. This is represented by the binary relation `involves` in the top level of the EER diagram. `Involves` is a many-to-many relation between steps and materials. Note that `step` data is historical, while `material` data is not.

Most queries in our benchmark require navigating between steps and materials via the `involves` relationship. The frequency of such queries arises because the database supports two mutually-dependent views of the data—a step-centered view, and a material-centered view—as described in Sections 3 and 7. The `involves` relation allows data to be conveniently translated from one view to the other.

Different laboratory projects will involve different kinds of materials and different kinds of experimental steps. These appear as entities in the lower level of the ERR diagram. The two levels of the diagram are connected by `is-a` links. Each kind of material is connected to `material` by an `is-a` link, and each kind of step is connected to `step` by an `is-a` link. In object-oriented terms, each material kind is a subclass of `material` and inherits its two attributes, and each step kind is a subclass of `step` and inherits its two attributes. Each kind of material and step may also define its own attributes. In the rest of this paper, we shall adopt object-oriented terminology. We thus refer to entities as objects, or class instances, and we assume that each object has a unique identifier.

The lower level of the EER diagram can be fleshed out in many different ways depending on the particular workflows being modeled. The only requirement is that the lower level have a step kind called `create`, to represent the creation of new material instances. Here, "creation" refers not to the physical creation of a material in the laboratory (which might require an entire workflow), but the creation in the database of a data structure (a class instance) to represent the material. By having a `create` step, the audit trail (event history) contains a record of each new material instance added to the database, when it was added, and by whom. Thus, for each instance of `material`, the database also contains an instance of `create`.

The LabFlow-1 benchmark provides a specific schema for the lower level of the EER diagram. This schema, presented in Appendix A, contains three kinds of material and seven kinds of step (including the `create` step). All examples in this paper are based on this schema. The schema was designed to be similar to those used in production at the Genome Center, though it does not correspond to a laboratory protocol actually employed at the Center and is somewhat simplified.

To keep this initial benchmark simple, several complicating factors have purposely been excluded from the schema in Appendix A. For example:

- The number of material and step kinds is small. A typical laboratory workflow project might involve 40 kinds of step, and a dozen kinds of material.

- The material kinds in our schema are flat. In general, they could form a *part-of* hierarchy, since in many laboratory workflows, a material is composed of numerous other materials. For instance, an 8x12 microtitre tray consists of 96 micro test tubes.

- The step kinds in our schema are flat. In general, they could form a *class* hierarchy, since there is often a hierarchy of experimental methods. For instance, there are several ways to determine the sequence of a DNA molecule, so the `determine_sequence` class could have several subclasses.

- The schema in Appendix A is fixed and does not change as workflow progresses. In practice, workflows evolve during the course of a project; and each workflow change is manifested in the

on multiple copies of the same material. In this case, a material would have not a single workflow state, but a set of copy-number/workflow-state pairs.

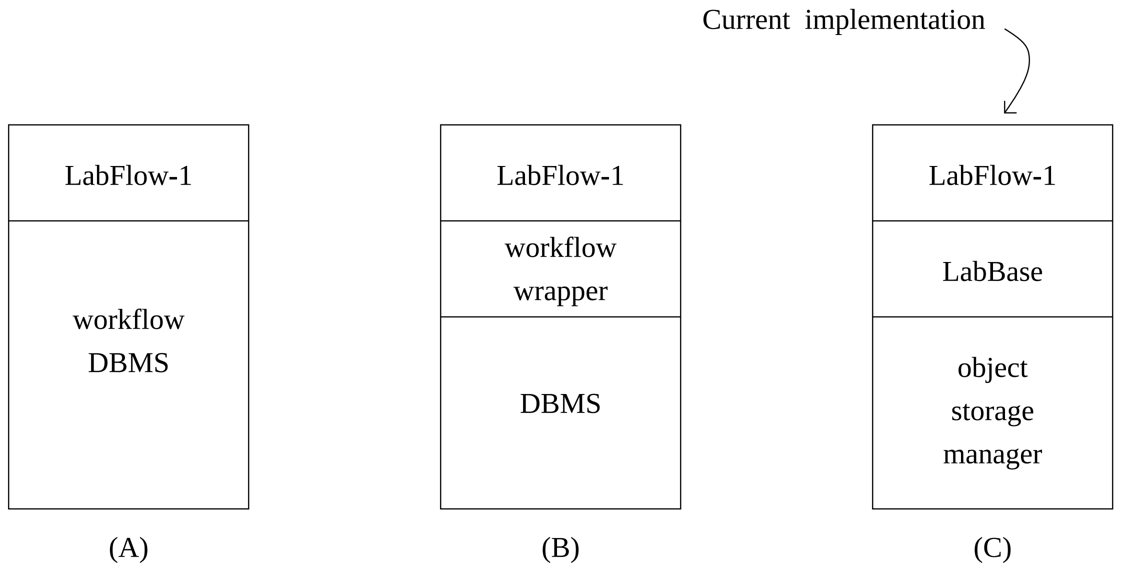

Figure 2: Three implementation architectures

(A) *LabFlow-1 can directly compare DBMSs intended for workflow tracking.* (B) *If a DBMS by itself is not suitable for workflow tracking, it may be extended by application programs ("wrappers") that implement a workflow data model and provide workflow data-management functionality. LabFlow-1 can then compare different DBMS/wrapper combinations.* (C) *In this paper, we compare different object storage managers in combination with a LabBase wrapper.*

database as a schema change.

Future versions of the benchmark will account for these features. In particular, we will introduce schema changes into the benchmark. This is unnecessary at present since our current benchmark implementation tests different storage managers with LabBase as a "wrapper." Any schema changes are thus handled by LabBase rather than by the systems under test.

An important feature of the schema in Appendix A is that steps have many more attributes than materials. In fact, some materials have no attributes other than `name` and `state`. This reflects the primary purpose of the database, which is to track workflow and maintain a history of events in the laboratory. In effect, much of the database is event-oriented (or step-oriented). As discussed in Section 3, information about materials is implicit in the history. In order to conveniently answer queries about materials, the database provides what might be called a material-oriented view, in which materials have numerous derived attributes. A material derives its attributes from the steps that have processed it.

5 Implementation

Figure 2 illustrates three different implementation architectures for systems that could be tested by the LabFlow-1 benchmark. This section briefly describes the three architectures, and then describes the architecture used in our benchmark tests in more detail. Further details can be found in [46, 47].

Architecture (A) represents the most direct test of a DBMS. Here, queries and updates from

LabFlow-1 are submitted directly to the DBMS, without any intervening software. This architecture is suitable for testing DBMSs that have been designed with workflow management in mind. Unfortunately, few (if any) DBMSs have been designed in this way, and for this reason, Architecture (B) represents the state-of-the-art implementation for systems tested by LabFlow-1. In this architecture, the DBMS is augmented with a so-called "wrapper", an application program that extends the functionality of the DBMS. Architecture (B) can be used in several ways: to compare different DBMSs with a fixed wrapper, to compare different wrappers with a fixed DBMS, or to compare different DBMS/wrapper combinations.

Architecture (C) is a special case of Architecture (B). Here, the workflow wrapper is LabBase, and the DBMS is an object storage manager. The Genome Center currently uses this architecture with ObjectStore as its storage manager [46, 47]. In this paper, we use Architecture (C) to evaluate various object storage managers. Since LabBase is implemented in persistent C++, we chose storage managers that provide persistent C++ as an interface to the application programmer. In this way, each workflow-data manager uses virtually the same LabBase implementation. Further details on the storage managers used are provided in Section 10.

LabBase extends object storage managers with three workflow-management features: a workflow query language, dynamic schema evolution, and mechanisms for rapid access to event histories. Section 6 outlines the query language. The rest of this section outlines the implementation of schema evolution and the access mechanisms. Further details can be found in [45, 46, 47, 53].

5.1 Schema Evolution and Storage Organization

One challenge for a workflow wrapper in Architecture (C) is to provide the user with schema evolution on top of a storage manager with a schema that is difficult to change.[7,8] The wrapper must therefore distinguish between the user's schema (which is dynamic) and the storage manager's schema (which may be static), and it must implement a mapping between the two. In particular, it must transform data, queries and updates from the user's schema to the storage manager's schema, and it must transform query answers in the reverse direction. Every user schema is described by the EER diagram in Figure 1. The benchmark schema in Appendix A is an example of a user schema.

The kind of schema changes that must be supported are those that result from workflow changes. Specifically, this means changes to the lower level of the EER diagram, *i.e.*, changes to the kinds of steps and materials stored in the database. In addition, as workflows change, data needs to be interpreted in the context of the workflow that created it. For this reason, a schema change does not result in a re-organization or migration of old data to the new schema (which represents a new workflow). Instead, each data object is associated forever with the class that created it, as in [52].

In our implementation, the storage manager has a fixed schema. It consists of exactly three classes, `sm_step`, `sm_material`, and `material_set`, as shown in Table 1. Each instance of `step` in the user's schema becomes an instance of `sm_step` in the storage schema. Likewise, each instance of `material` in the user's schema becomes an instance of `sm_material` in the storage schema. In addition, each workflow state is represented as an instance of `material_set`, as described later. Note that many attribute and subclass names in the user's schema are not present in the storage schema. In effect,

[7] Most persistent C++ implementations store C++ persistent class instances in nearly the same form as they would be stored in volatile memory—only pointers are modified. Thus, even adding a data member to a C++ class—the analog of adding a column to a relational schema—invalidates existing instances of the class.

[8] If the DBMS in Architecture (B) were flexible enough, one might rely on its schema-evolution capabilities to some extent. For instance, if the DBMS offered a semantic data model, one might implement each step kind in Figure 1 as a class hierarchy, whose root represents the step kind in the abstract, and whose subclasses represent different concrete versions of the step kind.

```
sm_step:
   who: string
   when: string
   step-kind: string
   attribute-names: list(string)
   attribute-values: list

sm_material:
   material-kind: string
   history: list(step)

material_set:
   state: string
   materials: set(material)
```

Table 1: Benchmark database schema as implemented in the object storage manager

Data is stored in the above schema, which remains fixed as the user's schema evolves. LabBase transforms data back-and-forth between this schema and the user's schema (Figure 1), as described in the text.

meta data in the user's schema is mapped into data values in the storage schema. The rest of this subsection describes this mapping in more detail. Steps and materials are mapped differently, so we look at each in turn.

Steps

Suppose the user creates an instance of some subclass of `step`. This object is stored as an instance of class `sm_step`. To store meta information, `sm_step` has three attributes that `step` does not: `step-kind`, `attribute-names` and `attribute-values`. `step-kind` stores the name of the object's class, as seen by the user; `attribute-names` stores a list of the object's attribute names, as seen by the user; and `attribute-values` stores a list of the object's attribute values, as seen by the user.[9]

For example, consider the following step instance from the user schema in Appendix A:

```
step123: instance-of t_position
   who: 'Lincoln'
   when: '23/jan/1995/14:36:57'
   material: material456
   left_length: 1500
   right_length: 1750
   mapping_successful: true
```

This is the object as the user sees it. It is an instance of class `t_position`, which is a subclass of `step`, where `step123` is its oid (object identifier). The attributes `who`, `when` and `material` record who carried out the step, when the step was carried out, and what material the step was applied to, respectively. The value of the latter attribute is `material456`, the oid of some material instance. The

[9] All lists are implemented as arrays or dynamic arrays in persistent C++.

attributes `left_length`, `right_length` and `mapping_successful` record the results of the step, *i.e.*, the information gleaned by a workflow activity. In the storage manager, this object is an instance of class `sm_step`, and is stored as follows:[10]

```
step123: instance-of sm_step
   who: 'Lincoln'
   when: '23/jan/1995/14:36:57'
   step-kind: 't_position'
   attribute-names: [material, left_length, right_length, mapping_successful]
   attribute-values: [material456, 1500, 1750, true]
```

With this organization, the database can contain instances of `step` of the same kind but with different attributes, *i.e.*, different types. This feature supports the most common form of workflow change: adding new attributes to a step. Creating a new step kind is also simple, and amounts to creating a new domain value for the `step-kind` attribute. Changes to the user's schema are therefore simple to implement, and do not require re-organization or migration of existing data.

Materials

Unlike steps, whose data is stored in one place, in the class `sm_step`, data about materials is split up and stored in three different places. This is because materials have three kinds of attributes: static, derived, and `state`. Static attributes have fixed values: once inserted into the database, they are never updated. Steps have static properties only, since each step instance represents an immutable record in an audit trail. All the attributes in Appendix A are static, except for `state`, which represents the workflow state of a material and is updated by each workflow activity. In addition, materials "inherit" attributes from the steps they have been involved in, as described in Sections 3 and 7. These are derived attributes, and their values can change. To provide efficient access to materials, LabBase uses a different storage organization and different access methods for each of the three kinds of attribute.

The static attributes of materials are stored using the machinery developed for steps. As described in Sections 4 and 8.3, each new material instance is added to the database by a `create` step. The static attributes of a material are stored with this step. In this way, materials can exploit the dynamic-schema-evolution capabilities of steps, described above.

The `state` attribute is implemented differently. As described in Section 2.2, `state` plays an important role in workflow control. As such, it is not part of the audit trail, and its values are destructively updated. `state` is also involved in a high proportion of benchmark queries, as shown in Tables 2 and 5. For these reasons, the `state` attribute is not stored with either steps or materials, but with a new class, called `material_set` in Table 1. Each instance of `material_set` represents a particular workflow state *and* the set of material instances currently in that state.[11] By putting a primary index on the `state` attribute of `material_set`, all the materials in a given workflow state can quickly be retrieved. This effectively provides a secondary index on attribute `state` of class `material` in the user's schema (Figure 1).

Finally, to efficiently implement derived attributes, several indexes and access methods are needed. These are described in detail later. For now, we need only mention that one of the indexes is historical and uses the `history` attribute of class `sm_material`. This attribute stores a list of all the steps that a material has been involved in, ordered chronologically.

[10]To save storage space, each attribute in the user's schema is assigned a numeric code. It is this code, and not the attribute name itself, that is stored in the list `attribute-names`. However, for simplicity of presentation, we shall generally ignore this detail.

[11]This is typically a small set at any given time.

To illustrate the storage of material instances, consider the following two objects from the user schema in Appendix A, a material instance, and the step instance that created it:

```
material456: instance-of clone
   name: 'C72a'
   state: 'waiting-for-activity-1'
   length:  3850

step104: instance-of create
   who: 'Steve'
   when: '22/jan/1995/11:28:32'
   material: material456
```

These are the objects as the user sees them. Here, `C72a` is a human-readable name assigned to the material instance by the laboratory, and `waiting_for_activity_1` is its workflow state. Moreover, the material instance was inserted into the database by `Steve` on `22/jan/1995/11:28:32`. These two objects are stored in the database as follows:

```
material456: instance-of sm_material
   material-kind: 'clone'
   history: [step104]

step104: instance-of sm_step
   who: 'Steve'
   when: '22/jan/1995/11:28:32'
   step-kind: create
   attribute-names: [material, name, length]
   attribute-values: [material456, 'C72a', 3850]

set12: instance-of material_set
   state: 'waiting-for-activity-1'
   materials: {material456, material394, material517, material486}
```

Notice that the lists in `step104` contain attributes from classes `create` and `clone` in the user's schema, *i.e.*, they contain both step and material attributes. Only `create` steps have this property. Finally, the object `set12` represents the set of material instances in the workflow state `waiting-for-activity-1`. These instances include `material456` as well as `material394`, `material517` and `material486`.

5.2 Historical Indexes and Access Methods

As discussed in Sections 3 and 4, the DBMS maintains an event history (or audit trail) of all workflow activity. Like all data, this history can be queried, and views can be defined on top of it. The most common queries require navigating between steps and materials via the `involves` relationship (see Figure 1). The frequency of such queries arises because the database supports two mutually-dependent views of the data—a step-centered view, and a material-centered view. The `involves` relation allows data to be conveniently translated from one view to the other. These translations involve two types of query: (*i*) given a step id, find all materials involved in the step; and (*ii*) given a material id, find steps involving the material, especially the *most-recent* step satisfying some condition. (Sections 6 and 7 elaborate on the idea of most-recent steps.) To efficiently evaluate such queries, LabBase provides special-purpose access structures and access methods.

There are many dependencies and integrity constraints between the `involves` relation and the `step` and `material` classes (Figure 1). For example, in Section 5.1, `material456` is an attribute value of `step123` in the user's schema. `material456` is therefore involved in `step123`, so the `involves` relation must contain the tuple `(step123, material456)`. Because of such constraints, adding a new step or material instance to the database can require adding several tuples to the `involves` relation, to maintain consistency. This maintenance can be a substantial cost, since adding new instances to the database is a large portion of the data accesses in the LabFlow-1 benchmark, as shown in Table 2. To minimize this cost, LabBase does not materialize the `involves` relation. Instead, it rapidly computes tuples in the relation on demand, using the special-purpose mechanisms described next. Thus, like all other elements of the user's schema, the `involves` relation is a view.

Of the two types of queries described above, queries of type (*i*) are the most straightforward. They require no special access structures, since steps contain pointers to all the materials they involve. These pointers are the values of various step attributes. Given a step id, LabBase retrieves the step and examines each of its attributes. It returns the values of those attributes whose data type is `material` or some subclass thereof.

Queries of type (*ii*) are more subtle. To support them, LabBase provides a special access structure. This structure is the `history` attribute of class `sm_material`, shown in Table 1. This attribute stores a list of all the steps that a material has been involved in, ordered chronologically. For example, suppose that `material456` represents a clone that has been involved in step instances `step104`, `step123`, `step128`, `step135` and `step142`, in that order. This information is stored as follows:

```
material456: instance-of sm_material
   material-kind: ‘clone’
   history: [step104, step142, step135, step128, step123]
```

Since we are usually interested in most-recent steps, the steps on the history list are ordered chronologically, *i.e.*, sorted by the value of the `when` field of each step. The history list grows as `material456` progresses through its workflows and becomes involved in more-and-more steps.

The history list of a material must be consistent with the `involves` relation. Thus, in the above example, `material456` must be involved in step instances `step104`, `step123`, `step128`, `step135` and `step142`, and no others. `material456` should therefore be the value of some attribute in each of these steps, and no others. To maintain consistency, LabBase updates the history lists whenever it creates new step instances.

Attribute Indexes

As described in Sections 3 and 7, a material effectively "inherits" the attributes of all the steps it has been involved in. For instance, if a step measures the weight of a material, then the step may have a `weight` attribute, which is "inherited" by the material. It is common to retrieve materials based on the values of their inherited attributes. (*e.g.*, retrieve all materials of a given weight.) LabBase provides indexes to efficiently carry out such associative retrievals.

In principle, these queries could be evaluated as follows: given an attribute name and value, retrieve all steps that have this attribute and value, and then retrieve the materials involved in these steps. For some attributes, these queries are very common; so for efficiency, LabBase provides a special index on each such attribute.[12] These indexes retrieve the materials directly, without accessing the intervening steps; and they are updated each time a step is added to the database. In addition, there is always

[12]The current implementation of LabBase supports only unique indexes, though extending attribute-indexing with non-unique indexes is simple and planned for a future release.

a unique index on `name`, which permits rapid retrieval of a material given its name. In the LabBase implementations discussed in this paper, these indexes are either supplied as an option within the storage manager's collection class library or supplied as a separate B-tree class.

6 Query Language

This section briefly describes a query/update language. The language provides a further specification of our event-history model, and is used in Section 8 to give examples of workflow queries. It is a deductive language in the tradition of Datalog and Prolog, and is very similar to the query language used at the Genome Center. The reasons the Genome Center uses a deductive query language have been described in detail elsewhere [24, 47, 46]. Broadly speaking, these reasons are expressiveness, simplicity and flexibility. For instance,

- complex structures (such as lists of lists) are easily modeled as function terms;
- complex and domain-specific operations can be provided as built-in predicates (like arithmetic in Prolog);
- deductive languages are easy to parse;
- they provide recursion;[13]
- they are not tied to a particular DBMS;
- they can be embedded in any programming language.

In addition, there is a well-developed literature and readily-available expertise on deductive-database and logic-programming technology.

6.1 Primitive Query Predicates

In this paper, queries are defined in terms of a small set of primitive query predicates. These predicates can be combined by logical operators to define complex queries, as in Datalog and Prolog. The rest of this section introduces these predicates and shows how to express complex queries with them. The discussion provides a more detailed specification of the data model for the LabFlow-1 benchmark. In this discussion, *class* is a class name, *attr* is an attribute name, `val` is an attribute value, and `obj` is an object identifier. For brevity, we shall often say "object" or "instance" instead of "object identifier".

For every class and every attribute, we define the following predicates:

- *attr*`(obj,val)` is true if and only if *attr* is an attribute of object `obj` and `val` is its attribute value.
- *class*`(obj)` is true if and only if object `obj` is an instance of *class*.
- `involves(s,m)` is true if and only if `s` is a step instance, `m` is a material instance, and step `s` involves material `m`.

To make things more concrete, we give examples of these predicates based on the schema of Appendix A. We suppose that all of the following atomic formulas are stored in the database, where `m` is the object id of a material, and `s` is the object id of a step:

[13] Although the Genome Center would like to have recursion, it is not currently supported by LabBase, nor is it required for the LabFlow-1 benchmark.

- `clone(m)` means that `m` is an instance of the class `clone`.
- `name(m,'C123')` means that the name of `m` is `C123`.
- `length(m,4000)` means that the length of `m` is 4000 base pairs.
- `assemble_sequence(s)` means that `s` is an instance of the class `assemble_sequence`.
- `when(s,'18/jan/1995/10:45:33')` means that step `s` was completed on 18 January 1995 at 10:45:33 am (*i.e.*, at 10:45 and 33 seconds).
- `who(s,'Steve')` means that step `s` was carried out by Steve.
- `material(s,m)` means that material `m` is the principal material processed by step `s`. For example, if `s` is an `assemble_sequence` step, then `m` is the clone whose sequence is being assembled.[14]
- `clone_sequence(s,'ATCGATCGATCGATCGATCG')` means that step `s` assembled the sequence `ATCGATCGATCGATCGATCG`. *i.e.*, This is the DNA sequence of clone `m` according to step `s`.[15]

Using binary and unary predicates in this way is a standard encoding of objects in logic. One of its main advantages is that adding new attributes to a class requires only the addition of new predicates to the database; so, existing predicates do not have to be changed. Since attribute additions are the most frequent kind of schema change, this approach helps minimize the sensitivity of the information management system to schema changes.

With these predicates, it is already possible to express some simple historical queries. In the examples below, and throughout this paper, we use the Prolog convention that variables begin with upper case letters, while constant symbols begin with lower case letters. Thus, in the next example, `S`, `W`, `M` and `N` are variables, while in the items above, `s` and `m` are constants.

Example 6.1 (Select-Project-Join) Retrieve the names of all clones whose sequence was assembled by Steve on or after 17 January 1995. Using the predicates described above, this query is expressed by the atomic formula `answer(N)` defined by the following Datalog rule:

```
answer(N) <- assemble_sequence(S),
             when(S,W),  W > '17/jan/1995/00:00:00',
             who(S,'Steve'),
             material(S,M),
             clone(M),
             name(M,N).
```

□

6.2 Most-Recent Steps

To ask more sophisticated historical queries, we need to introduce one more primitive query predicate. This predicate, which pervades much of the LabFlow-1 benchmark, allows us to choose the most-recent step from a set of steps. The reason for its importance is that the Genome Center is almost always interested in the most-recent (*i.e.*, most up-to-date) experimental results. If a query does not specify the time and date of a step, then the most-recent step is implied by default.

[14] Thus, `material(s,m)` always implies `involves(s,m)`, but not vice versa.

[15] In practice, the sequence of a clone would be several thousand characters long.

To make this idea more precise, recall that every step has a `when` attribute, which specifies the time and date of the step. Given a set of steps, the most-recent step is the one whose `when` attribute has the largest value. Note that in principal, the most-recent step may not be unique, since several steps may have the largest `when` value. (This situation does not seem to arise in practice however.)

To incorporate this notion into our query language, let ψ be a query that specifies a set of steps. Thus, $\psi(s)$ is true only if s is a step instance, and $\{S \mid \psi(S)\}$ is the set of steps specified by ψ. We define the following predicate:[16]

- `most-recent(step)`:$\{$`S | `ψ`(S)`$\}$ is true if and only if `step` is the most-recent step in the set $\{S \mid \psi(S)\}$.

For the purpose of this paper, ψ is a select-project-join query. In deductive-database terms, ψ is a conjunctive existential query. We use the Prolog convention for queries, in which a list of atomic formulas represents a conjunction, and free variables are existentially quantified.

Example 6.2 (Most-Recent Step) Retrieve the most-recent step involving the material named `C123`. This query is expressed by the following formula:

```
most-recent(Step):{S | involves(S,M), name(M,'C123').}
```

It is also expressed by the simpler formula

```
most-recent(Step):{S | steps(S,'C123')}
```

where the predicate `steps` is defined by the following Datalog rule:

```
steps(S,N) <- involves(S,M), name(M,N).
```

Intuitively, `steps(S,N)` means that `S` is a step involving a material named `N`. Given a value for `N`, this formula will retrieve a set of steps. □

Example 6.3 (Most-Recent Attribute Value) In the schema of Appendix A, `clone` is a material class, and `assemble_sequence` is a step class, with attribute `clone_sequence`, among others. We want to retrieve the most-recently assembled sequence for the clone named `C123`. This query is expressed by the atomic formula `answer('C123',Seq)`, where the predicate `answer` is defined by the following Datalog rules:

```
answer(CName,Seq) <- name(Clone,CName),
                     clone(Clone),
                     most-recent(Step):{S | steps(S,Clone)},
                     clone_sequence(Step,Seq).

       steps(S,C) <- assemble_sequence(S), material(S,C).
```

In the first rule, the variable `CName` represents the name of a clone, `Clone` represents the clone object itself, and `Seq` represents its DNA sequence. The second rule defines the predicate `steps(S,C)`, which means that a DNA sequence for clone `C` was assembled by step `S`. The first rule uses this predicate to retrieve every step, `S`, that has assembled a sequence for clone `C123`. The most recent of these steps is then bound to the variable `Step`. □

[16]This predicate can be expressed in SQL using a rather bulky expression in which ψ is specified twice: once in a subquery, to extract the maximum value of `when`; and once in the top-level query, to extract the most-recent step.

It is worth mentioning that this machinery provides the flexibility to ask what might be called "time travel" queries. In effect, we can go back to a given date, and ask "most recent" queries as of that date. For instance, the following query retrieves the step, S, satisfying ψ that was most recent as of a given `date`:

$$\texttt{most-recent(Step):}\{\texttt{S | }\psi\texttt{(S), when(S,Time), Time < date}\}$$

Finally, we would like to emphasize that the query language described in this section does not imply a particular storage organization for the database. However, whatever storage organization and storage manager are used, the primitive query predicates described above must be provided as a view. This view need not be implemented in SQL. For instance, in the operational version of LabBase, these view predicates are implemented in persistent C++ on top of an ObjectStore database with a different schema and special access structures (see Section 5.1). Similarly, if Exodus [10] were the storage manager, these predicates would be implemented in E, the Exodus programming language. In this way, storage management, access structures, and query processing are separated. Many different storage organizations are possible, as are many different query processing strategies. Testing different organizations and different storage managers is one purpose of the LabFlow-1 benchmark. Testing different query processors and different logic-programming systems is another.

6.3 Update Primitives

In addition to the above predicates, the query language includes Prolog-like operations for inserting and deleting atomic formulas (tuples) from the database, and a predicate for creating new instances of a class. Section 8 uses the predicates below to specify updates in the LabFlow-1 benchmark.

- `assert(p)` inserts the atomic formula `p` into the database. This predicate is always true and has the standard Prolog semantics [16].
- `retract(p)` deletes the atomic formula `p` from the database. This predicate is true if `p` was in the database prior to deletion. It has the standard Prolog semantics [16, 8].
- `create:`*class*`(obj)` creates a new object identifier, `obj`, for *class*, and inserts the formula *class*`(obj)` into the database. This predicate is true as long as *class* is the name of a class in the database schema.

7 Material-Oriented Views

The Genome Center's database acts as an on-line laboratory notebook. Information about materials is gleaned through a series of experimental steps, and each step is recorded in the database. Such a notebook is a central part of any scientific enterprise. However, it means that information about each material is not recorded in one place, but is spread over many steps. For example, information about the DNA sequence of a genetic marker is associated with a *sequencing step*, while information about the position of the marker in a genetic map is associated with a *mapping step*. In addition, some steps may have to be repeated several times, until good results are achieved. Information about a particular property of a particular material may thus be spread over several versions of the same step. This is a common occurrence in laboratory notebooks.

From these scattered pieces of information, we must produce a view of each material. The view should include the most-recent (*i.e.*, most up-to-date) experimental results. To create this view, we define a number of computed attributes. These attributes take information from a step, and associate it with a material. For each attribute *attr* of a step, we define an attribute `mr:`*attr* of materials. Recall that *attr*`(s,v)` means that `v` is the value of attribute *attr* in step `s`. The formula `mr:`*attr*`(m,v)` means that `v` is the *most-recent* value of attribute *attr* of material `m`, as determined by some step. Formally, `mr:`*attr* is defined by the following rule:[17]

```
mr:attr(M,V) <- material(M),
                most-recent(Step):{S | involves(S,M), attr(S,_).},
                attr(Step,V).
```

This rule finds the most-recent step involving material `M` that has a value for attribute `attr`. This value, `V`, is then returned.

These rules define a material-oriented view of an event-oriented database. This view simplifies the formulation of queries about materials. A programmer can now ask for the properties of a material without knowing how the property values were determined, *i.e.*, without having a detailed knowledge of laboratory workflow. He does not have to know the name of the relevant laboratory steps, nor the order in which they were carried out, nor the history of workflow changes. Likewise, application programs that query materials will not depend on the workflow, and will not have to be changed whenever the workflow changes.

Another important property of this view is that its *definition* is independent of the workflow. Thus, the view definition does not have to be changed each time the workflow changes; and we do not need a separate view definition for each workflow. This one definition provides a view of all materials ever processed in the lab, regardless of what workflow they were processed under. This generality is made possible by the predicate `involves(s,m)`, which provides a general link between steps and materials. Maintaining the extent of this predicate in the face of schema and workflow changes is relatively simple and is discussed in Section 8.3.

Example 7.1 (Querying the View) In the schema of Appendix A, `clone` and `tclone` are material classes. For each clone, retrieve its name and the most-recent values of the attributes `clone_sequence` and `child_tclone`. For the tclone child, retrieve its name and the most-recent values of the attributes `l_sequence`, `r_sequence`, `left_length` and `right_length`. Return all attribute values except the object id of the tclone. This query is expressed by the following rule:

```
answer(CName,CSeq,TCName,LSeq,RSeq,LLen,RLen)
                             <- clone(Clone),
                                name(Clone,CName),
                                mr:clone_sequence(Clone,CSeq),
                                mr:child_tclone(Clone,TClone),
                                name(TClone,TCName),
                                mr:l_sequence(TClone,LSeq),
                                mr:r_sequence(TClone,RSeq),
                                mr:left_length(TClone,LLen),
                                mr:right_length(TClone,RLen).
```

□

Sometimes, a *set* of materials has properties that derive from interactions between the individual materials. For instance, two pieces of DNA may or may not stick together. In this case, "sticking together" is a property of a pair of materials. Such interactions are central to genome mapping and sequencing, and to laboratory experiments in general. Like the properties of single materials, the

[17] In this rule, and throughout this paper, we use the Prolog convention that $p(a, \_)$ means $\exists X p(a, X)$.

properties of material sets are usually determined by experiment. Repeated experiments on the same set can yield different results; and the most-recent results are of greatest interest.

It is not hard to extend the material-oriented view defined above from single materials to small sets of materials. For each attribute, *attr*, and each positive integer, k, we introduce a predicate, `mr:`*attr*`(M1,...,Mk,V)`, of arity $k+1$. Intuitively, this predicate means that `V` is the most-recent value of attribute *attr* of the material set {`M1,...,Mk`}, as measured by some laboratory step. Formally, this predicate is defined by the following rule:

```
mr:attr(M1,M2,...,Mk,V) <- material(M),
                           most-recent(Step):{S | involves(S,M1),
                                                  involves(S,M2),
                                                    ...
                                                  involves(S,Mk),
                                                  attr(S,_).},
                           attr(Step,V).
```

This rule finds the most-recent step involving materials `M1`, `M2`,...,`Mk` that has a value for attribute *attr*. This value, `V`, is then returned.

The database schema in Appendix A assigns very few attributes to materials. However, the material-oriented view defined in this section provides materials with a great many *derived* attributes. Every attribute, *attr*, of every step, becomes a derived attribute, `mr:`*attr*, of some materials. Each time a material is processed by a workflow step, it acquires the attributes of that step. The derived attributes of a material thus depend on the workflow steps that have processed it. Consequently, different instances of the *same* material class can have *different* derived attributes. This is possible for two reasons: (*i*) different materials may take different paths through the same workflow graph, and may thus be processed by different steps; and (*ii*) the workflow graph is continually changing, so materials processed at different times may be processed by different steps. In both cases, different materials from the same class may acquire different derived attributes. It is therefore impossible to assign a fixed schema to all instances of a given material class. The schema of a material instance depends not only on its class, but also on its *history*.

It is worth noting that deriving most-recent values for materials is similar to certain problems of time and action in logic programming and artificial intelligence. For instance, the event calculus [31, 30] is a logic-programming methodology developed for recording and querying event histories. It includes special access structures to quickly retrieve most-recent results. As another example, the situation calculus [36, 44] encodes event histories as function terms. Queries in the situation calculus ask about the present database state, which is defined to be the result of the most-recent actions.

For our purposes, the material-oriented views defined in this section can be expressed in any of these formalisms, or in SQL, or they can be implemented as special built-in predicates using C or some other programming language. It is also acceptable for these views to be materialized and stored in database relations. In the latter case, the views can be maintained incrementally, or transactions that add steps to the database can update the view relations directly. This requires some care, however, since steps can be entered into the database in any order, and there is no guarantee that a step being entered is the most recent. To use the terminology of temporal databases, "most recent" is based on valid time, not transaction time [56].

Comparing the performance of these alternative implementations of material-oriented views would be an interesting application of the LabFlow-1 benchmark.

8 Typical Queries and Updates

This section gives examples of queries and updates that are typical of those at the Genome Center, and of those in the LabFlow-1 benchmark. They are specified in terms of the deductive language introduced in Section 6 and the views developed in Section 7. All examples are based on the schema in Appendix A.

8.1 Querying the Event History

Because the database is historical, schema changes present a challenge to query formulation and query processing. Historical queries must do the right thing even though they may span many schema changes. For instance, the laboratory may need to produce a report of all clones sequenced during some period of time, even though the sequencing method—and the schema of its data—changed several times during this period.

There are several solutions to this problem depending on the kind of schema change and the nature of the query. One solution is to define a general view of the data that is oriented around materials instead of steps, as described in Section 7. This solution is appropriate for queries about materials. For queries about steps, other solutions are needed. The problem is that usually we want to query not just a particular kind of step, but its numerous variations as well. The two most common ways to vary a step are to add (or delete) attributes to it, or to split a complex step into a combination of simpler steps. This section examines both cases.

Attribute Changes:

It is very common for instances of the same step class to have different attributes, depending on when the instances were created. In effect, instances with different attributes belong to different versions of the same class. Usually, biologists are not interested in what particular version an instance belongs to. Instead, they are interested in particular attributes. For example, a biologist may be interested in steps that produced a DNA sequence for a clone. Different versions of the step may have measured various parameters in addition to the sequence, but that is irrelevant. The result is that many queries ask about the existence of attribute values.

Example 8.1 (Event Look-Up) In the schema of Appendix A, `determine_sequence` is a step class, and `tclone` is a material class. The former class has attributes `material`, `l_sequence`, `r_sequence` and `sequencing_successful`, among others. For each tclone, find its most-recent unsuccessful sequencing step that recorded values for `l_sequence` and `r_sequence`. Return the two sequences and the name of the tclone. This query is expressed by the atomic formula `answer(TCName,LSeq,RSeq)`, where `TCName` is the name of a tclone, and `LSeq` and `RSeq` are the two sequences. The predicate `answer` is defined by the following rules:

```
answer(TCName,LSeq,RSeq) <- tclone(TClone),
                            name(TClone,TCName),
                            most_recent(Step):{S | steps(S,TClone)},
                            l_sequence(Step,LSeq),
                            r_sequence(Step,RSeq).

steps(S,TClone) <- determine_sequence(S),
                   material(S,TClone),
                   sequencing_successful(S,false),
```

```
l_sequence(S,_),
r_sequence(S,_).
```

□

The query in Example 8.1 does *not* include any knowledge of how the class `determine_sequence` has evolved; nor does it depend on an explicit versioning mechanism. Any version of `determine_sequence` is of interest as long as it involves the given tclone, has attributes `l_sequence`, `r_sequence`, and `sequencing_successful`, and the value of the latter attribute is `false`.

Class Splitting:

The vast majority of workflow changes add a new attribute to a step. A less frequent, but more significant change is where one step is split into a sequence of two steps, which accomplish the same end, but by different means. This kind of workflow change represents a major commitment by the laboratory; and for some queries, the application programmer must be aware of them. In terms of the database schema, one step class is split into two new step classes. Of course, we must retain the old unsplit class in order to make sense of data gathered in the past. The result is that two new classes are introduced into the database schema.

As an example, notice that the step `determine_sequence` in Appendix A has two kinds of attributes, those for the left side of a tclone, and those for the right side. This step might at some time be replaced by a sequence of two steps. Corresponding to these two new steps would be two new classes, with the following schema:

```
determine_left_sequence: ISA step
    material: tclone
    l_sequence: dna_sequence
    l_start: integer
    l_length: integer
    l_sequencing_successful: boolean

determine_right_sequence: ISA step
    material: tclone
    r_sequence: dna_sequence
    r_start: integer
    r_length: integer
    r_sequencing_successful: boolean
```

These two classes are added to the schema of Appendix A. They do *not* replace the class `determine_sequence`, which remains in the schema. Thus, data stored in all three classes can be queried.

Notice that the schema change did not change the attribute names. In effect, the old attributes have been distributed among the new classes. Because of this, many common queries are not sensitive to this schema change, including all queries to the material-oriented view of Section 7. For instance, the two queries `mr:l_sequence(M,Seq)` and `mr:r_sequence(M,Seq)` will continue to retrieve the most-recent values of the attributes `l_sequence` and `r_sequence`, respectively, for material `M`. Formulating these queries does not require any knowledge or awareness of this schema change.

Unfortunately, such ignorance is not always possible, especially for step-oriented queries. Example 8.2 below repeats the query of Example 8.1 in the context of the above schema change. The query is disjunctive, with one disjunct (*i.e.* one set of rules) for each schema. In one case, the requested information is associated with a single step, while in the other case, it is distributed among two steps.

Example 8.2 (Disjunctive Queries) Repeat the query of Example 8.1 assuming that tclones are sequenced either by a single step, `determine_sequence`, or by two steps, `determine_left_sequence` and `determine_right_sequence`, as described above. In all cases, return the name of the tclone, its left sequence, and its right sequence. As before, consider only unsuccessful sequencing steps for which the sequence attributes exist. This is accomplished by the following rules:

```
answer(TCName,LSeq,RSeq,1)
                <- tclone(TClone),
                   most_recent(Step):{S | steps(S,TClone)},
                   l_sequence(Step,LSeq),
                   r_sequence(Step,RSeq),
                   name(TClone,Name).

answer(TCName,LSeq,RSeq,2)
                <- tclone(TClone),
                   name(TClone,Name),
                   most_recent(LStep):{S | stepsL(S,TClone)},
                   l_sequence(LStep,LSeq),
                   most_recent(RStep):{S | stepsR(S,TClone)},
                   r_sequence(RStep,RSeq).

steps(S,TClone) <- determine_sequence(S),
                   material(S,TClone),
                   sequencing_successful(S,false),
                   l_sequence(S,_),
                   r_sequence(S,_).

stepsL(S,TClone) <- determine_left_sequence(S),
                    material(S,TClone),
                    sequencing_successful(S,false),
                    l_sequence(S,_).

stepsR(S,TClone) <- determine_right_sequence(S),
                    material(S,TClone),
                    sequencing_successful(S,false),
                    r_sequence(S,_).
```

The predicate `answer` now has a fourth argument to indicate whether the left and right sequences were determined by one step or two. □

8.2 Workflow Control

Recall from Section 2.2 that the flow of materials in a production line is guided by a workflow graph. An arc in a workflow graph represents the transition of materials from one workflow state to another. Each arc is labeled with a name and a constraint. The name identifies the transition, and the constraint delimits when the transition can occur.

Querying States:

Workflow activities frequently retrieve materials based on their workflow state. Common queries are to retrieve all the materials in a given state, and to retrieve the state of a material given its name. For instance, the query `state(M,waiting_for_sequencing)` retrieves all materials, `M`, in state `waiting_for_sequencing`. Likewise, the following rule defines a predicate that retrieves the state of a material given its name:

```
answer(State,MName) <- state(M,State), name(M,MName).
```

The query `answer(S,'TC123')` thus retrieves the state, `S`, of the material named `TC123`.

State Changes:

From workflow graphs, we generate a set of state-change transactions, one for each arc in each graph. These transactions are invoked by application programs, as discussed in Section 2.2. The transactions include tests and preconditions, and are easily specified in Prolog. Let `tname` be the name of a transition in a workflow graph with initial state `s1` and final state `s2`. We define a unary predicate `do:tname(M)` that changes the state of a material, `M`. The predicate first checks that the material is in state `s1` and that any constraints associated with the transition are satisfied. If so, then the predicate changes the material from state `s1` to `s2`. It is defined by the following Prolog rule:

```
do:tname(M) <- state(M,s1),
               test:tname(M),
               retract(state(M,s1)),
               assert(state(M,s2)).
```

Here, `test:tname` is a predicate representing constraints on the transition `tname`. Each transition in a workflow graph provides a (possibly empty) set of rules defining such a predicate.

Notice that the predicate `do:tname` uses the Prolog `assert` and `retract` predicates for specifying database updates, as described in Section 6.3. According to standard Prolog semantics,[18] the formula `do:tname(M)` is `true` if and only if the transaction succeeds. An application program should therefore log an error if Prolog returns `false`.

Example 8.3 (Workflow State Transitions) The following rule changes the state of a tclone after a successful sequencing step:

```
do:sequencing_ok(M) <- state(M,waiting_for_sequencing),
                       test:sequencing_ok(M),
                       retract(state(M,waiting_for_sequencing)),
                       assert(state(M,waiting_for_incorporation)).

test:sequencing_ok(M) <-
```

In this case, there are no constraints on the transition, so the predicate `test:sequencing_ok` has an empty premise and always succeeds. □

[18] A logical semantics for `assert` and `retract` can be found in [7, 6].

8.3 Workflow Tracking

The queries in this subsection insert new step and material instances into the database. The insertion of step instances into the database creates an event history (or audit trail) of workflow activity. Each insert, whether of steps or materials, causes several changes to the user's view of the data (Figure 1). We specify these changes using the update predicates of Section 6.3. If the user's view of the data is materialized, then these specifications may be executed directly. However, in the current implementation, much of the user's view is computed, not materialized, as described in Section 5. Thus each insert is transformed by LabBase into into equivalent changes to the underlying stored data (Table 1).

New Step Instances:

After a workflow step is completed, a record of the step is stored in the database. It is at this point that experimental observations are entered into the database. This is done by creating an instance of a particular step class, filling in its attribute values, and recording links between the new step and the materials it involves. One complication is that for a given kind of step, the attributes are not fixed, because of workflow and schema changes: at any given time, different versions of the same step may be applied to different materials. Such versions differ in their attribute sets (*i.e.*, in the kind of data that they gather and record). Thus, different inserts into the same class may involve different attributes. To deal with this conveniently, we introduce macros that accept a list of attribute-value pairs as argument, and expand into a sequence of database updates.

To make this idea more precise, let `step_kind` be a subclass of class `step`, and let `attr1`,...,`attrN` be a set of attributes. The following expression is a macro that creates an instance of `step_kind` and fills in attribute values:

```
insert:step_kind[attr1=Value1, attr2=Value2,...,attrN=ValueN]
```

This macro also maintains the relation `involves`, as the example below shows. Recall from Section 4 that `involves(s,m)` means that step `s` involves material `m`.

Example 8.4 (Step Creation) Clones, tclones and transposons are three different kinds of DNA sequence. A tclone is a clone that has a transposon embedded within it as a subsequence. In the workflow of Appendix B, one step involves estimating the position of a transposon within a clone. This step is an experiment carried out by a biologist. The result of the experiment is as estimate of the length of DNA sequence to the left and right of the transposon. The experiment may or may not succeed; but in either case, it is recorded in the database as a step, and in particular, as an instance of the class `t_position`, from the schema in Appendix A.

For instance, suppose that on 15 February 1995 at 22:17:55, Tony successfully completes such an experiment. The experiment shows that in the tclone named TC123, a transposon begins 976 base pairs from the right end of the tclone, and ends 1362 base pairs from the left end. Tony records this event in the database by issuing the following command (presumably as generated from some user-interface program):

```
name(TClone,'TC123'),
insert:t_position[who = 'Tony',
                  when = '15/feb/1995/22:17:55',
                  material = TClone,
                  left_length = 976,
                  right_length = 1362,
                  mapping_successful = true].
```

This expression macro-expands into the following Prolog command:

```
name(TClone,'TC123'),
create:t_position(Step),
assert(who(Step,'Tony')),
assert(when(Step,'15/feb/1995/22:17:55')),
assert(material(Step,TClone)),
assert(left_length(Step,976)),
assert(right_length(Step,1362)),
assert(mapping_successful(Step,true)),
assert(involves(Step,TClone)).
```

Notice that the last line of this command updates the `involves` relation, a general-purpose connection between steps and materials. □

In general, steps can produce data with complex values, such as arrays, tuples, lists, sets, or combinations thereof. Such values appear in the workload for our the LabFlow-1 benchmark. Steps can also involve several materials, since a step can have several attributes of type `material`. For instance, the `associate_tclone` step in Appendix A involves three materials, stored in the attributes `parent_clone`, `parent_transposon` and `child_tclone`. Each such attribute results in a tuple being inserted into the `involves` relation when an instance of the step is created.

New Material Instances:

Every day the Genome Center receives new experimental materials from other laboratories or supply companies. It also creates new materials out of existing ones. Each new material is assigned a human-readable name, and is put into the initial state of some workflow. This activity is recorded in the database in two ways: first, by creating a new instance of the step class `create`, to record the act of creation; and second, by creating a new instance of the class `material`, to represent the new material. Attribute values for the new material and step instances are then inserted into the database.

To be more precise, let `material_kind` be a subclass of the class `material`, with attributes `name`, `state`, `attr1`,..., `attrN`. We define the following predicate, where each argument represents an attribute value:

```
insert:material_kind(Name,State,Attr1,...,AttrN,Who,When)
```

which creates an instance of `material_kind` and fills in its attribute values, and creates an instance of `create` and fills in its attribute values. `Who` and `When` are values identifying the person (or machine) that created the material, and the time of creation.

As discussed in Section 4, materials are different from steps in that materials have few attributes. Moreover, they do not tend to acquire new ones as time passes. In short, material schemas do not change the way step schemas do. Thus, inserting a material instance into the database does not require the flexible macro-expansion facility described above for step instances.

Example 8.5 (Material Creation) The following Prolog rule creates a new instance of class `clone` from the schema in Appendix A:

```
insert:clone(Name,State,Length,Who,When)
                    <- create:clone(Clone),
                       assert(name(Clone,Name)),
                       assert(state(Clone,State)),
```

```
                        assert(length(Clone,Length)),
                        insert:create[who=Who,
                                      when=When,
                                      material=Clone]
```

Here, `insert:create` is a macro that builds a new instance of `create` and fills in its attribute values, as described above. □

8.4 Aggregation Queries

The queries presented here use aggregation to extract and summarize parts of the event history. There are many diverse queries of this type, and they tend to be large. The specific aggregation queries in the LabFlow-1 benchmark are described informally in Appendix B.4.

In this section, we present two relatively simple examples of aggregation queries. They are typical of those used at the Genome Center. These two queries use two built-in aggregation predicates, `count` and `setof`. The former predicate counts the number of answers to a query. In particular, `count(Query,N)` is true iff the given `Query` has `N` answers. The latter predicate creates a set consisting of all the answers to a query, after putting each answer into a user-specified form. In particular, `setof(Template,Query,Set)` retrieves all the answers to the given `Query`, plugs each answer into the given `Template`, and then collects the instantiated templates into a `Set`. This is a standard Prolog predicate, similar to the more-familiar `findall` predicate except that duplicate query answers are eliminated [8].

Counting:

In the schema in Appendix A, `clone` and `tclone` are material classes, while `associate_tclone`, `determine_sequence` and `assemble_sequence` are step classes. Here is a query involving these classes: For each clone whose sequence has been assembled, how many associated tclones had their left and right sequences determined *before* the clone sequence was assembled? Return the name of the clone and the tclone count.

```
tclone_count(CName,TCCount)
   <- clone(Clone),
      state(Clone,'Sequence Assembled'),
      most-recent(Step):{S | assemble-sequence(S), material(S,Clone).},
      when(Step,WhenAssembled),
      name(Clone,CName),
      count(tclones(TClone,Clone,WhenAssembled),TCCount).

tclones(TClone,Clone,Time)
   <- involves(Step,Clone),
      associate_tclone(Step),
      parent_clone(Step,Clone),
      child_tclone(Step,TClone),
      when(Step,When),
      When < Time,
      sequenced(TClone,Time).

sequenced(TClone,Time)
```

```
<- involves(Step,TClone),
   determine_sequence(Step),
   material(Step,TClone),
   sequencing_successful(Step,true),
   l_sequence(Step,_),
   r_sequence(Step,_),
   when(Step,When),
   When < Time.
```

The first rule selects a clone, checks that its sequence has been assembled, retrieves the most-recent sequence-assembly step for the clone, and extracts the assembly time. Finally, it retrieves the name of the clone, `CName`, and counts how many associated tclones were sequenced before the clone sequence was assembled, `TCCount`. The remaining two rules specify what is to be counted. In effect, the count is done on the result of a select-project-join query. It is important that duplicates be eliminated in the result of this query.

The second rule retrieves every tclone, `TClone`, that was sequenced before the parent clone was assembled. All tclone steps accessed by this rule must occur before a given time, *i.e.*, before the clone sequence was assembled. The rule first finds a step that associates a tclone with the clone, checks that the association was early enough, and finally checks whether the tclone was sequenced in time, as determined by the third rule. To do this, the third rule retrieves a sequencing step for the tclone, checks that it successfully produced a left and right sequence, and checks that the step was completed in time.

Set and List Generation:

As a genome lab produces DNA sequences, a common operation is to search international genome databases (such as EMBL and GenBank [9]) for similar (or *homologous*) sequences. The results of such searches are often stored locally in the laboratory's database. A widely used program for carrying our these homology searches is called *BLAST* (Basic Local Alignment Search Tool) [2]. Given a DNA sequence and a threshold, BLAST returns a set of triples of the form

$$\langle genbank\_id,\ description,\ score \rangle$$

where *genbank_id* identifies a homologous sequence, *description* is GenBank's short description of it, and *score* is an estimate of the probability that the similarity to the given DNA sequence occurred by chance (so a low score indicates a close match). Only triples with *score* less than or equal to the given threshold are returned.

Many workflows at the Genome Center and other genome laboratories include steps that carry out BLAST searches. For example, we could augment the workflow in Appendix B with a step called `blast_step` that carries out similarity searches on the sequences read from tclones. This step would naturally have the following scheme:

```
blast_step: ISA step
    material: tclone
    homologies: setof(tuple(string,string,number))
```

Here, the `homologies` attribute has a complex data type because it stores the result of a BLAST search.

A biologically interesting query would be to return the set of GenBank id's and descriptions for all homologies in the database with a score less than some specified threshold. The following rules answer this query using standard Prolog notation in which square brackets represent lists.

```
genbank_ids(Threshold,Set) <- setof([Id,Descr],significant(Id,Descr),Set)

significant(Id,Descr) <- blast_step(Step),
                         homologies(Step,BlastSet),
                         element([Id,Desc,Score],BlastSet),
                         Score < Threshold.

element(E,[X|Set]) <- element(E,Set)
element(E,[E|Set]).
```

Here, the second rule retrieves the id and description of all significant homologies in the database. For each blast step, this rule retrieves the set of homologies (`BlastSet`), from which it examines each triple, and returns the id and description of those triples whose score is below threshold. The first rule then uses the Prolog predicate `setof` to gather the individual answers into a set, forming a set of pairs of the form `[Id,Descr]`. Thus, given a specific threshold, the first rule returns a set of pairs.

Notice that the second rule uses the predicate `element` to retrieve each triple in `BlastSet`. Intuitively, `element(E,S)` is true iff `E` is an element of set `S`. This predicate can be defined by a recursive rule, as above, or it can be a built-in predicate.

9 Benchmark Database and Workload

The database for our benchmark maintains a record of laboratory activity. It records what was done, when it was done, who did it, and what the results were. The database is thus an event history or audit trail. We therefore need to provide a simple yet realistic sequence of events, both to build the database and to serve as a workload for our benchmark.

To some extent this issue is addressed by the TPC benchmarks that simulate banking debit/credit transactions [49]. In our terminology, these benchmarks have one kind of material (bank accounts), and one kind of event (change account balance). They also have one kind of query: look up an account record given its key, and return its current balance. The database itself is relational, with simple data types like integer and string, and it stores an event history, though this history is never queried. The workloads for these TPC benchmarks consist of a *random* sequence of events.

This is very different from workflow. Workflow involves different kinds of materials and events; it involves complex data types and queries; and the queries may examine the entire event history. Moreover, workflow events are not random, but are highly correlated: the choice of what to do next to a material depends on what has already been done to it.[19] Thus, the issue of generating a realistic workload cannot be solved by generating a random sequence of simple events, as in TPC benchmarks. This would not capture the idea of materials flowing through a network of workflow stations. Simply put, the TPC banking benchmarks deal with large numbers of simple, random events, while workflow deals with a smaller number of complex, correlated events.

One way to develop a workflow benchmark would be to use a real database and a real workload from the Genome Center, generated during its day-to-day operations. We decided against this option because the resulting benchmark would be complex, hard to understand, and difficult to scale. Instead, our benchmark is based on a relatively simple workflow simulation. This simulation, described in Appendix B, models many of the important features encountered in real workflows at the Genome Center. For instance, the workflow has cycles, it has success and failure states, and it models multiple,

[19] Workflow events are not deterministic however, since the exact sequence of workflow steps that a material undergoes—and their outcomes—cannot be predicted in advance.

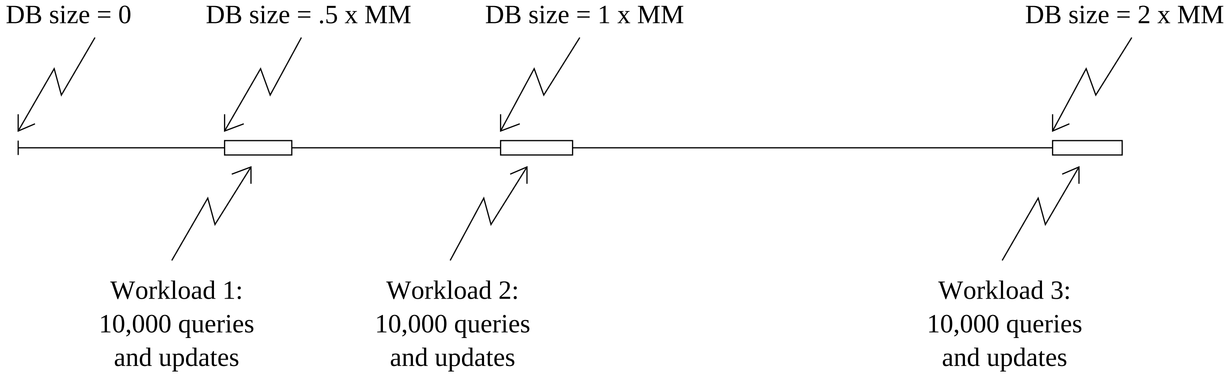

Figure 3: The Derivation of Three Workloads from the Database Log

co-operating production lines. As materials are processed by the workflow, each workflow activity produces a stream of database accesses, consisting largely of queries and inserts. These accesses are recorded as they arrive at the database, producing a *log* of database activity. Our benchmark is based on this log.

Starting from an empty database, the inserts in the log produce a constantly expanding database. Our benchmark monitors the performance of the DBMS as the database expands. Specifically, we perform measurements at three different database sizes: when the database is half the size of physical main memory, the same size as main memory, and twice the size. By defining measurement points in terms of memory size, the benchmark can be scaled up as memories grow. At each database size, we identify an interval of 10,000 queries and updates from the database log. We augment each interval with a small number of report-generation queries, to simulate the contribution to the DBMS workload by administrative departments (as described in Appendix B.4). The three augmented intervals then become three workloads, one for each database size. The three workloads are similar but not identical, since the workflow simulation will not generate exactly the same sequence of 10,000 queries and updates more than once. The derivation of the three workloads from the database log is illustrated in Figure 3.

The rest of this section describes the construction of the workloads in more detail, as well as the benchmark measurements. The measurements record elapsed time, user cpu time, system cpu time, database size, and number of major page faults. All times are in seconds. Of these measurements, users will be most interested in elapsed time and database size. The other measurements are intended for database developers and debuggers.

9.1 Generating The Workload

The workload for our benchmark is generated by the workflow simulation described in Appendix B. The simulation maintains a logical log of all database updates and queries. We ran the simulation against a version of LabBase that relies on ObjectStore for storage management.[20] We allowed the simulation to continue until the file size of the database was slightly larger than 64Mbyte. We refer to the log resulting from this simulation run as the *initial log*.

We then modified the initial log as follows:

- Based on periodic measurements of the size of the database file during the simulation run, we selected points within the log that correspond to database-file sizes of approximately 16Mbyte,

[20] Section 10 describes in detail the versions of LabBase that we tested using the benchmark.

32Mbyte, and 64Mbyte. Since the benchmark machine has 32Mbyte of physical memory, these sizes are respectively 0.5, 1, and 2 times physical memory size.

- Starting at each of these three points in the log, we modified the subsequent sequence of 10,000 queries and updates. Specifically, we inserted queries to measure resource usage during execution of the log, and to measure resource usage for commits and for two read-only queries that simulate ad-hoc reporting activity. We also inserted queries to measure database size. We call each of the three modified sequences a *benchmark interval.*
- We removed all read-only queries that lay outside the benchmark intervals (in order to reduce the total amount of time needed to run the benchmark).

We call the result of the above modifications the *benchmark log.* The benchmark is run by executing LabBase with the benchmark log as input; query output is saved in another file and compared between different benchmark runs to confirm consistent operation of the various versions of LabBase.

9.2 Description of the Benchmark Intervals

In more detail, the benchmark interval at each of the three database-file sizes consists of the following queries:

1. set baseline for resource usage
2. commit
3. get incremental resource usage
4. set baseline for resource usage
5. 10,000 queries and updates from the initial log
6. 2 report queries added to the log to simulate ad hoc queries (see Appendix B.4)
7. get incremental resource usage
8. set baseline for resource usage
9. commit
10. get incremental resource usage
11. get database size

A few additional queries for measuring resource usage were also inserted into each benchmark interval to aid in understanding differences between the systems under test and confirm the correct operation of the tests; details are available from the authors. There are no commits in the initial log, so the commits inserted in the benchmark intervals are the only commits in the benchmark log. Avoiding commits outside of benchmark intervals reduces the total time needed to run the benchmark, because commits require a significant amount of time in both storage managers being tested.

As described in Section 2.2, the queries and updates performed by each workflow activity can be divided into five broad categories: (*i*) retrieve the materials in a given state; (*ii*) query the results of other workflow activities; (*iii*) insert new material instances into the database; (*iv*) insert new step instances into the database; (*v*) modify the workflow state of materials. Table 2 lists the mean number of database accesses in each category over the three benchmark intervals. The five categories are refined in Appendix B, where it can be seen (in Table 5) that the variance in workload composition from one benchmark interval to the next is negligible.

Database Access Category	Mean # Accesses
Retrieve materials from state	88.7
Query workflow results	259.6
Insert new material instances	1438.0
Insert new step instances	3909.7
Modify material states	4304.0
Total	10,000

Table 2: The Average Composition of a Benchmark Interval

10 Storage-Manager Comparisons

The purpose of our benchmark is to measure both the functional and the performance characteristics of DBMSs for managing workflow data. In the tests discussed here, we used several versions of the LabBase data server, which varied in storage management. The five versions for which we report results are:

1. OStore—a version relying on ObjectStore (v3.0) [32, 42] for storage management.
2. Texas—a version relying on the Texas storage manager (v0.3) [51, 59] for storage management.
3. Texas+TC—a version almost identical to Texas, and using the same storage manager, but with additional object clustering implemented in client code.
4. Ostore-mm and Texas-mm—versions without any persistent storage management, and running entirely in main memory.

The code running in the Ostore-mm and Texas-mm tests was identical to that for the OStore and Texas tests, respectively, except that no persistent database was used. Technically, for the Ostore-mm and Texas-mm tests all objects were allocated in the transient database. The consequence is that for Ostore-mm objects are allocated by `malloc`, whereas for Texas-mm the same allocator was used as for Texas and Texas+TC, since Texas uses its own version of `malloc` for both transient and persistent allocation.

ObjectStore and Texas present very similar interfaces to the programmer—their interface is essentially a persistent C++, in which C++ objects can be allocated in a persistent heap as well as in the familiar, transient heap. Both ObjectStore and Texas provide a distinguished mechanism for allocating objects on the persistent heap, but once an object is allocated all code that uses it can be oblivious of whether it is persistent or transient. The ObjectStore and Texas interfaces are so similar that out of approximately 11,000 lines of C++ code constituting the LabBase server, only a few hundred differ between the two storage managers. These differing sections are demarcated by precompiler directives (*e.g.*, `#ifdef/#endif` blocks.) Roughly speaking, the LabBase server implements the top level of the schema in Figure 1 (the part above the dashed line) directly in C++. Thus this part of the schema is compiled in. The lower level is implemented by making entries in a data dictionary, and is defined by database users at run time. The `involves` relationship is implemented by keeping a (C++) pointer from each material instance to a list of pointers to step instances related to that material by `involves`.

ObjectStore and Texas also both rely on pointer swizzling at page-fault time, as described in detail in [51]. However, internally they are rather different. ObjectStore offers concurrent access with lockbased concurrency control implemented in a page server that mediates all access to the database.[21] Texas does not support concurrent access, and Texas programs access their database files directly.

Both the OStore and Ostore-mm versions used ObjectStore indexes. The Texas, Texas+TC, and Texas-mm versions used b-tree indexes supplied by the authors of the Texas storage manager (though not part of the standard Texas distribution).

10.1 Performance Comparison

The benchmark machine for the results reported here was a Sun 4/50 (SPARCstation IPX) running SunOS 4.1.3. The system had 32Mbyte of main memory, and two SUN0424 disk drives. System swap space was allocated on one of the drives. The benchmark machine was not used for any other purpose during benchmarking, but the benchmarks were not run in single-user mode. For Texas the database file, the executable, and the log were allocated on the other disk. Because of licensing restrictions and the undesirability of disturbing our production ObjectStore-based applications, we kept the ObjectStore database file, log, and page server on a separate machine, a SPARCstation 20. The ObjectStore cache file (an `mmap`ed file used as temporary backing store by ObjectStore) was on the benchmark machine. We discuss the possible performance impact of these varying configurations later in this section.

Table 3 summarizes the results for the five different versions of the LabBase data server.

In each interval, query 7 measures resource usage for the 10,000 queries and updates from the initial log, plus the two additional queries that simulate ad hoc reporting queries. Query 11 measures database file size after a commit at the end of the benchmark interval.

At a nominal database size of approximately 0.5X main-memory size, all five versions of the LabBase data server consume similar resources, except for database-file size, which is 46% greater in Texas+TC and Texas than in the OStore version. (Of course, database file size is not meaningful for the Ostore-mm or Texas-mm versions.) The larger size of the Texas database file is due to

- a 64-bit pointer size for disk-based data, and
- a segregated allocation scheme that rounds object sizes up to approximately the nearest power of 2, and which for the object sizes in LabBase left a substantial proportion of allocated space unused.

Starting at a nominal database size of approximately 1X main-memory size, the Texas version began to thrash, and performance decayed so quickly that it seemed unfruitful to allow the benchmark run to continue to the 2X benchmark interval. The data-server process, as observed by SunOS's `ps` (1), was almost continually in 'D' state (non-interruptible wait—presumably for disk I/O). The larger database size in Texas (than in OStore) cannot be the primary cause of the problem, because the number of hard page faults (majflt) measured at query 7 for OStore at the even larger file size at the 2X-main-memory benchmark interval was only 153,742, as opposed to 2,718,898 for Texas at the 1X interval. In our use of ObjectStore in previous implementations of the production version of LabBase, we saw the beginnings of thrashing behavior, so we employed ObjectStore's ability to assign objects to storage segments (instances of the `os_segment` class) in an attempt to achieve better locality of reference than would be achieved in a single persistent heap. The OStore implementation of LabBase uses four such segments, three of which contain relatively small amounts of frequently accessed data and one of which contains a relatively large amount of infrequently accessed data.

[21] Thus, when based on the ObjectStore storage manager, the LabBase server is what Carey et al. term a "client-level server" [11].

		Database Server Version				
Intvl	Resource	OStore	Texas+TC	Texas	Ostore-mm	Texas-mm
0.5X	elapsed sec	1,424	1,469	1,402	1,384	1,407
	user cpu sec	1,381	1,449	1,385	1,364	1,383
	sys cpu sec	16	7	6	5	5
	majflt	329	468	397	463	696
	size (bytes)	16,629,760	24,281,088	24,600,576	—	—
1X	elapsed sec	3,243	2,783	63,214	309,305	2,570
	user cpu sec	2,568	2,613	2,679	3,453	2,501
	sys cpu sec	137	15	2,958	20,981	8
	majflt	9,847	5,190	2,718,898	17,451,919	2,433
	size (bytes)	33,824,768	48,259,072	48,889,856	—	—
2X	elapsed sec	7,057	n/a	n/a	n/a	n/a
	user cpu sec	4,711	n/a	n/a	n/a	n/a
	sys cpu sec	812	n/a	n/a	n/a	n/a
	majflt	153,742	n/a	n/a	n/a	n/a
	size (bytes)	62,136,320	n/a	n/a	—	—

Table 3: Resource Usage for five Storage-Management Schemes.

Elapsed time and user and system cpu were measured by SunOS's `times` *(3V) function (system-V flavor). "majflt" was measured by* `getrusage` *(2), and is the number of "page faults requiring physical I/O" (i.e., major faults). All resource usage was measured at query 7, except for the size of database files, which was determined using* `stat` *(2V) at query 11. The column labeled "intvl" indicates a particular benchmark interval at 0.5, 1, or 2 times physical memory size.*

To test the hypothesis that poor locality of reference was the chief cause of Texas's thrashing behavior, we ran the same data server as in OStore, but without any persistent database, and consequently without the benefit of clustering based on the `os_segment` class; we called this test Ostore-mm. To our surprise Ostore-mm displayed far worse thrashing behavior than Texas, and we deemed it unfruitful (and infeasible) to continue the benchmark to the 2X interval. Seeking to understand why page-faulting behavior in Ostore-mm was so much worse than in Texas, we analyzed the object sizes in Texas, and hypothesized that Texas's segregated storage allocator, with one exception, stored frequently-accessed objects and rarely-accessed objects in different segments. In other words, by accident Texas's allocation provided reasonable storage segmentation, with the exception of one putatively mis-allocated class of objects.

To test this hypothesis, we modified the Texas code to pre-allocate extents of the putatively mis-allocated objects. (Texas does not presently offer a storage-segment mechanism.) The Texas+TC test uses this modified Texas code. The results in Table 3 show that, at the 1X benchmark interval, paging activity is indeed substantially reduced. However, as the size of the database continued to grow beyond the 1X interval, paging activity increased dramatically, with the result that we did not gather performance numbers for the 2X interval of the Texas test. We are still seeking an explanation for this behavior.

We also ran the same data server as Texas without any persistent database, in the test called Texas-mm. For unknown reasons the performance at the 1X interval is superior to that in all other

Intvl	Query	OStore	Texas+TC	Texas
0.5X	3	47	802	816
	10	8	72	66
1X	3	70	863	929
	10	21	79	87
2X	3	162	n/a	n/a
	10	22	n/a	n/a

Table 4: Elapsed Time for Commits (in seconds).

tests. We are currently investigating the reason for this performance.

None of the measurements presented in Table 3 captures commit costs. The ObjectStore page server resided on a different machine from the LabBase data server during the benchmark OStore tests, and it was not feasible for us to relate resource usage in the ObjectStore page server to activities of a particular LabBase client because the server was also being used for production activities. Therefore the only meaningful figure we can report for commits is elapsed seconds, presented in Table 4. (Commit times for Ostore-mm and Texas-mm were of course 0 and are not shown.) It is clear from this table that most commit times were over an order of magnitude lower in the OStore test than in Texas and Texas+TC. Salient hypotheses for these differences are:

1. They reflect inherent differences in the efficiency of the commit mechanisms in ObjectStore and Texas.
2. They were caused by differences in disk configuration. The database and log were on separate disks for OStore, but on the same disk for Texas and Texas+TC.
3. They were caused by cpu differences. Much of the work of commit was performed on a faster cpu for OStore than for Texas and Texas+TC.

Hypothesis 1 seems most plausible. According to Texas documentation, logging is performed only during commit, which involves two phases. In the first phase, all changes are written to a log; and in the second phase, updates are applied to the database file. Resource utilization measurements gathered during the benchmark runs show that commit for the Texas-based LabBase servers is I/O bound. For example, for the Texas+TC commit at query 2 (as measured at query 3) in benchmark interval 1X, cpu utilization was 8.2%, and 30,321 disk writes were performed in 863 seconds elapsed time, for an average rate of about 35 writes/second. Since the database file size was 48mbtye, the average size of each write was probably below 2Kbyte. It is worth observing that these tests measure the time needed to commit a large amount of work: even the commits measured at query 10 involve on the order of 10^5 updates. The relative efficiencies for ObjectStore and Texas observed for commits of large amounts of work do not necessarily obtain for commits of smaller amounts of work, and in fact Table 4 shows that the cost of the Texas commits is closer to the cost of ObjectStore commits when the amount of work being committed is smaller (*i.e.*, at query 10 instead of query 3). We plan to augment the benchmark to measure costs of committing small amounts of work.

10.2 Functional Comparison

In contrast with commercial relational database management systems there are huge differences in the functional and operational characteristics of today's object databases. Therefore a critical objective of our benchmark is to shed light on similarities and differences of such functional and operational characteristics. This section discusses such similarities and differences between ObjectStore and the Texas Storage System.

There are several categories in which we compare the two systems: concurrency control, class libraries, tuning options, and database administration.

Concurrency Control As discussed above, Texas is a single-user system (with recovery), whereas ObjectStore allows concurrent multi-user access to its databases. (In addition, ObjectStore offers a versioning capability, which is intended to support design databases and which LabBase does not use.)

Libraries ObjectStore provides an extensive class library, including collection class templates. Such class libraries substantially improve programmer productivity (and also cause the application program code to depend on a particular storage manager, as we found when we modified LabBase to use Texas). The ObjectStore collection classes are also used in ObjectStore's preprocessed query language and support ObjectStore indexes.

Tuning Options ObjectStore provides a number of tuning options, the most important of which for LabBase is the ability to group objects by storage segment. The benchmark results presented in Table 3 demonstrate that the ability to control object placement can be critical for performance. Although Texas+TC displays better locality of reference than Texas, most objects in Texas+TC are allocated in a reasonable storage segment only by accident. A robust, production-quality solution for a program relying on Texas would require the programmer to pre-allocate most objects to achieve a reasonable clustering. Such a capability belongs in the storage manager (as in ObjectStore) rather than in client code (as in Texas).

Database Administration ObjectStore provides a number of tools for confirming the consistency of databases and for analyzing their contents. ObjectStore offers a facility for schema evolution (which can be important for a persistent C++, since otherwise any change to the physical layout of class data members invalidates database contents). In the next release (v4) ObjectStore promises to offer archive logging, which can be used to roll forward a snapshot backup.

To summarize, our benchmark revealed significant differences among different storage managers and implementation of LabBase. The most notable difference between the storage managers was in facilities to control object allocation by storage segment, which critically affects performance of the benchmark application when the database size exceeds main-memory size. The benchmark also revealed significant differences in the time needed to perform commits in the two systems.

References

[1] Elmagarmid A, K, Y. Leu, W. Litwin, and M. Rusinkiewcz. A multidatabase transaction model for interbase. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 507–518, Brisbane, Australia, August 13–16 1990.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology (England)*, 215(3):403–410, October 1990.

[3] *Standard Guide for Laboratory Information Management Systems (LIMS)*. American Society for Testing and Materials, 1916 Race St., Philadelphia PA 19103, U.S.A, 1993.

[4] T.L. Anderson, A.J. Berre, M. Mallison, H.H. Porter, and B. Schneider. The hypermodel benchmark. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 317–331, Venice, Italy, March 1990.

[5] Claire M. Berg, Gan Wang, Linda D. Stausbaugh, and Douglas A. Berg. Transposon-facilitated sequencing of DNAs cloned in plasmids. *Methods in Enzymology*, 218:279–306, 1993.

[6] A.J. Bonner and M. Kifer. Transaction Logic Programming. In *Proceedings of the International Conference on Logic Programming (ICLP)*, pages 257–279, Budapest, Hungary, June 21–24 1993. MIT Press.

[7] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, April 1995. Available at `http://www.db.toronto.edu:8020/transaction-logic.html`.

[8] I. Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 1990.

[9] C. Burks, M. Cassidy, M. J. Cinkosky, K. E. Cumella, P. Gilna, et al. GenBank. *Nucleic Acids Research*, pages 2221–2225, 1991.

[10] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufmann, 1990.

[11] M.J. Carey, D.J. DeWitt, M.J. Franklin, et al. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, MN, May 1994.

[12] M.J. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 benchmark. Technical report, Computer Sciences Department, University of Wisconsin-Madison, January 1994. Available at `ftp://ftp.cs.wisc.edu/oo7/techreport.ps`.

[13] R.G.G. Cattell. An engineering database benchmark. In [25], chapter 6, pages 247–281.

[14] A. Chaudhri. An Annotated Bibliography of Benchmarks for Object Databases. *SIGMOD Record*, 24(1):50–57, March 1995.

[15] I-Min A. Chen and Victor M. Markowitz. The Object-Protocol Model, version 3.0. Technical Report LBL-32738, Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, CA, 94720, USA, December 1994. This document and others on OPM available via `http://gizmo.lbl.gov/DM_TOOLS/OPM/opm.html`.

[16] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1994.

[17] *Communications of the ACM*, 34(11), November 1991. Special issue on the Human Genome Project.

[18] U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan. Third generation TP monitors: A database challenge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–397, Washington, DD, May 1993.

[19] E. Dyson. Workflow. In *Forbes*, page 192. November 23 1992.

[20] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, CA, 1992.

[21] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested sagas. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 14(1), March 1991.

[22] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems*, 3(2):119–153, April 1995.

[23] 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.

[24] Nathan Goodman, Steve Rozen, and Lincoln Stein. Requirements for a deductive query language in the MapBase genome-mapping database. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, pages 259–278. Kluwer, 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/requirements.ps`.

[25] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann, San Mateo, CA, 1991.

[26] M. Hsu, Ed. Special issue on workflow and extended transaction systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 16(2), June 1993.

[27] M. Hsu, Ed. Special issue on workflow systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 18(1), March 1995.

[28] H.V. Jagadish, I.S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 113–124, San Jose, CA, May 1995.

[29] Setrag Khoshafian and Marek Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing.* John Wiley & Sons, Inc., 1995.

[30] R.A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming (JLP)*, 12(1&2):121–146, January 1992.

[31] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[32] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[33] S. Leutenegger and D. Diaz. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 22–31, Washington, D.C., May 1993.

[34] V.M. Markowitz and A. Shoshani. Representing extended entity-relationship structures in relational databases: A modular approach. *ACM Transactions on Database Systems (TODS)*, 17(3):423–464, September 1992.

[35] D.C. Mattes. LIMS and good laboratory practice. In [38], pages 332–345. 1985.

[36] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

[37] R.D. McDowall. Introduction to laboratory information management systems. In [38], pages 1–16. 1985.

[38] R.D. McDowell, editor. *Laboratory Information Management Systems: Concepts, Integration, Implementation*. Sigma Press, Wilmslow, U.K., 1985.

[39] C.B. Medeiros, G. Vossen, and M. Weske. WASA: A workflow-based architecture to support scientific database applications. In *International Workshop and Conference on Database and Expert Systems Applications (DEXA)*, London, U.K., Sept 4–8 1995.

[40] C. Mohan. Tutorial: A survey and critique of advanced transaction models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 521, Minneapolis, MN, May 1994. Tutorial.

[41] Allen S. Nakagawa. *LIMS: Implementation and Management*. Royal Society of Chemistry, Thomas Granham House, The Science Park, Cambridge CB4 4WF, England, 1994.

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

[43] P. O'Neal. The set query benchmark. In [25], chapter 5, pages 209–245.

[44] R. Reiter. On formalizing database updates: Preliminary report. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, March 1992.

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

[46] Steve Rozen, Lincoln Stein, and Nathan Goodman. Constructing a domain-specific DBMS using a persistent object system. In M.P. Atkinson, V. Benzaken, and D. Maier, editors, *Persistent Object Systems*, Workshops in Computing. Springer-Verlag and British Computer Society, 1995. Presented at POS-VI, Sep. 1994. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1994/labbase-design.ps.Z`.

[47] Steve Rozen, Lincoln Stein, and Nathan Goodman. LabBase: A database to manage laboratory data in a large-scale genome-mapping project. *IEEE Engineering in Medicine and Biology*, 1995. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1995/labbase.ps.gz`.

[48] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.

[49] O. Serlin. The history of debit credit and the TPC. In [25], chapter 2, pages 19–117.

[50] A. Sheth. Workflow automation: Applications technology and research. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 469, San Jose, CA, May 1995. Tutorial. Slides available at `http:/www.cs.uga.edu/LSDIS`.

[51] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems (POS-V)*, San Minato, Italy, September 1992. Available at `ftp://cs.utexas.edu/pub/garbage/texaspstore.ps`.

[52] A.H. Skarra and S.B. Zdonick. The management of changing types in an object-oriented database. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 483–495, 1986.

[53] Lincoln Stein, Steve Rozen, and Nathan Goodman. Managing laboratory workflow with LabBase. In *Proceedings of the 1994 Conference on Computers in Medicine (CompMed94)*. World Scientific Publishing Company, 1995. In press. Available at `ftp://genome.wi.mit.edu/pub/papers/Y1995/workflow.ps.Z`.

[54] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 storage benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2–11, Minneapolis, MN, May 1993.

[55] Michael Strathmann, Bruce A. Hamilton, Carol A. Mayeda, Melvin I. Simon, Elliot M. Meyerowitz, and Michael J. Palazzolo. Transposon-facilitated DNA sequencing. *Proc. Natl. Acad. Sci. USA*, pages 1247–1250, February 1991.

[56] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases*. The Benjamin/Cummings Publishing Company, Inc., 1993.

[57] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18:197–222, June 1986.

[58] Helmut Wächter and Andreas Reuter. The ConTract model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–264. Morgan Kaufmann, San Mateo, CA, 1992.

[59] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *International Workshop on Object Orientation in Operating Systems*, 1992. Available at `ftp://cs.utexas.edu/pub/garbage/swizz.ps`.

A Appendix: Benchmark Database Schema

As described in Section 4, the database contains two main kinds of object: *materials* and *steps*. In object-oriented terms, the database has a material class and a step class, with subclasses representing different kinds of materials and steps. This appendix describes the schema of these subclasses. Primitive data types for class attributes are `string`, `integer`, `boolean`, `dna_sequence` and any class name. Type constructors are `list` and `tuple`. When the data type is a class name, the value of the attribute must be the id of some instance of the class. In addition to the classes described here, the database has a binary relation called `involves` that connects materials to the steps that use them, as described in Section 4.

Materials: Each kind of material is a subclass of the class `material`, and inherits from it the attributes `name` and `state`.

```
material:
    name: string
    state: string

clone: ISA material
    length: integer

tclone: ISA material

transposon: ISA material
```

Steps: Each kind of step is a subclass of the class `step`, and inherits from it the attributes `who` and `when`. Intuitively, the `create` step indicates the creation of new materials, *i.e.*, new instances of the class `material`.

```
step:
    who: string
    when: string

create: ISA step
    material: material
    name: string

pick_tclones: ISA step
    material: clone

associate_tclones: ISA step
    parent_transposon: transposon
    parent_clone: clone
    child_tclone: tclone

t_position: ISA step
    material: tclone
    left_length: integer
    right_length: integer
```

```
        mapping_successful: boolean

    select_tclone: ISA step
        material: tclone

    determine_sequence: ISA step
        material: tclone
        l_sequence: dna_sequence
        l_start: integer
        l_length: integer
        r_sequence: dna_sequence
        r_start: integer
        r_length: integer
        sequencing_successful: boolean

    assemble_sequence: ISA step
        material: clone
        clone_sequence: dna_sequence
        t_list: list(tuple(tclone, integer))
        assembly_successful: boolean
```

B Appendix: Benchmark Workflow

A DNA molecule can be thought of as a sequence of characters, called *bases*.[22] Determining the sequence of bases in a DNA molecule is called *sequencing*. The benchmark in this paper is based on a simulated workflow for DNA sequencing. The workflow is a collection of biological experiments and computational activities carried out in a coordinated manner. It is designed to efficiently sequence DNA molecules about 3,000 bases long, called *clones*. Determining the DNA sequence of the entire Human genome, which has about 3 billion bases, would require sequencing several million clones, which would require several million executions of the workflow.

The next subsection describes the biological motivation and laboratory protocol upon which the LabFlow-1 benchmark is based. Those comfortable with a high degree of abstraction might want to skip to the subsequent subsections, which describe the workflow, the simulation, and other aspects of the benchmark.

B.1 DNA Sequencing

Although a DNA molecule can be thought of as a sequence of characters, it is not possible to "read" long DNA sequences the way one reads a magnetic tape. The main problem is that present technology can "read" only 300–400 consecutive bases. Accurate sequencing of a 3,000-base clone thus requires about twenty separate reads (since each read is performed twice to ensure accuracy). Another problem is that after reading 300 bases, reading the *next* 300 bases[23] is prohibitively expensive to use on a large scale. Therefore, laboratories usually do not read long DNA sequences sequentially.

[22] The alphabet of DNA consists of four characters, a, t, c and g, which denote four different nucleic acids.
[23] via a technique known as *primer walking*.

A common solution is to make many 300-base reads at random (and unknown) locations in a DNA sequence.[24] Unfortunately, because this process is random, many reads are required to ensure that the entire sequence is read. In fact, before each base in the sequence is read twice, many bases will be read more than twice, and some will be read many times. These extra reads increase the cost of sequencing each clone.

A more efficient (and complex) way to sequence long DNA molecules is known as *transposon-facilitated sequencing* [55, 5]. The workflow in our benchmark is based on this technique. In transposon-facilitated sequencing, a *transposon* is inserted into many copies of a clone at random positions. The transposon is a short piece of DNA with known sequence. A clone with a transposon inserted into it is called a *transposant*, or, as we call it in the LabFlow-1 schema, a tclone. Tclones have two critical properties: (i) the position of the transposon in a tclone is easily determined; and (ii) we can read the DNA sequence of the tclone in the neighborhood of its transposon. Tclones thus provide a kind of addressing scheme for a clone: they provide a sample of random "addresses" within the clone where we can read the clone's DNA sequence directly (though we can still read only 300–400 bases with each read operation). To determine the entire DNA sequence of a clone, one selects a set of tclones with inter-transposon distances of under 350 bases. One then reads the DNA sequence to the left and right of each transposon. These reads ensure that each base in the clone is read twice. In this way, the number of reads needed to sequence a 3,000-base clone approaches the minimum of approximately twenty.

B.2 The Workflow

Figure 4 shows a graph representing a transposon-facilitated sequencing workflow. Each node (or oval) in the graph represents a workflow activity, and each arc represents the flow of materials from one activity to another. This particular workflow involves two kinds of materials, *clones* and *tclones*, both of which are DNA molecules. Arcs labeled c represent the flow of clones from one workflow activity to another, while arcs labeled t represent the flow of tclones. Note that the graph has a path (with cycles) for clones, and another path (without cycles) for tclones. These two paths correspond to two (coordinated) production lines in the laboratory, one for clones and one for tclones. Intuitively, clones are "pumped" into the top of the workflow graph, then they "flow" through the graph, and finally they "drain" out of the graph at the bottom. When a clone enters the workflow (at the create_clone activity), its DNA sequence is unknown; and when a clone leaves the workflow (at the assemble_sequence activity), its DNA sequence is known.

Tclones play a subservient role in the workflow: they facilitate the sequencing of clones. Tclones are created by the pick_tclones activity in Figure 4, and enter the workflow at this point. This activity creates many tclones for each clone. The "chicken foot" arcs in the figure are intended to denote this 1-to-n relationship between clones and tclones. Not all tclones are useful for sequencing clones however. When a tclone turns out to be useless, it exits the workflow graph to the right, at one of two possible failure points.

The workflow activities in Figure 4 correspond roughly to the step classes in Appendix A. Sometimes, however, a single activity involves several steps. This is often the case when an activity involves several materials. For instance, for each clone, the pick_tclones activity creates many tclones. This activity therefore inserts into the database a single instance of the `pick_tclones` step and many instances of the `create` step.

Each workflow activity produces a stream of database accesses, including queries and updates.

[24] This is done by breaking up the long sequence into many small fragments, each of length 300-400 bases, and reading each small fragment separately.

Each activity creates and inserts new step instances into the database, and certain activities also insert new material instances. These inserts record the results of individual workflow activities. Certain activities also query the results of other activities. As materials flow through the workflow graph, each activity processes a stream of materials and generates a stream of database accesses. The confluence of the query/update streams from all activities forms the core of the workload for the laboratory's DBMS. (The workload also includes some data-dredging and report-generation queries, as described in Sections 3 and 9, and in Section B.4 below.)

There are *input states* associated with each activity. An activity takes materials from its input states, processes them, and then puts the materials into new states that depend on the outcome of the processing. The new state may be a terminal state, but usually it is the input state for some other activity. In this way, materials flow from one activity to another. The notion of "state" has both a physical realization in the laboratory, and a computational realization in the lab's DBMS.

B.3 The Simulation

The LabFlow-1 benchmark is based on a simulation of the workflow in Figure 4. The simulator is a collection of six programs, one for each node in the figure. Each program simulates a workflow activity. Like real workflow activities, these programs create and manipulate instances of the material and step classes given in Appendix A. Likewise, each program takes material instances from input states, processes them, and puts them into new states. To simulate a workflow, the six programs are run one after the other in a fixed order in a loop against a LabBase database. The loop runs forever, or until it is interrupted, simulating the continual creation and processing of materials. All queries and updates generated by the six programs are recorded in a log.

Below, we give a description of each of the six programs and the database accesses they generate. In this description, we say "clone" instead of "instance of class `clone`", and we say "tclone" instead of "instance of class `tclone`". Moreover, when we say that an instance is inserted into the database, this implies that its attributes have been filled with simulated values of the right data type.

create_clones: This program simulates the creation of clones in the laboratory, providing the workflow with a continual stream of new clones. The program ensures that the workflow is always operating on 5 clones whose DNA sequence is unknown. The program performs the following specific actions:

1. Issue a query to count the number of clones in the workflow that do not yet have an assembled sequence.
2. Do nothing unless this number is less than 5; otherwise create enough new clones to raise it to 5.
3. Insert the new clones into the database, and put them into an initial workflow state (the input state for the pick_tclones activity).

pick_tclones: In the laboratory, tclones (called children) are derived from clones (called parents) by copying the clone and inserting into it a special DNA sequence (called a transposon). The pick_tclones program simulates this activity. It performs the following specific actions for each clone in its input state:

1. Generate 32 new tclones, and insert each one into the database.
2. Associate each tclone with its parent clone; *i.e.*, create new instances of the `associate_tclone` step, and insert them into the database. (See Appendix A.)

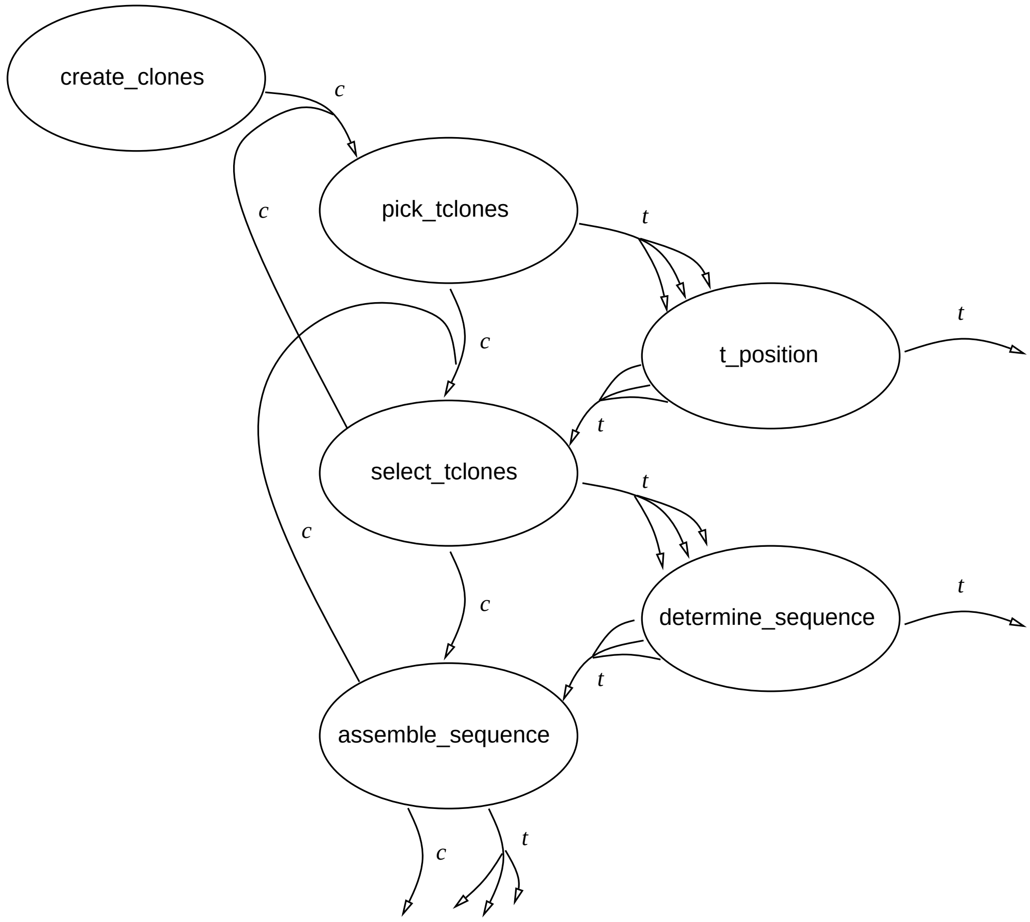

Figure 4: The workflow used to generate the benchmark log.

This graph represents the benchmark workflow from a sample-centric, dataflow perspective. Each node (or oval) represents a workflow activity, which is implemented by a program with the same name (and which is described in more detail in the text). Two kinds of material, called clones and tclones, flow through this graph. Arcs labeled c represent the flow of clones between activities, and arcs labeled t represent the flow of tclones. The "chicken-foot" arcs denote a 1-to-n relation between the executions of certain workflow activities. In particular, the t_position *and* determine_sequence *activities are executed many times for each execution of* pick_clones, select_tclones, *or* assemble_sequence. *The upward arc from* assemble_sequence *to* select_tclones *is followed when the* assemble_sequence *operation fails. It causes a clone to return to an earlier workflow state for more processing. The upward arc from* select_tclones *to* pick_tclones *indicates a similar re-do loop. The work on a clone or tclone is complete when it moves along the c or t arc (respectively) descending from* assemble_sequence. *In addition, certain tclones are discarded when they move along the arcs extending to the right of* t_position *and* determine_sequence.

3. Move the tclones to their next workflow state (the input state for the t_position activity.)
4. Insert an instance of the `pick_tclones` step into the database. (See Appendix A.)
5. Move the clone to its next workflow state (the input state for the select_tclones step).

t_position: For each tclone, the laboratory measures the position of the transposon in the tclone's DNA sequence. This process (called *mapping*) is not always successful. The t_position program simulates this mapping process with a 90% success rate. The program performs the following specific actions for each tclone in its input state:

1. Create a new instance of the `t_position` step, and insert it into the database.
2. If the mapping was successful, then move the tclone to its next workflow step (the input state for the select_tclones activity).
3. If the mapping failed, then move the tclone to a failure state.

select_tclones: For each clone, the laboratory selects some of its tclones to use in determining the clone's DNA sequence. The select_tclones program simulates this selection process. It performs the following specific actions for each clone in its input state:

1. Retrieve all the associated tclones that (*i*) have been successfully mapped (by the t_position activity), and (*ii*) have not failed to be sequenced (by the determine_sequence activity).
2. From this set of tclones, search for a minimal subset whose transposons are evenly distributed across the DNA sequence of the parent clone at intervals of at most 350 bases (*i.e.*, the end points of adjacent transposons are at most 350 bases apart).
3. If such a subset of tclones does *not* exist, then it is premature to select tclones for sequencing. Therefore, move the clone back to the input state for the pick_tclones activity (to create more tclones for the clone).
4. If such a subset of tclones *does* exist, then the DNA sequence of the clone can be determined from partial DNA sequences of the tclones. So, mark the tclones as selected, and move both the parent clone and the selected tclones to their next workflow states. Specifically, for each tclone in the selected subset, create a new instance of the `select_tclone` step, and insert it into the database. If the tclone is in the input state for select_tclones, then move it to the input state for determine_sequence.[25] Finally, move the parent clone to the input state for assemble_sequence.

determine_sequence: The laboratory reads the DNA sequence of a tclone in an (approximately) 350-base neighborhood of its transposon. This process is not always successful. The determine_sequence program simulates this sequencing process with a 90% success rate. It performs the following specific actions for each tclone in its input state:

1. Create a new instance of the `determine_sequence` step, and insert it into the database.
2. If the sequencing was successful, then move the tclone to its next workflow step (the input state for the assemble_sequence activity).
3. If the sequencing failed, then move the tclone to a failure state.

[25] Because of the workflow loop between activities select_tclones and assemble_sequence, a given tclone may be selected many times. The first time it is selected, it moves from the input state for select_tclones to the input state for determine_sequence.

assemble_sequence: The laboratory assembles (*i.e.*, computes) the DNA sequence of a clone from the overlapping, partial DNA sequences of its tclone children. The **assemble_sequence** program simulates this process. It operates on clones whose selected tclones (see **select_tclones**) have *all* been sequenced (see **determine_sequence**). For each such clone in its input state, the program performs the following specific actions:

1. Assemble the sequence of the clone from the selected tclones.
2. Check for gaps in the assembled sequence (which can arise because some selected tclones may not have been *successfully* sequenced).
3. If there *are* gaps, then create an instance of the `assemble_sequence` step and insert it into the database, setting the value of attribute `assembly_successful` to *false*. Also, move the clone back to the input state for the **select_tclone** activity (to select more tclones for the clone).
4. If there are *no* gaps, then create an instance of the `assemble_sequence` step and insert it into the database, setting the value of attribute `assembly_successful` to *true*, and the value of attribute `t_list` to a list of the successfully sequenced child tclones. Finally, move the clone to a 'finished' state, move the successfully sequenced tclones to a 'used' state, and move all other tclones to a 'not used' state.

As described in Sections 2.2 and 9.2, the queries and updates performed by each workflow activity can be divided into five broad categories: (*i*) retrieve the materials from the input state; (*ii*) query the results of other workflow activities; (*iii*) insert new material instances into the database; (*iv*) insert new step instances into the database; (*v*) modify the workflow state of materials. Table 2 lists the mean number of database accesses in each category over the three benchmark intervals. Table 5 provides a more detailed view, giving the number of accesses in each of the three intervals. In addition, each category of access is subdivided according to workflow activity, material kind, step kind, or some other criterion. From Table 5, it can be seen that the variance in workload composition from one benchmark interval to the next is negligible.

B.4 Report Queries

The database accesses described above are due entirely to workflow activity and consist largely of database inserts. These inserts create an event history. This is not the entire workload however. In a real environment, other human activities (such as administration, data analysis, or process re-engineering) generate database transactions that query this event history.

To simulate a more realistic workload, each of our three benchmark intervals includes the following queries (which simulate ad hoc or status-report queries):

- "For each state, how many materials are in that state?"
- "For each `clone` with an assembled sequence (*i.e.*, for each instance of the `assemble_sequence` step whose `assembly_successful` attribute has a value of *true*), retrieve the following information:
  - the number of times tclones were picked for the clone,
  - the size of the list of tclones used to assemble the clone's sequence,
  - the number of tclones associated with the clone,
  - the number of associated tclones that were successfully mapped,

- the number of associated tclones that could not be mapped,
- the number of associated tclones that were successfully sequenced,
- the number of associated tclones that could not be sequenced."

Database Access Category	Benchmark Interval 1	2	3
(*i*) Get inputs for workflow activities:			
pick_tclones	18	19	18
t_position	18	18	18
select_tclone	18	18	17
determine_sequence	17	18	17
assemble_sequence	17	18	17
total:	88	91	87
(*ii*) Query other workflow activities:			
create_clones	17	18	17
select_tclone	108	112	105
assemble_sequence	136	136	130
total:	261	266	252
(*iii*) Insert new material instances:			
clone	26	24	23
tclone	1,376	1,425	1,440
total:	1,402	1,449	1,463
(*iv*) Insert new step instances:			
pick_tclone	43	44	45
associate_tclone	1,376	1,425	1,440
t_position	1,376	1,376	1,416
select_tclones	669	630	602
determine_sequence	387	359	340
assemble_sequence	68	68	65
total:	3,919	3,902	3,908
(*v*) Modify material states:			
insert into initial state	1,402	1,449	1,463
move to new state	2,928	2,843	2,827
total:	4,330	4,292	4,290
grand total:	10,000	10,000	10,000

Table 5: Detailed Composition of each Benchmark Interval