Preprint of article to appear in the Journal of Cluster Computing, Special Issue on e-Science, 2009.

Experience with BXGrid: A Data Repository and Computing Grid for Biometrics Research

Hoang Bui · Michael Kelly · Christopher Lyon · Mark Pasquier · Deborah Thomas · Patrick Flynn · Douglas Thain

Abstract *Research in the field of biometrics depends on the effective management and analysis of many terabytes of digital data. The quality of an experimental result is often highly dependent upon the sheer amount of data marshalled to support it. However, the current state of the art requires researchers to have a heroic level of expertise in systems software to perform large scale experiments. To address this, we have designed and implemented BXGrid, a data repository and workflow abstraction for biometrics research. The system is composed of a relational database, an active storage cluster, and a campus computing grid. End users interact with the system through a high level abstraction of four stages: Select, Transform, AllPairs, and Analyze. A high degree of availability and reliability is achieved through transparent fail over, three phase operations, and independent auditing. BXGrid is currently in daily production use by an active biometrics research group at the University of Notre Dame. We discuss our experience in constructing and using the system and offer lessons learned in conducting collaborative research in e-Science.*

1 Introduction

Research in the field of biometrics depends on the effective management of large amounts of data and computation. Current research projects in biometrics acquire many terabytes of images and video of subjects in many different modes and situations, annotated with detailed metadata. To study the effectiveness of new algorithms for identifying people, researchers must exhaustively compare large numbers of measurements with a variety of custom functions. The quality of the end results is often dependent upon the sheer amount of data marshalled to support it.

Department of Computer Science and Engineering, University of Notre Dame

Unfortunately, large scale experiments currently require a heroic level of expertise in computer systems. Users must be effective at configuring and using grid computing systems, relational databases, distributed filesystems, and be aware of the many underlying functional constraints and performance interactions. Once a problem is solved at a small scale, there is no guarantee that it can be simply expanded without employing new techniques. Data, tools, and techniques are difficult to share even between researchers at the same institution, because they rely on a complex stack of hand tuned software.

To address these challenges, we have constructed BXGrid, an end-to-end computing system for conducting biometrics research. BXGrid assists with the entire research process from data acquisition all the way to generating results for publication. Because the entire chain of research is kept consistently within one system, multiple users may easily share tools and results, building off of each other's work. BXGrid also helps to ensure scientific integrity by automating a variety of consistency checks, external data audits, and reproduction of existing results.

In this paper, we describe the motivating scientific need, the design and architecture of the system, and our experience in building and operating it in a production mode. A brief introduction to biometrics is necessary to describe the nature of the data and the high level abstraction of four stages: Select, Transform, All-Pairs, Analyze. We describe the architecture of the system, which consists of a relational database, an active storage cluster, and a computing grid, each specialized to carry out one component of the workflow. We describe how the system is used to ingest, manipulate, and preserve data throughout its liftime. Our current implementation has been used to store about 172,864 images and movies totalling 2.1 terabytes, and is currently ingesting data at the rate of one terabyte per month.

BXGrid is a collaboration between a systems research group and a biometrics research group at the University of

Notre Dame. The development of the system has taken a number of unexpected turns. Along the way, we have learned the following lessons which may prove to be useful to others embarking on similar projects. We briefly state each lesson here, and then elaborate upon them in Section 8 below.

1. Get a prototype running right away.
2. Ingest provisional data, not just archival data.
3. Allow objects to have many different names, each serving a distinct purpose.
4. Use crowdsourcing to divide and conquer burdensome tasks.
5. Don't use an XML representation as an internal schema.
6. Treat data consistency as an important goal, but not an operational invariant.
7. Embed deliberate failures to achieve fault tolerance.
8. Allow outsiders to perform integrity checks.
9. Expect events that should "never" happen.
10. Let the users guide the interface design... up to a point.


```
<Recording id="nd4R25000">
  <URL root="nd4/"
    relative="Fall2007/02463d1650.tiff"/>
  <CaptureDate>11/13/2007</CaptureDate>
  <CaptureTime>13:00:00</CaptureTime>
  <Format value="tiff"/>
  <Camera name="LG2200"/>
  <Subject id="nd1S02463">
  <Application>
    <Iris>
      <Eyes>
        <Eye which="Left"
          color="Brown" Pose="0"
          Motion="Still"
          treatment="No"
          conditions="default">
        </Eye>
      </Eyes>
      <Stage id="nd4T00014"/>
      </Subject>
      <Collection id="nd4C00010"/>
      <Environment id="nd4E00029"/>
      <Sensor id="nd4N00016"/>
      <Illuminant id="nd4I00011"/>
      <Illuminant id="stdI0001"/>
      <Weather condition="Inside"/>
      <Wearing glasses="No"
        source="Retrospectively"/>
    </Iris>
  </Application>
  <Description>shot number=1</Description>
</Recording>
```

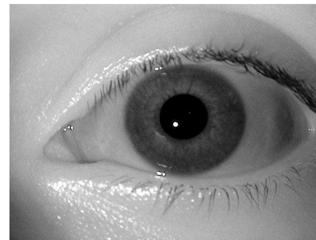



Fig. 1 Sample Iris and Metadata

2 Biometrics Research at Notre Dame

Biometrics systems are designed to verify an identity claim or choose an identity from a known set using a measurement of a physical trait. The most popular biometric today is the fingerprint [13]. Other biometrics such as the iris [2], the shape of the hand [9], and the face [22] have been studied and characterized to a point where commercial products are available, and more esoteric biometrics such as the shape or photometric appearance of the ear [21] are currently being explored. There remain many open research questions in the field, particularly how to make biometrics effective for imperfect recordings and diverse populations.

The Computer Vision Research Lab (CVRL) at the University of Notre Dame acquires a large amount of biometric data. This data is used internally to design and test new biometric algorithms, and is also exported to national standards agencies to develop rigorous tests for commercial biometric systems. All data are collected under the provisions of an experimental protocol reviewed annually by the University and Human Subjects Review Board; a consent form is required from every subject at each data acquisition opportunity. Multiple images over time from a wide variety of subjects from different demographics are needed. Sample collection efforts have generated approximately 10 terabytes of raw and processed data since 2002, and 75 gigabytes per week is likely to be collected in the near future. Assuming we keep two copies of all the data, this gives us 150 gigabytes of data being acquired every week. This means every seven weeks, which is about the number of times we acquire data every semester, we need another terabyte of storage space for all the data acquired. Once we acquire data, it needs to be organized in such a way that is easily accessible for future use. Furthermore, we need to store multiple copies of the data for redundancy so that we can recover data if it is lost, and ensure the integrity of all copies of data. Finally, biometric samples include metadata in addition to images, videos, and other sensor outputs. Every biometric sample is accompanied by the identity of the corresponding subject, the sensor in use, the time of day, a characterization of illumination (if applicable), and other extrinsic labels. Each human subject also has a metadata record storing ethnicity, age, and other demographic attributes. This metadata accompanies the samples, is used to index them to generate subsets for experiments, and must be maintained along with sample data.

Currently, images are stored in an AFS [7] filesystem named by (1) the date the data was acquired (2) the biometric acquired and (3) the sensor used to acquire the data.

For example, we are currently acquiring 12 still images and video of each subject's iris on an LG 2200 EOU iris camera. So, for a particular day of acquisition, we have a folder labeled with the date of acquisition and type of data, called *200x-xxx-lg-still*, that contains 12 images per subject. Similarly, we have another folder called *200x-xxx-lg-video*. We then know that the iris data of the *14th* subject's data acquired on day *64* of *2008* is located in folders *2008-064-lg-still* and *2008-064-lg-video* and has the prefix *2008-064-014* before each of the corresponding images. Corresponding files store the metadata in a format mandated by government sponsors, shown in Figure 1

While this organization is simple to achieve, it has many drawbacks. It is difficult to search for samples with given properties, because this requires combing through each metadata file exhaustively. Adding new data to the system is very labor intensive because each acquisition session requires custom scripting to generate the metadata and name the files, and each image must be studied manually. Because a single file server cannot support the load imposed by large experiments, users typically work by copying data out of the repository to local disks for execution. As a result, results are scattered across many different computers in many different forms, and it is almost impossible to share results in any rigorous way. Instead of a filesystem, we need a *repository* that supports all of these activities efficiently, encouraging users to reuse and share results as much as possible.

3 Abstractions for Biometrics Research

We are motivated by the advice of Gray [6], who suggests that the most effective way to design a new database is to ask the potential users to pose several hard questions that they would like answered, temporarily ignoring the technical difficulties involved. In working with the biometrics group, we discovered that almost all of the proposed questions involved combining four simple *abstractions* shown in Figure 2:

- **Select(R)** = Select a set of images and metadata from the repository based on requirements R, such as eye color, gender, camera, or location.
- **Transform(S, F)** = Apply function F to all members of set S, yielding the output of F attached to the same metadata as the input. This abstraction is typically used to convert file types, or to reduce an image into a feature space such as an iris code or a face geometry.
- **AllPairs(S, F)** = Compare all elements in set S using function, producing a matrix M where each element M[x][y] = F(S[x],S[y]). This abstraction is used to create a *similarity matrix* that represents the action of a biometric matcher on a large body of data.
- **Quality(M, D)** = Reduce matrix M into a metric D that represents the overall quality of the match. This could be a single value such as the rank one recognition rate, or a graph such as an ROC curve.

Given these abstractions as an interface to the repository, we can now compose a variety of fundamental research questions in biometrics. As a starting point, we will show some examples of **Select()** that can be expressed in SQL:

Q1 Find all irises for subjects who are male, Asian, and born after 1985.

```
SELECT * FROM irises LEFT JOIN subjects
USING(subjectid) WHERE gender = 'Male'
AND race='Asian' AND YOB>1985
```

Q2 Find all face images for whom the corresponding subject also has blue eye images.

```
SELECT * FROM faces WHERE subjectid IN
(SELECT subjectid FROM irises
 WHERE color='Blue')
```

Q3 Find all subjects for whom we have both a video clip and a still image acquired in the last week.

```
SELECT * from subjects WHERE subjectid IN
 (SELECT DISTINCT subjectid FROM face_videos
  WHERE date > "2008-01-01" INTERSECT
  SELECT DISTINCT subjectid FROM faces
  WHERE date > "2008-01-01")
```

So far, these queries allow us to extract data of interest from the repository, but the real power comes from the ability to execute an entire experiment using the remaining abstractions. In the following examples, a query like those above has been compressed into a **Select()** expression whose results are further processed:

Q4 For a given selection of data, which matcher (M or N) provides more accurate results? To answer this, compute the quality of a similarity matrix for each:

```
S = Select(D)
Q1 = Quality(AllPairs(Transform(S,F),M))
Q2 = Quality(AllPairs(Transform(S,F),N))
```

Q5 Does matching function M have a demographic bias? To answer this, compute the quality of its matches across several different demographics:

```
foreach demographic D {
  S = Select(D)
  Q[D] = Quality(AllPairs(Transform(S,F),M))
}
```

Q6 Is it effective to combine results from different biometric matchers? To answer this, compute a similarity matrix for multiple matchers, then average them and compare to each of the individual matchers.

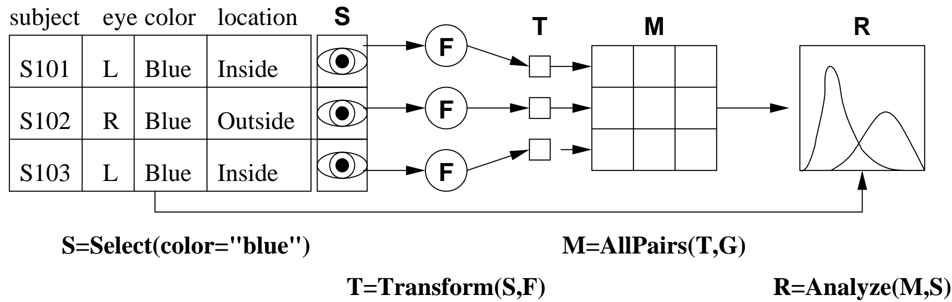


Fig. 2 Workflow Abstractions for Biometrics

```
T = Transform(Select(R))
foreach matcher M[i] {
   A = AllPairs(T,M[i])
   T += A
   Q[i] = Quality(A)
}
Q[T] = Quality(T)
```

Many different research questions in biometrics follow a similar form. By simplifying and standardizing each of these stages, we can accelerate discovery and enable more direct comparison of competing techniques.

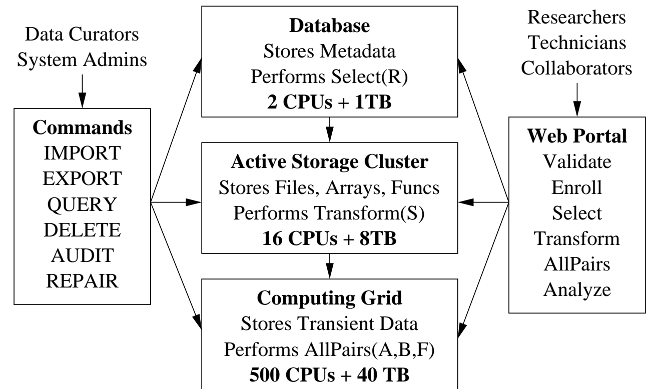


Fig. 3 System Architecture

4 System Architecture

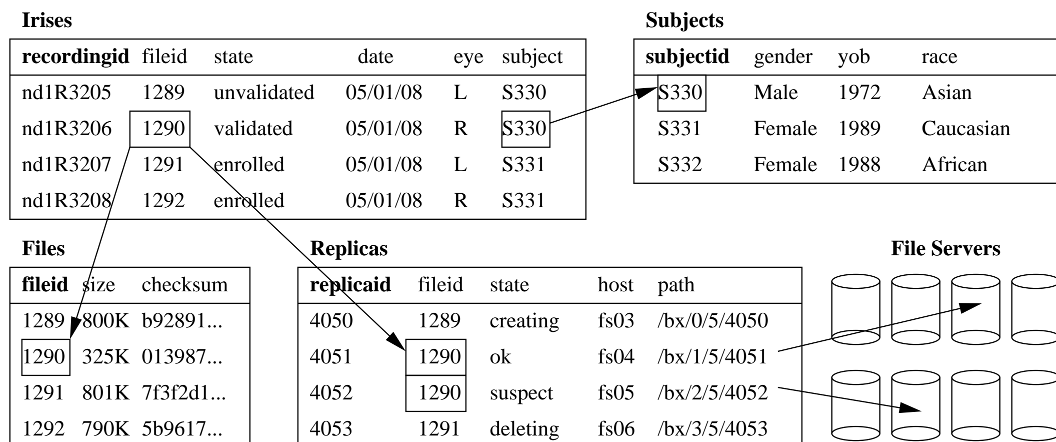
The BXGrid data repository is designed to assist in all stages of research from initial data acquisition to generating results for publication. It consists of three major components – a database, an active storage cluster, and a computing grid – each used to carry out the portion of the workload for which it is most suited. The entire system is accessible through a command line tool that facilitates batch processing, and a web portal for interactive data exploration.

Database. A conventional relational database is used to manage all of the metadata and perform the **Select** portion of each workload. Each category of data: iris images, iris videos, face images, face videos, etc. has a distinct table with a strong schema, so as to maximally exploit the query and constraint capabilities of the database. Additional relations record ancillary data such as subjects, cameras, recording environments, and so forth. An open source database running on a single conventional machine with a dual core CPU, 2 GB of RAM, and 1 TB of storage can easily scale to millions of records and serve tens of users simultaneously, so no extraordinary measures are required to achieve good performance in this component.

Active Storage Cluster. The actual images, videos, and other large data files are stored in a scalable active storage cluster. This cluster is composed of conventional machines with large local single disks, each running a Chirp [18] active storage server. Each unique file in the system is identified by a unique integer file ID, and then replicated multiple times across the cluster. The database records the list of unique files, and the location of each replica of that file. Files are immutable once added to the repository, which makes it easy to implement both fail-over and recovery: a reader must simply find any available file replicas, and a writer must simply find any available disk. Figure 4 gives an example of the relationship between metadata, files, and replicas. The iris recording R3206 refers to fileID 1290, whose size and checksum are stored in the `Files` table. The `Replicas` table indicates that this file currently has two replicas on the file servers `fs04` and `fs05`.

This is an *active storage cluster* [14] because it also provides embedded computing power. Most **Transform** operations are I/O bound and operate on a significant subset of the repository, so those small codes are shipped to the storage nodes for execution. As we will show below, this improves the performance of individual operations, and also exploits the natural parallelism of the system. In addition, we can improve capacity and performance simultaneously by provisioning new nodes without a service interruption. Our current storage cluster is an array of 16 dual-core machines, each with 2 GB of RAM and 750GB of disk.

Computing Grid. Finally, a campus computing grid is used to perform **AllPairs** operations, which are much more CPU intensive. For this purpose, we use our local 500 CPU Condor pool, where each node is also equipped with a Chirp



Irises

recordingid	fileid	state	date	eye	subject
nd1R3205	1289	unvalidated	05/01/08	L	S330
nd1R3206	1290	validated	05/01/08	R	S330
nd1R3207	1291	enrolled	05/01/08	L	S331
nd1R3208	1292	enrolled	05/01/08	R	S331

Subjects

subjectid	gender	yob	race
S330	Male	1972	Asian
S331	Female	1989	Caucasian
S332	Female	1988	African

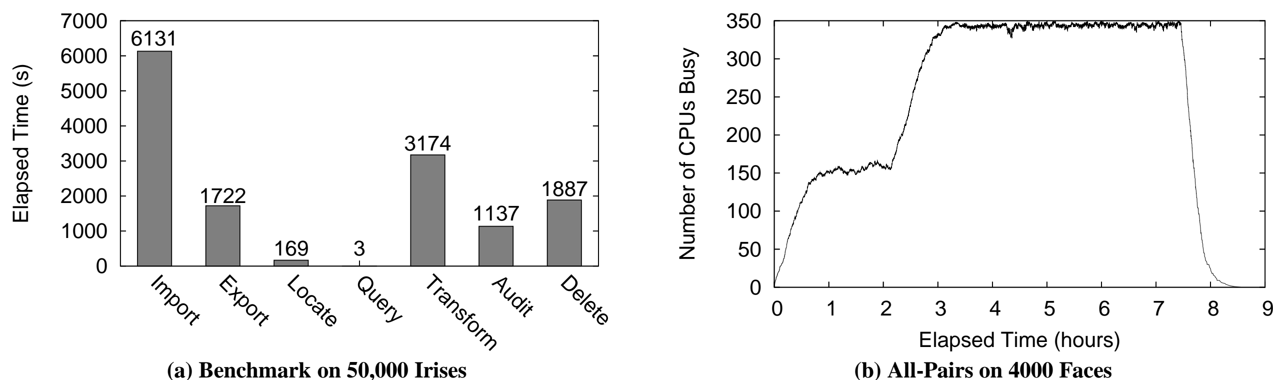
Files

fileid	size	checksum
1289	800K	b92891...
1290	325K	013987...
1291	801K	7f3f2d1...
1292	790K	5b9617...

Replicas

replicaid	fileid	state	host	path
4050	1289	creating	fs03	/bx/0/5/4050
4051	1290	ok	fs04	/bx/1/5/4051
4052	1290	suspect	fs05	/bx/2/5/4052
4053	1291	deleting	fs06	/bx/3/5/4053

Fig. 4 Fragment of Database Schema



(a) Benchmark on 50,000 Irises **(b) All-Pairs on 4000 Faces**

Fig. 5 Performance of the Repository Operations

fileserver to export each local disk. However, unlike the active storage cluster, these resources are neither reliable nor dedicated to BXGrid. CPUs may be used to run jobs, but they may be evicted at anytime according to the needs of the owner. Local disks may be used for temporary data, but may be deleted at any time. Data being processed might be intercepted by the owner of a machine or a snooper on the network, so we cannot process raw biometric data. Despite those challenges, this system is appropriate for executing the All-Pairs component of the workload. Once a dataset has been transformed to a non-invertible feature space in the active storage cluster, it can be replicated to the various nodes of the computing grid to perform an All-Pairs computation. In a previous paper [10], we described how to make All-Pairs robust and efficient for large workloads.

Command Line Tool. The lowest level interface to the system is a command line tool that automates data ingestion, export, deletion, and recovery. The operations are:

```
IMPORT    <set> FROM <metadata>
EXPORT    <set> WHERE <expr> AS <pattern>
LOCATE    <set> WHERE <expr>
QUERY     <set> WHERE <expr>
TRANSFORM <set> TO <set> USING <function>
ALLPAIRS  <set> AND <set> USING <function>
DELETE    <expr>
AUDIT     <n>
REPAIR    <n>
```

`IMPORT` loads metadata and data into the repository from the caller's workstation. `EXPORT` retrieves both metadata and data from the repository. `LOCATE` does the same, but only returns the location of files, instead of retrieving them. `QUERY` simply returns the metadata without the files. `TRANSFORM` and `ALLPAIRS` invoke the corresponding abstraction on the active storage cluster and the computing grid. `DELETE` destroys all of the metadata and files matching a particular expression; this is most commonly used to reverse an `IMPORT` of bad data. `AUDIT` and `REPAIR` are used to detect and repair corrupted data and react to reconfigurations.

Figure 5(a) shows the runtime of each of the key operations on up to 50,000 iris images of about 300KB each, with triple replication. Most operations require multiple transactions against the database and the storage cluster. `IMPORT` operates at one-third the speed of `EXPORT`, because it must make three copies of each data. `LOCATE` is significantly faster than `EXPORT` because it does not actually fetch any data, and `QUERY` is essentially instantaneous. `TRANSFORM` is used to

convert images from JPEG to TIFF format, which is a common conversion. Figure 5(b) shows the most expensive operation, ALLPAIRS, on a 4000x4000 comparison on the computing grid over the course of eight hours. The performance of this step depends on the availability of CPUs. As can be seen, only 150 CPUs were initially available, but nearly 350 became available after 2.5 hours.

Web Portal. Most end users interact with the system through the web portal, which allows for interactive browsing, data export in various forms, dataset management, and system administration. Figure 6 shows some example pages from the portal: (a) shows the validation interface, where end users match newly acquired data against existing data, (b) shows the interface for selecting and browsing datasets, and (c) shows the interface for drilling into records.

5 The Data Lifecycle

In this section, we describe how the various users and stakeholders of BXGrid interact with the system at each stage in the life of the data. Althuogh BXGrid is not directly involved in data acquisition, it is a useful starting point that explains the nature of the data and the possible errors that can be introduced.

Acquire. Several data acquisition campaigns are run each year. Each campaign involves a particular physical setting (e.g. hallway, outside), multiple sensors (camera, video, 3-D), and different poses (sitting, standing) for each subject. As subjects arrive at the lab, they must check in with an ID card, and then are guided from station to station by a lab technician. On any given day, up to eighty subjects participate, producing as many as twenty recordings each. Each is labelled with a globally unique "shot id" that indicates the date, time, subject, sensor, and comment. A number of errors can creep in at this stage. An error in transcription at check-in could associate the wrong subject with a set of images. If a subject steps out of line, an entire sequence of recordings could be mis-labeled. If a technician errs in taking a picture, a left eye could be recorded as a right eye, or vice versa. An image could be misaligned or overexposed, rendering it useless to experimentation.

Import. The lab operator imports data in batch at the end of each day or week using the command line tool. The tool checks for basic schema correctness in the input, and rejects the entire batch if the schema is incorrect or the files are missing. Otherwise, it generates a new batch number and loads the metadata and data into the repository, replicating as needed. Figure 5(a) shows the time to import, export, or delete a large number of records in BXGrid. Although the system is not yet highly optimized, the performance is sufficient to support the actual rate of data acquisition.

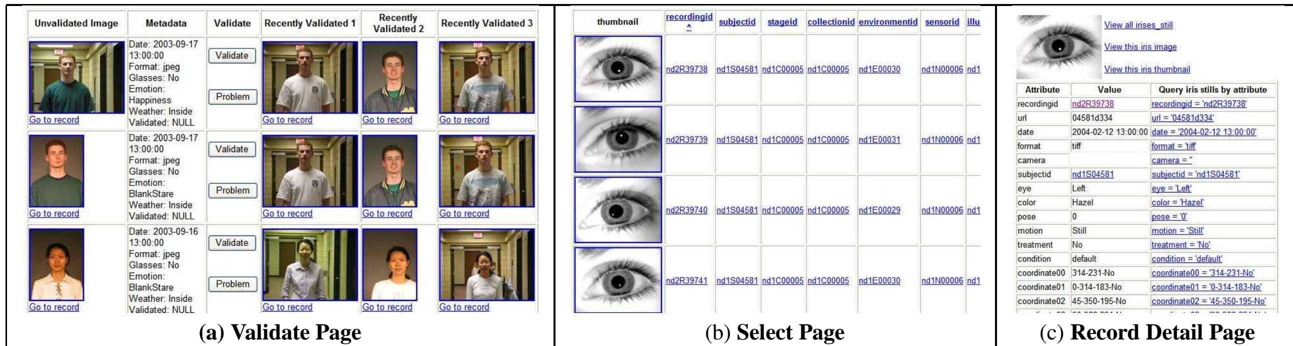
Validate. Because of the high probablity of errors in acquisition, newly imported data must be validated. All records are initially marked as `unvalidated`. For an record to be `validated`, a technician must review the image and metadata via the web portal. The portal displays the unvalidated image side by side with images taken of the same subject from several previous acquisition sessions, shown in Figure 6(a). If the technician identifies an error, they can flag it as a `problem`, which will require manual repair by a domain expert. Otherwise, the image may be marked as `validated`. By exposing this task through the web portal, the very labor intensive activity can be "crowdsourced" by sharing the task among multiple students or technicians.

Enroll. A second mark of approval is required before a recording is accepted into the repository. The curator supervising the validation process may view a web interface that gives an overview of the number of records in each state, and who has validated them. The quality of work may be reviewed by selecting validated records at random, or by searching for the work of any one technician. At this point, decisions may still be reversed, and individual problems fixed by editing the metadata directly. In the case of a completely flubbed acquisition, the entire dataset can be backed out by invoking `DELETE` on the batch id. Once satisfied, the supervisor may enroll the entire dataset through the web interface, which will mark all of the records as `enrolled`, and assign various identifiers required by outside agencies. The dataset is then fully accepted into the repository and may be used for experimentation.

Select. The first step in experimentation is to select a dataset through the web portal. Because most users are not SQL experts, the primary method of selecting data is to entire collections of data with labels such as "Spring 2008 Indoor Faces". These results can be viewed graphically and then successively refined with simple expressions such as "eye = Left". Those with SQL expertise can perform more complex queries through a text interface, view the results graphically, and then save the results for other users.

Transform. Most raw data must be reduced into a feature space or other form more suitable for processing. To facilitate this, the user may select from a library of standard transformations, or upload their own binary code that performs exactly one transformation. After selecting the function and the selected dataset, the transformation is performed on the active storage cluster, resulting in a new dataset that may be further selected or transformed. The new transformed dataset is considered to be derived from a parent dataset. Therefore, it retains most of the metadata which comes from the parent set. For example, a function transforms an iris image to an iris code. The correspondent will inherit information such as: left eye, subjectid, environmentid... from the original iris image.

All-Pairs. Likewise, to perform a large scale comparison, the user uploads or chooses an existing comparison function and a saved data set. This task is very computation



(a) **Validate Page** (b) **Select Page** (c) **Record Detail Page**

Fig. 6 Examples of the Web Portal Interface

intensive, and requires dispatch to a computational grid. Our implementation of All-Pairs is described in an earlier paper [10] and briefly works as follows. First, the system measures the size of the input data and the sample runtime of the function to build a model of the system. It then chooses a suitable number of hosts to harness, and the distributes the input data to the grid using a spanning tree. The workload is partitioned, and the function is dispatched to the data using Condor [19]. Figure 5(e) shows a timeline of a typical All-Pairs job, comparing all 4466 images to each other, harnessing up to 350 CPUs over eight hours, varying due to competition from other users. As can be seen, the scale of the problem is such that it would be impractical to run solely in the database or even the active storage cluster.

Analyze. The result of an All-Pairs run is a large matrix where each cell represents the result of a single comparison. Because some of the matrices are potentially very large (the 60K X 60K result is 28.8 GB), they are stored by a custom matrix library that partitions the results across the active storage cluster, keeping only an "index record" on the database server. Because there are a relatively small number of standardized ways to present data in this field, the system can automatically generate publication-ready outputs in a number of forms. For example, a histogram can be used to show the distribution of comparison scores between matching and non-matching subjects. Or, an ROC curve can represent the accept and reject rates at various levels of sensitivity.

Share. Finally, because BXGrid stores results at every intermediate step of the data lifecycle, users can draw on one another's results. The system records every newly created dataset as a child of an existing dataset via one of the four abstract operations. Figure 7 shows an example of this. User A Selects data from the archive of face images, transforms it via a function, computes the similarity matrix via AllPairs, and produces an ROC graph of the result. If User B wishes to improve upon User A's matching algorithm, B may simply select the same dataset, apply a new transform function, repeat the experiment, and compare the output graphs. A year later, user C could repeat the same experiment on a larger dataset by issuing the same query against the (larger)

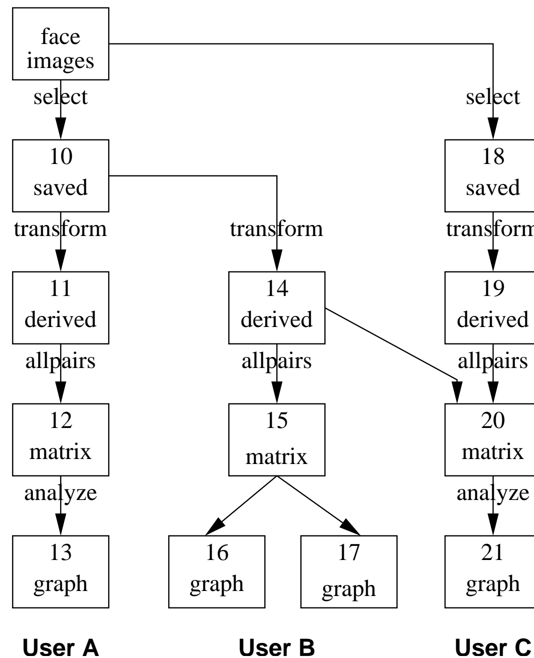


Fig. 7 Sharing Datasets for Cooperative Discovery

archive, but apply the same function and produce new results. In this way, experiments can be precisely reproduced and compared.

6 Naming

Designing a naming system for BXGrid was a considerable challenge. The CVRL already had several concurrent naming schemes that satisfied different stakeholders. However, these naming schemes were used in an ad-hoc manner, and the generation and meaning of names was not clearly documented. BXGrid requires some sort of name to uniquely identify an object, and none of the existing names applied through the entire data lifecycle. It was necessary for a joint group to meet regularly over the course of a year before the entire naming scheme was clearly defined and in production use.

Each object in the system gains the following names through the data lifecycle. Each plays a different role in the repository: For each, we state the name and give an example of an identifier.

ShotID (2008-093-020-3_R-lg4000.tiff)
This is the first name assigned to a recording in the lab, serving as the simple file name before importing into BXGrid. It specifies the date, the shot number (with respect to that date), and the sensor used to record the image or video. In the lab, a running metadata file records additional information, such as the subject's identity, indexed by each ShotID for that date.

BatchID (1232662885)
For each `IMPORT` command, BXGrid generates a BatchID, which serves as a transaction number for the operation. A batch usually consists of a few hundred recordings from a day or a week of acquisition. The primary purpose of this name is to give the importing user an easy way to `DELETE` recently imported data that has some systematic problem.

FileID (233336)
For each imported recording, BXGrid generates a FileID integer, which uniquely identifies the metadata and the associated data file (image or video). A FileID is simply incremented for each new recording, and is never re-used or changed, regardless of other names in use.

ReplicaID (698583)
Each imported file has several replicas in the system, each of which is identified by a unique ReplicaID. This allows BXGrid to unambiguously refer to a particular copy of a file, in case of data loss or corruption.

SequenceID (02463d1890)
This identifier is used internally by the CVRL as a unique identifier within experiments. It consists of the subject number (02463) and the number of recordings taken of that subject (1890). Because the subject associated with a recording is not yet verified, this name is not assigned until the recording is validated and enrolled. BXGrid records the maximum SequenceID for each subject, and automates the assignment of names during enrollment.

RecordingID (nd5R65000)
A RecordingID uniquely identifies a recording transmitted to the sponsoring agency. It uniquely identifies a recording across projects at multiple institutions, so `nd5` indicates Notre Dame, series 5, and `65000` indicates the recording number. BXGrid also automates the assignment of RecordingIDs to recordings during enrollment.

7 Reliability and Availability

BXGrid must be a highly robust system. First, it must be *reliable*: once imported, data in the system should survive the expected rate of hardware failures, and automatically migrate as new hardware is provisioned. Second, it must also be *available*. Acquisition of data occurs on dozens of weekdays during the academic year. Students and faculty interact with the system to do research at all hours of the night and day. Data analysis tasks may take days or weeks. Good performance is also desirable, but not at the expense of reliability and availability.

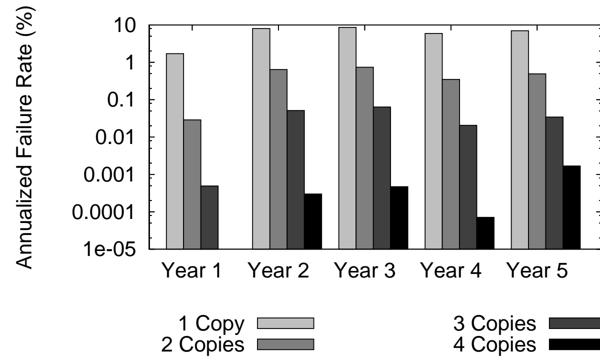


Fig. 8 Expected Failure Rate for Replicated Data

Figure 8 shows the expected probability of data loss due to disk failure based on the values observed by Google [12], which are significantly higher than those reported by manufacturers. For years one through five in the life of a disk, the annualized failure rate f is the probability that the disk will fail in that particular year. The probability of data loss of two disks is simply f^2, three disks f^3, and so forth. For three data copies, the probability of failure is less than 0.001 percent in the first year, and less than 0.1 percent in years two through five. To sustain the data beyond the conventional disk lifetime of five years, we should plan to provision new equipment

Transparent Fail Over. Because the active storage cluster records each replica as a self-contained whole, the failure of any device does not have any immediate impact on the others. Operations that read the repository retrieve the set of available replicas, then try each in random order until success is obtained. Operations that import new data select any available file server at random: if the selected one does not respond, another may be chosen. If no replicas (or file servers) are available, then the request may either block or return an error, depending on the user's configuration. Given a sufficient replication factor, even the failure of several servers at once will only impact performance.

Sustaining acceptable performance during a failure requires some care and imposes a modest performance penalty on normal operations. Each file server operation has an internal timeout and retry, which is designed to hide transient failures such as network outages, server reboots, and dropped TCP connections. Without any advance knowledge of the amount of data to be transferred, this timeout must be set very high – five minutes – in order to acommodate files measured in gigabytes. If a file server is not available, then an operation will be retried for up to five minutes, holding up

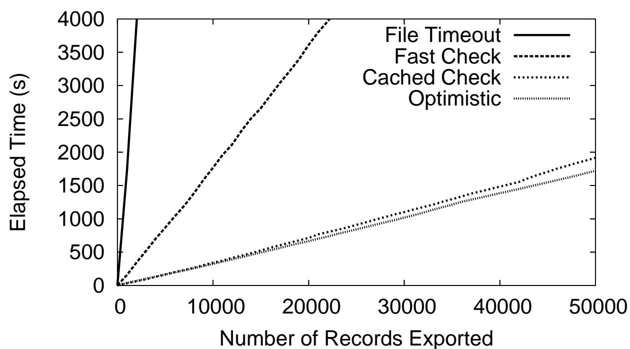


Fig. 9 Performance of Transparent Failover Techniques

the entire workload. To avoid this problem, we add an inexpensive test for server health before downloading a file: the client requests a `stat` on the file with a short timeout of three seconds. If this succeeds, then the client now has the file size and can choose a download timeout proportional to the file size. If it fails, the client requests a different replica and tries again with another service. Of course, this test also has a cost of three seconds on a failed server, so the client should cache this result for a limited time (five minutes) before attempting to contact the server again.

Figure 9 demonstrates this by comparing the performance of several variations of transparent failover while exporting 50,000 iris images. The “Optimistic” case has all 16 servers are operating and simply downloads files without any additional checks. The remaining cases have one file server disabled. “File Timeout” relies solely on the failure of file downloads, and makes very little progress. “Fast Check” does better, but is still significantly slower, because approximately every 16th request is delayed by three seconds. “Cached Check” does best, because it only pays the three second penalty every five minutes. However, it is still measurably worse than the optimistic case, because each transaction involves the additional check.

Three Phase Updates. Most updates on the repository require modifying both the database server and one or more storage servers. Because this cannot be done atomically, there is the danger of inconsistency between the two after a failure. To address this problem, all changes to the repository require three phases: (1) record an intention in the database, (2) modify the file server(s), (3) complete the intention in the database. For example, when adding a new file to the system, the `IMPORT` command chooses a location for the first replica, writes that intention to the database and marks its state as `creating`. It then uploads the file into the desired location, and then completes by updating the state to `ok`. Likewise, `DELETE` records the intention of `deleting` to the database, deletes a file, and then removes the record entirely. Other tools that read the database simply must take care to read data only in the `ok` state. In the event of a failure, there may be records left behind in the intermediate states, but the `REPAIR` tool can complete or abort the action without ambiguity.

Asynchronous Audit and Repair. An important aspect of preserving data for the long haul is providing the end user with an independent means for checking the integrity of the system. Although the system can (and should) perform all manner of integrity checks when data are imported or exported, changes to the system, software, or environment may damage the repository in ways that may not be observed until much later. Thus, we allow the curator to check the integrity of a set or to scan the entire system on demand.

The `AUDIT` command works as follows. For every file, the system locates all replicas, computes the size and checksum of each replica, and compares it to the stored values. An error is reported if there are an insufficient number of replicas in the `ok` state, inconsistencies in the checksums, and replicas for files that no longer exist. In addition, the auditing tool checks for referential integrity in the metadata, ensuring that each recording refers to a valid entry in the ancillary data tables. (We do not use the database to enforce referential integrity when inserting data, because we do not wish to delay the preservation of digital data simply because the paperwork representing the ancillary data has not yet been processed.)

This is a very data intensive process that gains significant benefit from the capabilities of the active storage cluster. The serial task of interrogating the database can be accomplished in seconds, but the checksumming requires visiting every byte stored, and it would be highly inefficient to move all of this data over the network. Instead, we can perform the checksums on the active storage nodes in parallel. To demonstrate this, we constructed three versions of the auditing code. The first uses the repository like a conventional file system, reading all of the data over the network into a checksum process at the database node. The second uses the active storage cluster to perform the checksums at the remote hosts, but only performs them sequentially. The third dispatches all the checksum requests in bulk parallel to sixteen active storage units. We measure the performance of each method on 50,000 iris images of about 300KB each:

Audit Method	**Execution Time**	**Speedup**
Conventional File System	5:43:12	1X
Sequential Active Storage	1:39:22	3.4X
Parallel Active Storage	0:08:21	41.1X

When the repository is scaled up to a million recordings, then the parallel active storage audit can be done in a few hours, while the conventional method would take days. For even larger sizes,the audit can be done incrementally by specifying a maximum number of files to check in the given invocation. This would allow the curator to spread checks across periods of low load. The `REPAIR` command does the

same as AUDIT but also repairs the system by making new replicas and deleting bad copies.

8 Lessons Learned

Like many e-Science projects, BXGrid is a collaboration between two research groups: one building the system, and the other using it to conduct research. Each group brought to the project different experience, terminology, and expectations. Although the overall system has been a success, the overall development did not proceed exactly according to plan. The following lessons summarize some of our experiences that may be of value to other e-Science projects.

Lesson 1: Get a prototype running right away. In the initial stages of the project, we spent a fair amount of energy elaborating the design and specifications of the system. We then constructed a prototype with the basic functions of the system, only to discover that a significant number of design decisions were just plain wrong (We describe many of these below). Simply having an operational prototype in place forced the design team to confront technical issues that would not have otherwise been apparent. If we had spent a year designing the "perfect" system without the benefit of practical experience, the project might have failed.

Lesson 2: Ingest provisional data, not just archival data. In our initial design for the system, we assumed that BXGrid would only ingest data of archival quality for permanent storage and experimental study. Our first prototype ingested an entire semester's worth of enrolled data at a time, which resulted in several problems. Ingesting a semester's worth of data took days, after which it was often discovered that there was some problem in the data, requiring the entire batch to be backed out, repaired, and ingested again. Because so much time had elapsed between acquisition and ingestion, it was often difficult for lab operators to remember the exact context of a session, making it much more difficult to correct errors. Finally, leaving valuable data in a temporary space for so long left it vulnerable to system failures. With the current BXGrid design, data is ingested in a provisional state daily. In addition to minimizing the window of vulnerability, this makes the provisional data easy to explore with the entire machinery of the system. Validators can discover problems shortly after acquisition and work with lab operators to fix problems as soon as possible. An unexpected benefit of this technique is that it eliminated a number of ad-hoc methods for storing image metadata, thus enforcing a strong schema at an early stage in the data lifecycle.

Lesson 3: Allow objects to have many different names, each serving a distinct purpose. Establishing a clear definition of each of type of name used by the system was a significant and time consuming challenge in the collaboration. In our initial design, we struggled to make use of an existing name as a unique key to name every object in the system. This turned out to be a mistake, because the existing names were not stable, or did not apply throughout the lifetime of the data. For example, the RecordingID is not assigned until enrollment, and only applies to data transmitted to an external standards agency. The ShotID is unique, but might change during validation if the subject was mis-identified. After several attempts to work with these names, we finally fell back to defining a distinct set of names (FileID, ReplicaID) whose only purpose is to provide uniqueness within BXGrid. Once this was done, we could employ the system to automatically generate the other categories of names, while leaving the operators free to rename and correct errors without compromising the integrity of the system.

Lesson 4: Use crowdsourcing to divide and conquer burdensome tasks. As described above, validation is the process of manually identifying what data objects to accept for archival. In the past, validating an entire semester's worth of data was an enormous task left to one lab technician at the end of each semester. A large, monotonous task performed under time pressure by one person is inevitably error prone. In the intial design of BXGrid, we did not consider data validation to be in the scope of the project. However, once we began ingesting provisional data (Lesson 2), it became clear that portion of the data lifecycle could be machine-assisted, shared between multiple users, and performed incrementally. With the new system, a backlog of several semesters of data has been validated in a matter of days, and newly acquired data is validated by a team of ten lab technicians who can do higher quality work in much smaller increments.

Lesson 5: Don't use an XML representation as an internal schema. An important consumer of data from BXGrid is a national standards agency that accepts metadata according to a specific XML schema. Our initial design for the system used the agency's XML schema for our internal representation, and as our preferred external representation. However, this did not work well, because the agency often made minor changes in the XML representation, each of which required changes to all layers of our system. In addition, the local users of the system preferred a simpler text representation of the metadata, because this facilitated script processing of the data. After several iterations, we divorced our internal schema from the XML representation, established a simpler text representation for external use, and implemented conversion to XML as an external script. This arrangement made our internal users happier, and also empowered those in charge of communicating with the agency to tweak the XML output as needed.

Lesson 6: Treat data consistency as an important goal. but not an operational invariant. BXGrid has a number of internal consistency requirements. However, the system does not guarantee that each of the consistency requirements will hold any given time, because such guarantees would

significantly reduce the availability of the system, or otherwise inconvenience the users. Further, events outside our control (e.g. server failure) may cause these constraints to be violated, requiring the system to be unavailable until repair. For example, at the storage level, each committed file must have a minimum of three data replicas, and the state of each replica must be reflected correctly in the database. At the metadata level, each recording should have a corresponding subject. Neither of these can be fully guaranteed, because an import process may fail before completion. Further, it is not desirable to roll back incomplete operations: it is better to have two replicas than none, and it is better to preserve an incomplete record than to not preserve it at all. These and other consistency constraints are handled by the periodic scan of the auditor process.

Lesson 7: Embed deliberate failures to achieve fault tolerance. While the system design considered fault tolerance from the beginning, the actual implementation lagged behind, because the underlying hardware was quite reliable. Programmers implementing new portions of the system would (naturally) implement the basic functionality, leave the fault tolerance until later, and then forget to complete it. We found that the most effective way to ensure that fault tolerance was actually achieved was to deliberately increase the failure rate. In the production system, we began taking servers offline randomly, and corrupting some replicas of the underlying objects which should be detected by checksums. As a result, fault tolerance was forced to become a higher priority in development.

Lesson 8: Allow outsiders to perform integrity checks. Our initial claims of fault tolerance within BXGrid were met with some understandable skepticism from users. Many had lost data on commercial RAID arrays that claimed to be reliable and yet failed to reconstruct properly after a failed disk. How could an experimental system like BXGrid be any better? While we cannot claim that BXGrid is bug-free, we have found that allowing users to perform their own integrity checks can increase trust in the system. The location of each replica of a file is exposed to the user, who can directly connect and verify that data is stored correctly. Any authorized user may run their own audit process to check the integrity of the system as they see fit.

Lesson 9: Expect events that should "never" happen. In our initial design discussions, we deliberately searched for invariants that could simplify the design of the system. For example, we agreed early on that as a matter of scientific integrity, ingested data would never be deleted, and enrolled data would never be modified. While these may be desirable properties for a scientific repository in the abstract, they ignore the very real costs of making mistakes. A user could accidentally ingest a terabyte of incorrect data; if it must be maintained forever, this will severely degrade the capacity and the performance of the system. With some operational experience, it became clear that both deletions and modifications would be necessary. To maintain the integrity of the system, we simply require that such operations require a high level of privilege, are logged in a distinct area of the system, and do not re-use unique identifiers.

Lesson 10: Let the users guide the interface design... up to a point.
The system designers proved to be very poor at predicting how the end users wanted interact with the system. For example, we built a general-purpose search feature into the validation interface, that would allow users to refine the view by any property: eye color, subject, camera, etc. As it turns out, the users always wanted to group by one property – subjects – and found the general interface to be cumbersome. With a few lines of code, we were able to provide a much simpler interface that grouped all work automatically by subject, thus increasing productivity dramatically. On the other hand, end users often have no understanding of whether a proposed feature will be easy or hard to implement. For example, we have received a number of requests to make the interface more interactive by adding AJAX technologies. While this might certainly be useful, the cost to implement far outweighs the potential benefit.

9 Related Work

BXGrid is preceded and inspired by several previous examples of scientific data repositories. Like the SDSS SkyServer [17], we map the primary user interactions to a custom query language and a relational database. However, we have chosen a different set of abstractions suited to the domain and applied different underlying computer systems (an active storage cluster and a computing grid) that are more closely aligned with the user's goals. BXGrid is also similar to SDM [11] in that we have coupled a database to a file system. However, in the case of SDM, the data model is centered around n-dimensional arrays, and multiple disks are used to support high throughput I/O in MPI [5], rather than active storage and data preservation. HEDC [16] is another example of a filesystem-database combination implemented on a single large enterprise-class machine. The Storage Resource Broker [1] and its successor iRODS [20] are powerful, general-purpose tool for managing filesystem hierarchies spread across multiple devices, tagged with searchable metadata implemented as a vertical schema. BXGrid differs in that the top-level interface is a database with a strict horizontal schema pointing to files in a hierarchy (rather than the other way around) which allows for the full expressiveness of SQL to be applied.

A common design question is whether large binary objects should be stored as binary objects in a database or as files in a filesystem. Searcs [15] observes that filesystems

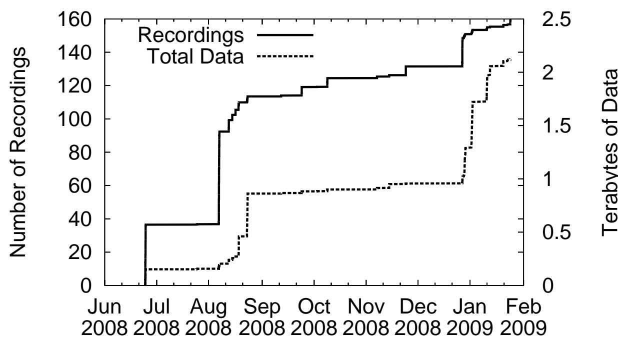


Fig. 10 System Growth Jul 2008 - Jan 2009

are more efficient than databases for objects above a certain size; 1 MB is the critical value in the particular case of NTFS and SQL Server. Although BXGrid could have been implemented solely as a clustered database, such a model would have a much more complex (and opaque) fault tolerance model, and would not allow legacy codes to address storage objects directly.

Our use of *abstractions* to represent high level workload structures is inspired by other systems such as Map-Reduce [3], Dryad [8], Swift [23], and Pegasus [4]. However, different categories of applications need different kinds of abstractions. Our workflow does not cleanly fit into any of the just-named abstractions because it encompasses several modes of data (relational, file, array) and types of computer systems (database, cluster, grid.)

10 Conclusion

Figure 10 shows the growth of BXGrid over time. The system began production operations in July 2008, and ingested a terabyte of data from previous years by September 2008. Through fall 2008, it collected daily acquisitions of iris images. Starting in January 2009, BXGrid began accepting video acquisitions, and is currently ingesting data at approximately one terabyte per month. At the time of writing, BXGrid is storing 172,864 recordings with triple replication, totalling 2.1 TB spread across 16 file servers for both reliability and performance.

The system is used daily by a dozen undergraduate operators, all of whom are trained in its use. We can confidently assign subsets of newly acquired data to students for validation, generate summaries of results to identify strong and weak performers, and handle exceptional cases (such as errors requiring metadata updates, file manipulation, or expert inspection of samples to resolve problems). The Web front-end and the support of multiple simultaneous users has removed a critical production bottleneck, and enabled data validation and enrollment within days of acquisition rather than months. Graduate students make use of the command-line interface to carry out experiments using the Select, Transform, AllPairs, Quality abstraction.

There are many avenues of future work. In biometrics specifically, there are many possible ways of computing on archived data to accelerate the scientific process. For example, the process of validating iris data is more time consuming and error prone than validating face data. Given the ability to perform All-Pairs on the computing grid, newly acquired data could be automatically compared against already acquired data to detect errors in the metadata. More generally, we believe that the concept of high level abstractions is an appealing method of making large scale computing accessible to the experts in other domains. Future work should identify what abstractions are needed in other fields of study, and what degree of re-use is possible across fields.

Acknowledgment: This work was supported by National Science Foundation grants CCF-06-21434, CNS-06-43229, and CNS-01-30839.

References

1. C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.
2. J. Daugman. How Iris Recognition Works. *IEEE Trans. on Circuits and Systems for Video Technology*, 14(1):21–30, 2004.
3. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.
4. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3), 2005.
5. J. J. Dongarra and D. W. Walker. MPI: A standard message passing interface. *Supercomputer*, pages 56–68, January 1996.
6. J. Gray and A. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science. *IEEE Data Engineering Bulletin*, 27:3–11, December 2004.
7. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.
8. 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.
9. A. K. Jain, A. Ross, and S. Pankanti. A Prototype Hand Geometry-Based Verification System. In *Proc. Audio- and Video-Based Biometric Person Authentication (AVBPA)*, pages 166–171, 1999.
10. C. Moretti, J. Bulosan, P. Flynn, and D. Thain. All-pairs: An abstraction for data intensive cloud computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.
11. J. No, R. Thakur, and A. Choudhary:. Integrating parallel file i/o and database support for high-performance scientific data management. In *IEEE High Performance Networking and Computing*, 2000.
12. E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *USENIX File and Storage Technologies*, 2007.
13. N. Ratha and R. Bolle. *Automatic Fingerprint Recognition Systems*. Springer, 2004.

14. E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large scale data mining and multimedia. In *Very Large Databases (VLDB)*, 1998.
15. R. Searcs, C. V. Ingen, and J. Gray. To blob or not to blob: Large object storage in a database or a filesystem. Technical Report MSR-TR-2006-45, Microsoft Research, April 2006.
16. E. Stolte, C. von Praun, G.Alonso, and T. Gross. Scientific data repositories . designing for a moving target. In *SIGMOD*, 2003.
17. A. S. Szalay, P. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. Technical Report MSR-TR-99-30, Microsoft Research, Feb 2000.
18. 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.
19. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.
20. M. Wan, R. Moore, and W. Schroeder. A prototype rule-based distributed data management system rajasekar. In *HPDC Workshop on Next Generation Distributed Data Management*, May 2006.
21. P. Yan and K. W. Bowyer. A fast algorithm for icp-based 3d shape biometrics. *Computer Vision and Image Understanding*, 107(3):195–202, 2007.
22. W. Zhao, R. Chellappa, P. Phillips, and A. Rosenfeld. Face Recognition: A Literature Survey. *ACM Computing Surveys*, 34(4):299–458, 2003.
23. 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.