

DataLab: Transactional Data-Parallel Computing on an Active Storage Cloud*

Brandon Rich and Douglas Thain

Department of Computer Science and Engineering, University of Notre Dame
Technical Report 2008-02

Abstract

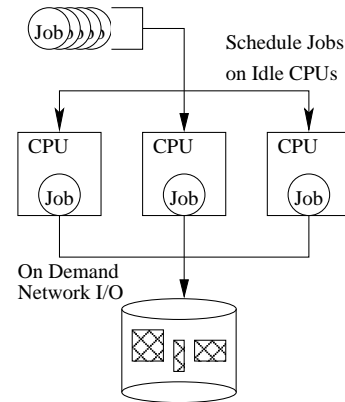
Active storage clouds are an attractive platform for executing large data intensive workloads found in many fields of science. However, active storage presents new system management challenges. A large system of fault-prone machines with local persistent state can easily degenerate into a mess of unreferenced data and runaway computations. To address this challenge, we advocate adapting the notion of distributed transactions from traditional databases. We demonstrate the use of distributed transactions in the context of DataLab, a software system for executing data parallel workloads on active storage clouds. We detail the underlying capabilities required from each node, explain how transactions are coordinated, and demonstrate the robust scaling of the system to 250 nodes while running an image processing application.

1 Introduction

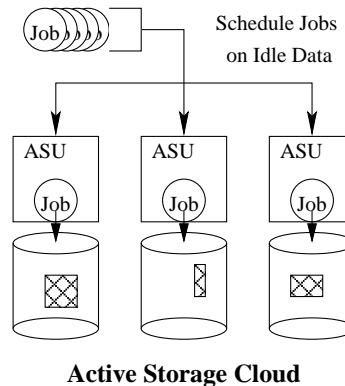
Traditional distributed computing has focused on the problem of harnessing large numbers of CPUs in order to attack partitionable computation intensive problems. However, an increasing number of applications in scientific fields are primarily data driven. Such applications are constrained not by the amount of CPU cycles available, but by the ability of the I/O system to deliver data, measured either by sequential data throughput or by number of random accesses per second. Such applications may obtain little benefit from a distributed computing system, because the benefit of distributing the computation is outweighed by the expense of moving the data to the computation.

In this paper, we consider how to design robust system software for an active storage cloud of hundreds to thousands of commodity machines. Such a system is subject to a large array of both transient and permanent failures. Ambitious scientific workloads that process terabytes of data over the course of days or weeks are certain to encounter

*This work was supported by National Science Foundation grants CNS-06-21434 and CNS-06-43229.



Conventional Compute Cluster



Active Storage Cloud

Figure 1. Cluster Comparison

such failures. If the system software is improperly constructed, transient failures can lead to aborted workloads, runaway computations, and accidentally unreferenced data, all of which require labor-intensive cleanup by the system operator.

To explore this problem, we present DataLab, a software system for executing data-parallel workloads on a testbed of 250 active storage units. To make DataLab robust against transient failures, we apply the concept of *distributed trans-*

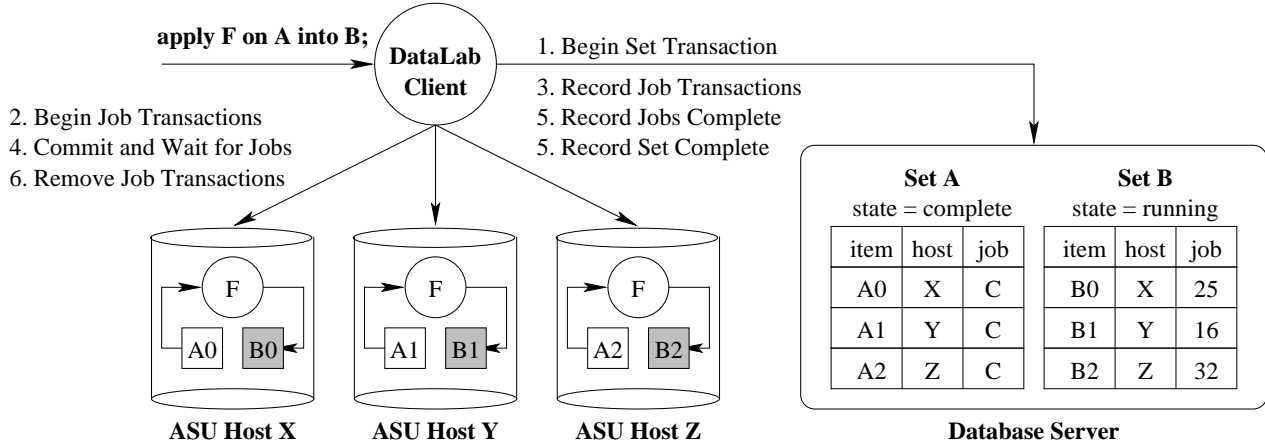


Figure 2. Architecture of DataLab

actions from the standard database literature, but relax the semantics as needed for a batch computing environment. Each individual active storage unit exports a transactional job execution interface, which in turn are aggregated into distributed transactions across the entire system. The result is a system in which a workload running for days on hundreds of nodes creating terabytes of data can be made robust against transient failures of both clients and servers. In the event of a permanent hardware failure, the system retains the ability to repair and roll forward to completion, or cleanly roll back the system to a previous state, without leaving behind orphaned computations or data.

DataLab builds on several previous works, particularly the notion of active storage [20], data-parallel computing on clusters [25, 26], and the use of declarative languages for distributed computing [4, 6, 27]. Our new contribution is a presentation of the techniques needed in order to create robust system software, and of the experience in designing, implementing, and operating an active storage cloud.

2 Overview of DataLab

A DataLab system consists of hundreds to thousands of *active storage units* that store data and execute functions, one central *database server* that keeps track of ASUs and their contents, and one or more *clients* that interact with the user and drive the system.

A user of DataLab operates on *sets*, which are named collections of files. Each element of a set can be processed by a *function*, which is an encapsulated program with one or more inputs and outputs and no hidden side effects. Three primary *commands* cause sets and functions to interact. The command *apply* creates a new set from an existing set by processing all of its elements with a function. The command *select* creates a new set from an old set by choosing

those members on which a function returns true. The command *compare* creates a new set from two existing sets by comparing all members to each other.

Figure 2 shows a small DataLab system executing the command *apply F on A into B*, which causes the function *F* to be applied on every element of the set *A*, producing a new set *B*. Note that each element of the new set *B* is retained on the same ASU where it is created. The location of each file is duly noted at the database server, but the data itself does not move. This illustrates a guiding principle of DataLab: *Avoid network data transfer wherever possible*. This property allows DataLab to scale up to a very large number of hosts by minimizing the need for an exceptional network interconnection technology.

Following are the components in more detail:

Active Storage Units. Each ASU is a commodity shared-nothing computer with its own CPU, RAM, local disk, and a Unix-like operating system. On each ASU, we run a Chirp [24] server, which serves two purposes. First, it provides a data access interface that allows clients to store and retrieve whole files, perform fine-grained I/O, and manipulate the local directory structure. Second, clients may transfer binary executables to the server for execution. Programs run on an ASU typically access data on that same ASU, but if needed, can address and access files on other ASUs.

Database Server. The database server maintains the overall state of a system. It contains the complete list of ASUs available in the system. For each set, it records the name and location of each of its members. Each function defined by a client is tracked in the database server and replicated to each node of the system. For each workload invoked by a client, the database server records a transaction that records the command details, the resources in use, the invoking user, and the start and stop times. The database server is obviously a critical component of the sys-

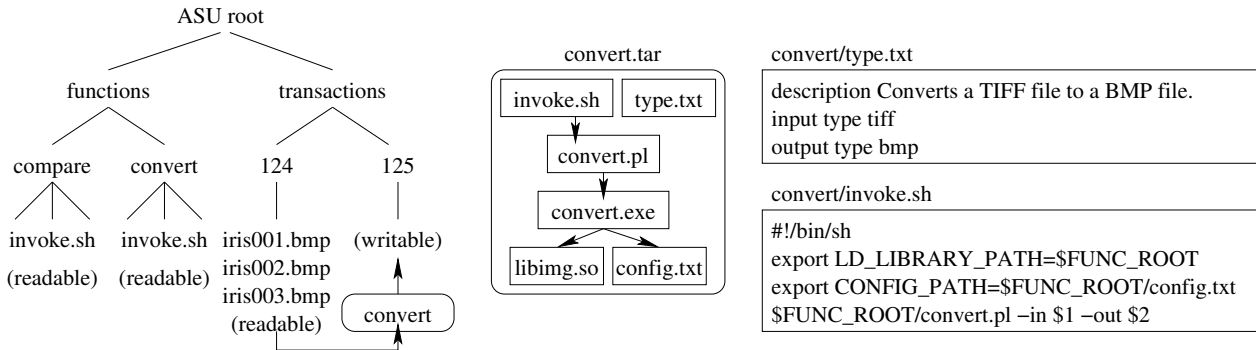


Figure 3. Detail of Function Definition and Invocation

tem, and should contain enough internal redundancy (e.g. RAID [17]) to satisfy the availability needs of the user.

Client Process. Users interact directly with a client process that may be invoked from any machine connected to the network. The client allows the user to import and export data, invoke commands, and view the status of previously executed commands by contacting the database server and active storage units as needed. Long running jobs are initiated by the client, but then proceed asynchronously. In the event of a crash or disconnection, a user may re-connect to the system and view the state of a job in progress. This is a necessity for workloads that may run for hours or days. No persistent state is recorded by the client, so multiple clients may operate simultaneously, and a given user may change terminals without loss of context.

Flow of Control. Whenever the user invokes a high level command such as `apply`, the client contacts the database server to start a new transaction and obtain the location of files to be processed. It then contacts each of the necessary ASUs to invoke jobs. As each unit completes its assigned work, the client collects the results and updates the database server as needed. To the user, this all proceeds asynchronously: the user is given a transaction number and can use the number to query the progress, block until completion, or abort the workload. This process is discussed in more detail in Section 7 below.

Functions. Programs that make use of active storage must be well disciplined. In order for DataLab to make appropriate placement decisions, it must have full knowledge of and control over the data accessed by a program. Unfortunately, most standard executables are not well disciplined: they may be scripts that invoke multiple sub-programs, they may depend on unusual shared libraries, or they may consult the environment for the location of certain files. In addition, executables have no standard calling convention: some make use of the standard input and output streams, some use files specified on the command line, while others may access hard-coded file names.

Rather than require users to re-write applications within

a given framework, we require that all programs be converted into a *function* by wrapping into a package with a standard invocation interface. Figure 3 shows a simple example of a function named `Convert`. The standard interface consists of two files which must be present in all functions. `type.txt` lists the number and types of inputs and outputs to the function. `invoke.sh` is the common entry point for all functions. An ASU can invoke any function by calling `invoke.sh` with the input and output file names given on the command line in standard order.

The body of a function must contain all of the components necessary to execute the code in a free-standing environment. For many programs, this may simply require a single executable. For others, this may require packaging up all the necessary dependencies such as sub-programs, shared libraries, and configuration files. The creator of the function must then set the necessary environment variables and invoke the program properly from `invoke.sh`. While constructing such a function is admittedly not “user friendly”, this is a necessary step in order to yield a controllable system.

Types. All files in a set are assumed to have the same type, represented by common file suffixes such as `.txt`, `.bmp`, and so forth. As noted above, each function also indicates file types for its inputs and outputs. Before invoking a command, the client will check that input types match, and assigns output types to the newly created sets. The suffix of output files is automatically set to the proper type. DataLab itself does not assign any inherent semantics to these types, but this does allow for basic error checking. For example, a user that attempts to pass `.bmp` files to `Convert` will receive a type error immediately instead of executing a large erroneous workload. In addition, typing is necessary to automatically satisfy the needs of binary programs within functions that expect particular filename suffixes.

Security. When an ASU executes a function, the process is contained within an identity box [23]. The identity box serves two purposes. First, it restricts access to files on that ASU based on the credentials presented by the client. A

client is identified by some set of credentials, which may be Kerberos [21], Globus [9], or simple hostnames. Second, the identity box implements remote access to other ASUs, allowing a job to address any unit in the system under the pathname `/chirp/hostname/path`. Of course, excessive remote communication will harm scalability, but is necessary for some kinds of interactions like `compare`.

3 Example Application

As a motivating example, we consider a workload used by biometrics researchers performing extensive image processing. Biometrics is the study of identifying humans from measurements of the body such as fingerprints, face images, body geometry, iris images, and so forth. For each of these datatypes, researchers collect a large body of data from human subjects, annotate it with information about how it was collected, share it with other institutions, and then perform a very large number of experiments on the same set of standard data. For example, a recent data collection effort at the University of Notre Dame involved collecting 60,000 images of human irises, each recording a high resolution image along with metadata such as the acquisition date, recording equipment, and so forth. Image data is processed using existing tools written in a variety of languages such as C, Perl, and Java.

Processing these large sets is a very I/O intensive, time consuming, and error-prone task. Fortunately, it is naturally parallel: most activities involve the systematic application of a function to all members of a dataset, so it is well matched to DataLab. Following is an example of how a biometrics user might employ DataLab.

To load 60,000 iris images into Datalab, the user first creates a new set called `irisArchive`, specifying that this set will contain data with the extension `tiff`. The `import` loads files from `/tmp/images` and spreads them across the storage cloud, registering each file with the database server.

```
create set irisArchive as tiff;
import "/tmp/images/" into irisArchive;
```

The images are stored in the original archive as TIFFs, but suppose this particular research group prefers to work with the simpler BMP format. To convert all images from TIFF into BMPs, the user applies a predefined function `ConvertToBMP` mapping the set `IrisArchive` into `Iris`. This causes all of the data to be converted in parallel across the system:

```
apply ConvertToBMP
on IrisArchive
into Iris;
```

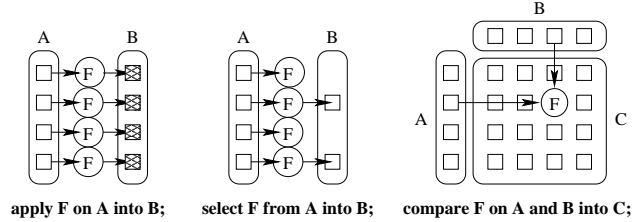


Figure 4. Basic Operations and Abstractions in DataLab

The first step in any systematic study of biometrics is segmentation. The acquisition equipment cannot perfectly center and crop the image on the iris, so instead it takes a larger image of the entire eye. A segmentation algorithm scans the image, distinguishes the pupil and iris, and outputs three pieces of data: the geometry of the iris, a template file indicating XYZ, and a mask bitmap that indicates which pixels of the image are relevant. The design of a segmentation algorithm is still an open research problem, so researchers tend to implement many variations on segmentation. To segment all of the irises into three output sets, a user may run:

```
apply Segment3B on Irises
into Geometry, Template, Mask;
```

In this case, we say that `Segment3B` has cardinality (1,3): it takes one file as an input, and produces three as output. Now, suppose that a researcher wishes to study iris images that have an unusual artifact. She designs a program `ContainsArtifact` that examines an image and template file and returns true if the image contains the artifact. Then, she uses the program to select subsets of the `Irises` and `Mask` sets:

```
select ContainsArtifact
from Irises, Mask
into ArtifactIrises, ArtifactMask;
```

Next, suppose that the researcher wishes to compare all of the selected artifact images against a small set of standard `.probe` images to observe the effect of the artifact on a standard matching algorithm. First, she repeats the above procedure to load, convert, and segment the images into DataLab:

```
create set Probes as txt;

import "/tmp/probes/" into Probes;

apply ConvertToBMP on Probes
into ProbeIrises;

apply Segment3B on ProbeIrises
```

```

into ProbeGeometry,
    ProbeTemplate,
    ProbeMask;

```

Then, she compares all probes against all artifacts, and exports the results into a text file:

```

compare Compare2D
on ProbeIris, ProbeTemplate
and ArtifactIris, ArtifactTemplate
into CompareResults;

export CompareResults
into "/tmp/results.txt";

```

The result of the comparison is simply the concatenated output of each instance of the function `Compare2D`:

```

probe001.bmp x iris003.bmp = 0.561
probe001.bmp x iris075.bmp = 0.230
...

```

This example serves to demonstrate the syntax of DataLab, and show that it maps well to applications that have bulk data processing needs. Similar kinds of workflows are found in astronomy [22], bioinformatics [1], and data mining [20]. Of course, Datalab is not a *universal* parallel programming language, but it is well-suited for an interesting category of data intensive workloads that span several fields of inquiry.

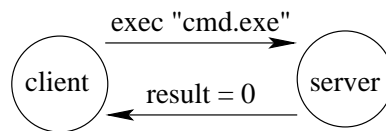
4 Need for Transactions

Failures are a significant presence in any large system constructed from commodity components. Kernel crashes, deliberate reboots, power failures, and network interruptions are quite ordinary occurrences. Any workload that attempts to harness hundreds or thousands of machines at once is almost certain to encounter at least one such failure such failures. How can we construct software that is robust to such problems?

In this work, we describe how to build a active storage cloud that is robust to *transient failures* that result in a loss of communication or a loss of data in RAM, but are eventually repaired by automatic measures or operator intervention and do not result in the loss of data on durable storage. We do not consider how to address permanent failures that result in the loss of durable data. Permanent failures are addressed by a completely different set of techniques such as replication, which are largely orthogonal to this work.

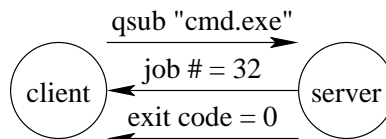
To demonstrate the problem of transient failures, consider the simple matter of executing a program `cmd.exe` on an active storage unit, assuming the program binary and input files are already in place on the system. Suppose that

we employ a standard interactive Unix tool such as `rexec` or `ssh` to perform remote execution. If we set aside some details of authentication, both of the protocols look like this:



The client sends `exec cmd.exe` to the server, and waits for a response indicating the exit status of the request. This protocol has several obvious problems: if the client and server are disconnected before the result is returned, the client cannot tell if the process has even started. If it re-sends the request, the result could be zero, one, or two processes running at once. Likewise, if the process must be stopped because it is non-terminating or is consuming too many resources, the client has no name by which to refer to it.

This is usually addressed by employing a batch execution interface like the following:



In this protocol, the process is submitted to the server using `qsub`, which returns a job id to the client, and the server may begin executing the job right away. The client may then use the job id to query or kill the process. If the job completes normally, an asynchronous message (usually an email) is sent back to the client, and the job record is deleted. This protocol is an improvement, but still has problems. If the client crashes after issuing `qsub` but before recording `jobid` to durable storage, the process is once again orphaned. Likewise, if the completion message is lost, the final job disposition is unknown.

Of course, if we consider the case of a single user trying to execute a handful of short tasks on one remote machine, it is hardly worth the trouble to solve this automatically. The user may manually login to the remote machine, peruse the process table and file system, and can likely deduce whether the desired execution failed, is still running, was killed, or completed normally.

But, if we consider a computing cloud of hundreds of nodes running thousands of jobs, perhaps triggered transparently by some higher level software such as a web portal or grid middleware, the problem becomes much more serious. If there is no foolproof way to match requests with tasks, then undesired running processes and output files will begin to accumulate on the system. If users have tasks that run for days or generate outputs measured in terabytes, then

only a few "accidents" are needed before the system capacity is seriously degraded.

Frey et al. [10] demonstrated that the solution to submission part of this problem is *two-phase commit*: the server must report a job number back to the client, who must record it in stable storage before sending a `commit` message that allows the job to begin executing. DataLab takes this another step farther by making an *entire workflow* transactional, allowing for the precise commit or abort of both computation and data across the entire system.

5 Transaction Requirements

A DataLab transaction is similar but not identical to a traditional database transaction. A database transaction is usually described as having four ACID properties: atomicity, consistency, isolation, and durability. Three of these properties do not apply to DataLab. Atomicity is not desirable: if a workload runs for hours or days, a user will want to view the overall progress and perhaps preview results. Consistency is unnecessary: apart from typechecking when a command is submitted, there are no constraints between data items. Isolation of transactions is desirable, but this enforced by the simple fact that sets are created once and are then immutable.

Instead, we describe a DataLab transaction as **ARRD**:

Asynchronous. Once a DataLab transaction is initiated by a client, it may run for hours or days before the system reports the transaction completed. During this time, the client is free to view or abort the transaction unilaterally.

Reversible. Any transaction, whether running or completed, may be rolled back to release all computation and storage resources consumed by the transaction. This is an important property, because a mistyped command or buggy function might result in the consumption of all storage and computing capacity across hundreds of nodes.

Robust. Once initiated, a transaction will never be unilaterally aborted by the system. Transient failures may temporarily prevent forward progress, but cannot prevent the eventual completion of the transaction.

Durable. A transaction is not complete until all outputs are recorded in persistent storage.

To achieve this, we define two levels of transactions. A *job transaction* is used to execute a single program on an active storage unit. A *set transaction* is recorded at the database server and encompasses all of the job transactions and data tracking needed for a single DataLab command. We consider both kinds of transactions in turn.

6 Job Transactions

To design the job transaction interface for the ASU, we begin with the job-state diagram defined by the OGSA Ba-

sic Execution Service [8] and added the two-phase commit described by Frey [10]. We removed concepts related to staging data in and out, because function running on an ASU read and write executables and data directly on the local disk, where they have a lifetime much longer than any single job.

The result is shown in Figures 5 and 6. To execute a job, the client sends **begin** to the server, which specifies the function executable, input and output files, and other details, and returns a unique job number. The job is not runnable until the client records the job number in durable storage and response with a **commit**. The local ASU scheduler will eventually run the job, and it will reach one of three terminal states of completed, failed, or killed. The client may block until completion using **wait** or poll using **status**. Once the job's final state is retrieved, the client acknowledges with **remove**.

Once a job is removed, the ASU may assume that the client has no further interest in the job, and can remove it from its active data structures. The ASU may also unilaterally remove a job left in the initial state for an unreasonable time (e.g. one day), assuming that the client has crashed. Database theory holds that the final disposition of a transaction must be held indefinitely. In practice, removed transactions are logged in a separate location and deleted at the discretion of the local administrator.

Because we assume that the typical workload in DataLab is I/O bound, there is little benefit and significant danger in time sharing a disk, where a loss of locality can seriously harm performance. Accordingly, we implement a single-slot first-come-first-served scheduler for the ASU job queue.

7 Set Transactions

A *set transaction* is a distributed transaction that encompasses multiple job transactions and tracks all of the data created by those jobs. Like a job transaction, a set transaction must proceed in a manner such that a transient failure at any stage will still permit the client to either continue to completion or abort the transaction cleanly.

As an example, we consider the transaction for the `apply` command, shown in Figure 2. (The other commands operate in a similar manner.) Figure 7 shows the state transitions for a set transaction, which are similar to those of a job transaction.

To execute an **apply** transaction, the client contacts the database server to generate a new set transaction number and data structure that records the set and function names, marks the set in the **initial** state, and returns the set transaction number to the client. The client then **commits** the set transaction into the **starting** state. In this state, the client enumerates each element in the input set, determines the

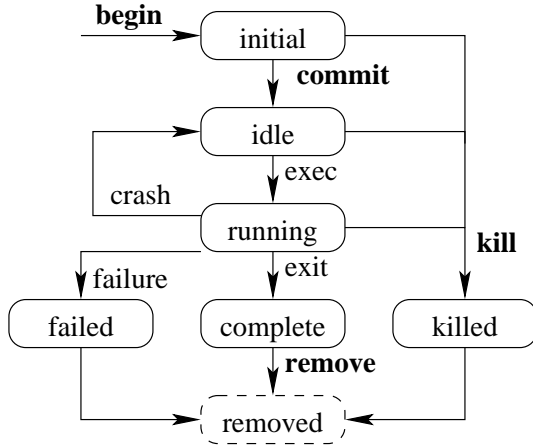


Figure 5. Job Transaction States

This figure shows the states of a job transaction. Bold arrows labels indicate client actions (see Table 6) while normal labels indicate server actions. **begin** creates a job in the initial state, but it cannot execute until the client calls **commit**. After a job reaches one of the terminal states complete, failed, or killed, the client must call **remove** to acknowledge receipt of the results.

output file name deterministically from the input file name and output type, **begins** a job transaction, records the number in the set, and **commits** the job transaction at the ASU. Once all jobs have been committed, the set is marked in the **running** state. At this point, the job is running asynchronously, and the client may continue to do other things. Periodically, the client polls each ASU involved for an update on its progress. As hosts report that they are finished, DataLab records to the database the final state of each host. When all jobs have completed, the set is marked as **complete** and may be used as input for further commands.

If the user wishes to **kill** the set transaction at any time, the client marks the set in the **aborting** state, and then **kills** and **removes** all outstanding job transactions. When this is complete, the set enters the **removing** state, and the client must contact each ASU and remove all associated output files. Once done, there is no remaining state associated with the transaction on the ASUs, and the set transaction record may be removed.

By following this discipline, DataLab is able to recover cleanly from the transient failure of any component. Whenever the client restarts after a failure, it connects to the database server to look for any set not in the **complete** state, and then proceeds to fulfill the outstanding actions in that state. If an ASU should crash and recover, it automatically restarts any job transactions in progress, and no further communication is necessary. If the database should crash and recover, it passively waits for interested clients to

Command	Arguments	Results
begin	(jobinfo)	→ jobid
commit	(jobid)	→ ok / fail
status	(jobid)	→ state
wait	(jobid,timeout)	→ state
kill	(jobid)	→ ok / fail
remove	(jobid)	→ ok / fail
list		→ (jobid,state)*

Figure 6. Job Transaction Operations

This table shows the client interface to job transactions. **begin** creates a new job and returns a job id, but the job may not run until the client calls **commit**. **status** returns the job state, owner, exit code, and so forth. **wait** blocks until the job has reached a terminal state or until a timeout expires. **kill** forcibly terminates a job. **remove** deletes the state associated with a terminated job. **list** returns information about jobs in all states.

return to set transactions in progress.

An interesting case arises when we consider the partial failure of a set transaction. It may not be possible for several elements of a set of complete processing. A function provided to DataLab is often a bespoke piece of code that may crash deterministically on certain inputs. A large acquired dataset may contain individual elements that are known to be corrupt or otherwise out of range for the desired function. On the other hand, an individual function evaluation might fail because of transient hardware faults (e.g. a cosmic ray corrupts memory) or because a temporary resource limitation (e.g. out of file descriptors.) DataLab cannot distinguish between these permanent and transient failures, so the default behavior is to retry failed jobs until they succeed. However, there are cases where the invoking user is aware of the problem, and simply wishes to accept whatever fraction of the output that DataLab is able to produce. For this, the **resign** command kills and removes any incomplete jobs and files, and then indicates that the overall set is complete, so that it can be used for later states.

8 Evaluation

Scalability. Although the focus of this paper is primarily on how to construct a robust system, we are obliged to demonstrate that the resulting system still has acceptable performance and scalability. To measure this, we deployed DataLab on a heterogeneous cluster of 250 ASUs, ranging in performance from 1.0 to 3.0 GHz, equipped with single SATA or PATA disks, all running variants of the Linux operating system. Figure 8 shows the performance of an iris segmentation function applied to a 6000 image set partitioned uniformly across a randomly chosen subset of the cluster.

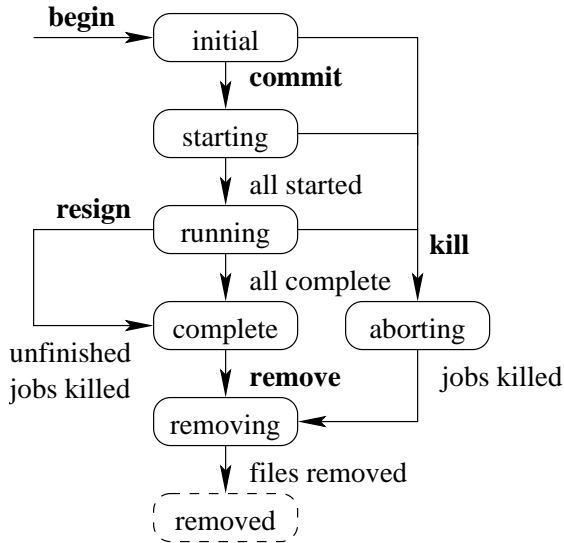


Figure 7. Set Transaction States

This figure shows the states of a set transaction. Bold text indicates explicit user actions, while normal text indicates actions taken by the client on the user’s behalf. A set in the **initial** state cannot proceed until the client calls **commit**. In the **starting** state, jobs transactions are begin, recorded, and committed. When all are started, the set transaction is **running**. When all all jobs complete, the client records the final state of each job and marks the set transaction as **complete**.

The estimated sequential execution time of this workload is fifteen hours. By adding ASUs to the system, a workload that would normally run overnight is reduced to less than ten minutes.

Figure 9 shows the effective speedup (sequential execution time divided by parallel execution time) of the system for a varying number of CPUs. As can be seen, the system does not scale perfectly linearly: at 250 hosts, the effective speedup is only 150. There are two possible reasons for this: one is that the load is not balanced equally among all nodes, the other is that there is some unremovable sequential component. We consider each possibility in turn.

Load Balancing. When the client creates a set in DataLab, its default behavior is to uniformly distribute files across each ASU. However, in a heterogeneous computing environment like DataLab, we cannot assume uniformity among the CPU and I/O capability of each ASU. In addition, there are unpredictable variations in performance that can disrupt an otherwise uniform balance. After executing a job on a set, DataLab lets the user view details on each ASU’s performance, along with a graph depicting the range of host performance. The solid line Figure 10 shows the load balance of a larger segmentation workload

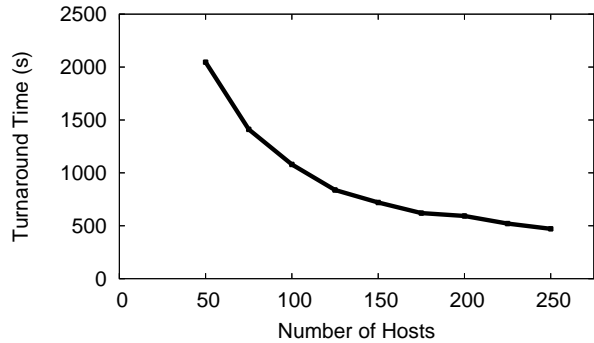


Figure 8. Scaling Performance

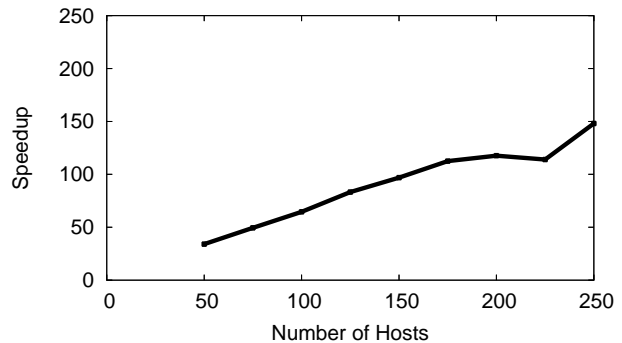


Figure 9. Scaling Efficiency

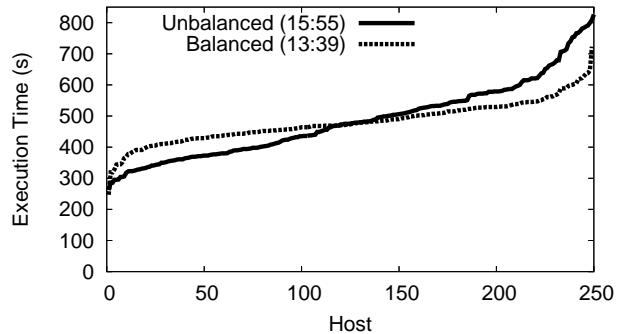


Figure 10. Effect of Load Balancing

on 250 hosts. The fastest hosts completes in 262 seconds, while the slowest in 826 seconds. Using this historical data, new datasets can be automatically balanced across the system. The dotted line in Figure 10 shows the performance of the same workload after rebalancing, reducing the execution time from 15:55 to 13:39, an improvement of 14 percent. However, there is still some imbalance remaining in the system, due to the unavoidable variances of networks and operating systems in large commodity systems.

Robustness. Finally, in order to demonstrate the robustness of DataLab, we re-ran the previous segmentation job

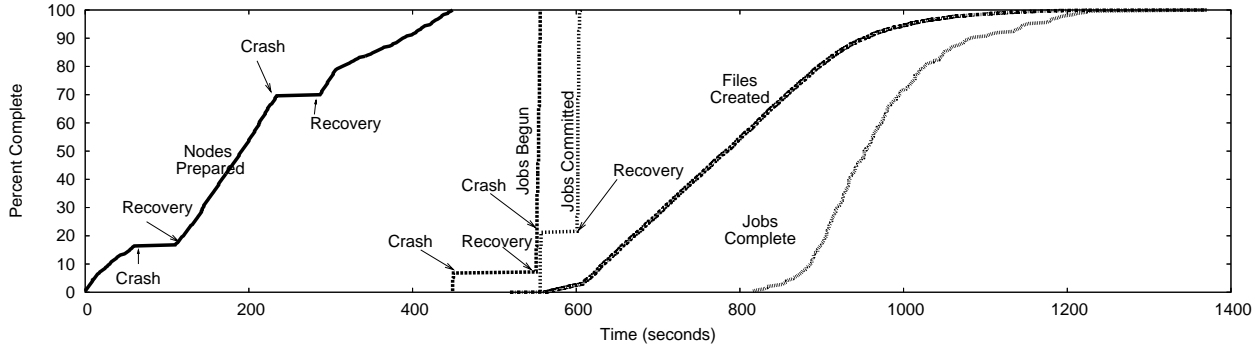


Figure 11. 250-Host Job With Failure Recovery.

of 6000 images on 250 hosts, logging the state transitions of each job transaction in the set, and (afterwards) the creation to of each file in the set. To test the recoverability of the system, we forced a client crash at various stages of the execution, and observed the recovery behavior. Figure 11 shows the progress of this job over time.

The first curve shows ASUs being prepared for execution. This is the most time consuming part of job startup. For each host, DataLab must query the database for the input files that reside there, construct a batch file listing all files to be processed, connect and authenticate to each host, and deploy the batch file. The next two curves show jobs entering the begin and commit phases. The fourth curve shows the percent of files created, and the final curve shows the percent of jobs that have been successfully completed and removed.

Several things may be observed from this graph. First, the insertion of crash events into the system causes a brief pause in the progress of the system, but as expected does not prevent the eventual completion. Second, the initialization phase in which the client must connect and distribute batch files to each node is a significant fraction of the execution time, which explains the imperfect scalability in Figure 9. Fortunately, this can be improved in future versions of DataLab by performing remote initializations in parallel.

9 Related Work

Abstractions are a powerful idea for organizing large distributed systems. Wickremisinghe, Vitter, and Chase [26] introduce the use of containers and functors for computing on active storage, and evaluate the concept through simulation. Map-Reduce [6] also provides the abstraction of set processing, using `map` for parallel set creation and `reduce` for iterative summarization. Map-Reduce processes streams of small record objects with custom code objects (i.e. disklets). DataLab also employs sets and functions, but provides different abstractions (`select` and `compare`) more

suitable to our problem domain, and manages large binary objects using existing codes and the filesystem interface.

A set is one example of a distributed data structure. Other examples of distributed data structures include hashables [11] for constructing internet services, distributed queues [3] for load balanced parallel data processing, and B-trees [16] for scalable storage systems.

The concept of *active storage* was first proposed by Riedel and Gibson [20] as a restricted computing facility integrated into low-level disk controllers. Acharya, Uysal, and Saltz [19] described the use of specialized I/O kernels called *disklets* to improve the performance of complex read operations such as database queries and image convolution using active disks. Later work has focused on applying the active storage concept to existing hardware by treating an entire conventional machine as an *active storage unit* (ASU) [14] within a larger network. Such ASUs have been used to accelerate common actions in distributed filesystems such as MVSS [15], Lerna [2], and Lustre [7].

A large body of work has explored the use of active storage clusters for processing *read only* queries on large datasets. For example, Diamond [12] enables large scale searching by using active storage techniques to discard most data without incurring network traffic. Dryad [13] enables the construction of large graphs of pipelined processes that implement complex SQL queries. DataCutter [5] enables the efficient query of large multi-dimensional arrays such as microscope image archives. Sawzall [18] is Map-Reduce program that performs aggregation and distributed query on stream-record data. DataLab has a somewhat different focus; the goal is not to perform efficient data subsetting for immediate export, but to allow for the efficient creation of new data sets through multiple steps without requiring data to leave the cluster.

10 Conclusion

In this paper, we have argued that parallel and distributed systems, even those running ordinary binaries on unstructured data, should borrow lessons from the database literature on distributed transaction processing. By introducing the notion of transactions, the entire computation and data set of a system is cleanly encapsulated, allowing for a large workload to be continued in the face of failures, or cleanly aborted when desired. Although DataLab is by no means a universal programming language, it is sufficient to demonstrate how transactions can be integrated into large data intensive workloads. Our future work will focus on making the DataLab language more expressive for a larger class of applications.

References

- [1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.
- [2] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Lerna: An active storage framework for flexible data access and management. In *High Performance Distributed Computing*, 2005.
- [3] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of IOPADS*, May 1999.
- [4] G. Belloch. Programming parallel algorithms. *Communications of the ACM*, March 1996.
- [5] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, 2000.
- [6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.
- [7] E. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active storage processing in a parallel file system. In *LCI Conference on Linux Clusters: The HPC Revolution*, 2005.
- [8] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA basic execution service v1.0. Open Grid Forum Document GFD.108, June 2007.
- [9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, San Francisco, CA, November 1998.
- [10] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *IEEE High Performance Distributed Computing*, pages 7–9, San Francisco, California, August 2001.
- [11] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *USENIX Operating Systems Design and Implementation*, October 2000.
- [12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *USENIX File and Storage Technologies (FAST)*, 2004.
- [13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.
- [14] X. Ma and A. Reddy. Implementation and evaluation of an active storage system prototype. In *Workshop on Novel Uses of System Area Networks*, Cambridge, MA, Feb 2002.
- [15] X. Ma and A. L. N. Reddy. MVSS: an active storage architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(9), September 2003.
- [16] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as a foundation for storage infrastructure. In *Operating System Design and Implementation*, 2004.
- [17] D. A. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD international conference on management of data*, pages 109–116, June 1988.
- [18] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):227–298.
- [19] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, Carnegie-Mellon University, 1997.
- [20] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large scale data mining and multimedia. In *Very Large Databases (VLDB)*, 1998.
- [21] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 191–200, 1988.
- [22] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, and D. R. Slutz. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *SIGMOD Conference*, 2000.
- [23] D. Thain. Identity boxing: A new technique for consistent global identity. In *IEEE/ACM Supercomputing*, November 2005.
- [24] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global file system for cluster and grid computing. *Journal of Grid Computing*, to appear in 2008.
- [25] J. Weissman and A. Grimshaw. Network partitioning of data parallel programs. In *IEEE High Performance Distributed Computing*, 1994.
- [26] R. Wickremesinghe, J. Vitter, and J. Chase. Distributed computing with load managed active storage. In *IEEE High Performance Distributed Computing*, July 2002.
- [27] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.