A RICH METADATA FILESYSTEM FOR SCIENTIFIC DATA

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Hoang Bui,

Dr. Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
May 2012

A RICH METADATA FILESYSTEM FOR SCIENTIFIC DATA

Abstract

by

Hoang Bui

As scientific research becomes more data intensive, there is an increasing need for scalable, reliable, and high performance storage systems. Such data repositories must provide both data archival services and rich metadata, and cleanly integrate with large scale computing resources. ROARS is a hybrid approach to distributed storage that provides both large, robust, and scalable storage and efficient rich metadata queries for scientific applications. This dissertation presents the design and implementation of ROARS, focusing primarily on the challenge of maintaining data integrity and achieving data scalability. We evaluate the performance of ROARS on a storage cluster compared to the Hadoop distributed file system. We observe that ROARS has read and write performance that scales with the number of storage nodes. We show the ability of ROARS to function correctly through multiple system failures and reconfigurations. We prove that ROARS is reliable not only for daily data access but also for longtime data preservation. We also demonstrate how to integrate ROARS with existing distributed frameworks to drive large scale distributed scientific experiments. ROARS has been in production use for over three years as the primary data repository for a biometrics research lab at the University of Notre Dame.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

This work could not be completed without the help and support of a number of individuals to whom I owe a great deal of gratitude.

First of all, I would like to thank my advisor, Dr. Douglas Thain. During my time at the University of Notre Dame, his guidance and motivation helps me grow as a researcher and as a person. His patience and dedicated mentoring has been extraordinary and irreplaceable. I will forever be grateful for what he has done for me.

I want to thank my committee members – Dr. Patrick Flynn, Dr. Scott Emrich and Dr. Brian Blake – for being very supportive throughout this process. I very much appreciate their help and counsel which led to the completion of this work.

Additionally, I want to express my gratitude towards my friends and colleagues at Notre Dame. In particular, I must thank Christopher Moretti, Deborah Thomas, Peter Bui, Patrick Donnelly, Michale Albrecht, Li Yu, Dinesh Rajan, Clarence Helm and Diane Wright.

I also would like to thank my parents for the sacrifices they have made to give me this opportunity, my wife Thuy Nguyen for being patient and understanding, my son Leo for teaching me the meaning of life.

Finally, I wish to dedicate this dissertation to my late father, Dr. Hue Bui.

CHAPTER 1

INTRODUCTION

The first decade of the 21st century has witnessed a paradigm shift in scientific research. Powerful supercomputers and large-scale distributed systems provide scientists with a seemingly unlimited computational resource. Moreover, the foundation of data collection from late 19th century paved the way for a new research era, moving from data exploration to data explosion. Armed with improved data collection methods and advanced devices, scientists, in many fields around the world, have the ability to observe and collect an unheard amount of data for their research. This growth, however, poses new questions and challenges to scientific data management paradigm. As the scientists have changed the way they collected new data, posed new hypothesises, and explored new theories, the data management process also needs to change.

Dealing with suddenly new found data is not easy. Scientists are constantly scrambling to find storage space for the newly collected data. There are questions such as: Where should I put my data? Should I add new hard drives to my storage system? Do I need a whole new storage system to cope with the data tsunami? As newly data is collected every second, scientists simply dump everything to a hard drive. When the hard drive is full, another one can be easily added.

However, expanding capacity of the storage system is not the only challenge. The huge growth in data and storage comes with the unwanted burden of managing a large data archive. As an archive grows, it becomes significantly harder to find what data items are needed, to migrate the data from one technology to another, to re-organize as the data and goals change, and to deal with equipment failures. In other words, the complexity of the problem grows exponentially and can be very overwhelming.

Another challenge is that scientific data is hardly static, in fact it is very active. If the data is kept in a shelf and rarely looked at, the simple solution of expanding storage system's capacity seems to make perfect sense. However, in scientific research that scenario is very unlikely. Data is useless without being shared, analyzed, processed, and fed into a computationally intensive cycle to confirm a new hypothesis or to evaluate new algorithms. If all data is stored in one location and is accessible to everyone, when more people try to access the traditional storage system at the same time, it is possible to bring the system down to a crawl. Storage systems can become a bottleneck that slows down everyone. A quick solution is to replicate and store the data at multiple locations, thus increasing the availability to users. However, keeping multiple copies of the same data increases the complexity of the storage system and burdens scientists with more tasks unrelated to their research. They need to keep track of the location of each replica and to make sure to update every copy when changes are made.

Scientific data usually contains associated metadata that describe various attributes about the data. This usually includes traditional filesystem metadata such as file size, ownership, and creation time, along with higher level information such as metadata regarding the measurements or the instruments used, information about the subjects or objects being studied, and other domain-specific knowledge deemed useful to the researchers. A challenge arises when you have not tens or hundreds but tens of thousands and millions of data objects. You want to able to store and share data with your colleagues. You want to query data with certain metadata constrains. However, as quickly as data grows, the task of finding the right set of data becomes more difficult. If the dataset gets too big, it is not feasible to search for needed data by scanning the whole repository. Clearly there is a need for a storage system that supports both storing and querying metadata. Therefore, scientific repositories are not only a large capacity storage system, but a facility for searching data quickly.

The two canonical models for data storage – the filesystem and the database – are not well suited for long term data preservation. Both concepts can be made parallel and/or distributed for both capacity and performance. The relational database is well suited for querying, sorting, and reducing many discrete data items, but requires a high degree of advanced schema design and system administration. A database can store large binary objects but is not highly optimized for this task [60]. On the other hand, the filesystem has a much lower barrier to entry, and is well suited for simply depositing large binary objects as they are created. Still, as a filesystem becomes larger, data querying, sorting, and searching can only be done efficiently if they match the chosen parallel structure. As a data repository grows, no single hierarchy is likely to meet all needs. So while end users prefer working with filesystems, current storage systems lack the query capabilities necessary for efficient operation.

With all these challenges and demands, there is a clear need for a comprehensive solution to store and manage scientific data both efficiently and effectively. To allow a scientist to solely focus on their important task which is doing research and to save them from the burden of mingling with data management, this work will propose a distributed storage solution that provides both large, robust, scalable data storage with fast, reliable data query.

As a first attempt to take on the challenges, we built a prototype repository named BXGrid. BXGrid is a solution for a data repository that focuses on biometrics data. The goal of BXGrid is to help biometrics researchers at Notre Dame manage and query data in an efficient and consistent manner. BXGrid includes three main components: a database, an active storage cluster, and a computing grid. Users access BXGrid through a command line tool. During the ingestion process, BXGrid automatically replicates and store each data item in three difference fileservers. Users also use the command line tool to interact with the system using options such as export, query, and audit/repair. Beside the command line tool, a web portal provides a facility for users to browse, validate, share and manage data. Since 2008, BXGrid has been used by Computer Vision Research Laboratory (CVRL) at the University of Notre Dame to manage over ten terabytes of biometrics data and has proved to be a reliable data repository for daily usage. However, BXGrid has its limitations. BXGrid is tailored specifically to manage biometrics data. BXGrid database schema has been manually modified several times in order to accommodate changes in the initial metadata schema design. Ideally, a multi-purpose repository would be able to adjust itself when new types of metadata is ingested. BXGrid also needs improvement to keep track of metadata changes since metadata often changes throughout the life cycle of scientific data. BXGrid provides a very limited means of logging metadata changes by only recording changes which are made through the web-portal. Users can also make metadata changes using command line tool which are not recorded.

To address BXGrid's limitations, this work presents ROARS (Rich Object

ARchival System), a rich metadata filesystem for scientific data. ROARS is inspired by BXGrid's design; it is a hybrid system that leverages the strengths of both distributed filesystems and relational databases to provide fault-tolerant scalable data storage and efficient rich metadata manipulation. ROARS consists of a Metadata Server (MDS) running MySQL [44] and multiple Storage Nodes running Chirp [72]. Although there exist a number of designs for scalable storage [6, 9, 27, 29, 31, 67, 80] ROARS occupies an unexplored design point that combines several unusual features and principles that together provide a powerful, scalable and manageable scientific data storage system:

1. **Discrete object storage.** Each data object is stored as a single, discrete object on local storage, replicated multiple times for safety and performance. This allows for a compact statement of locality needed for efficient batch computing.

2. **Rich searchable metadata.** Each data object is associated with a user metadata record of arbitrary (name,type,value) tuples, allowing the system to provide some search optimization without demanding elaborate schema design.

3. **Transactional metadata update.** Metadata can be changed and updated through the life cycle of each data objects. These changes need to be handled carefully in a transactional model. Because metadata is stored at several levels, any change made needs to be propagated from top to bottom, and it is not considered committed until all locations of metadata are updated.

4. **Materialized filesystem views.** Rather than impose a single filesystem hierarchy from the beginning, fast queries may be used to generate materi-

alized views that the user sees as a normal filesystem. In this way, multiple users may organize the same data as they see fit and make temporal snapshots to ensure reproducibility of results.

5. **Transparent, incremental management.** ROARS does not need to be taken offline, even briefly, in order to perform an integrity check, to add, to decommission servers, or to migrate data to new resources. All of these tasks can be performed incrementally while the system is running, and even be paused, rescheduled, or restarted without harm.

6. **Fault Tolerance.** Storage nodes operate independently. Each storage node in the system can fail or even be destroyed without affecting the behavior or performance of the other nodes. The metadata server is more critical, but it functions only as an (important) cache. If completely lost, the metadata cache can be reconstructed by a parallel scan of the object storage.

The contribution of this work is the design, implementation and evaluation of a scientific data repository, which would behave like a regular distributed filesystem, while providing the fast querying abilities of a database.

Chapter 2 discusses a literature review of previous works relating to data storage in general. The first section of this chapter will compare and contrast between traditional filesystems, network filesystems and databases. It also provides the key properties of each system, which lead to the discussion of their advantages as well as their disadvantages. It then addresses the state of more current distributed storage systems with the emergence of NoSQL.

In Chapter 3, I describe the design and implementation of ROARS. The first section details ROARS' architecture. Each part of ROARS' architecture is equally important and affects ROARS' key attributes including: transactional database-like features, searchable metadata, and data object storage. I also discuss the key design decisions making process. While some decisions were trivial to make, others were carefully considered after many lengthy debates.

Chapter 4 provides a set of experiments, which were used to evaluate the performance of ROARS. Experiments include fundamental activities such as import, export, query and delete data. Another set of experiments exam the correctness of ROARS' properties such as failure transparency, incremental data migration, divide and conquer auditing. Chapter 4 also compares the performance of ROARS to the Hadoop [29] filesystem, a widely popular distributed filesystem. Although ROARS and Hadoop has their similarities, ROARS differentiates with Hadoop in key areas such as the way ROARS replicates raw data and handles metadata.

Chapter 5 discuses how ROARS is used in BXGrid to manage a multi-terabytes repository of biometrics data. The first section briefly introduces biometrics research and biometrics data. Then we describe the data acquisition, archival process, and common errors which can affect data quality. The next section emphasizes the importance of maintaining data integrity to improve the quality of biometrics research. The chapter concludes with steps we have taken to maintain data quality and data integrity.

Data is not very useful if it stays idle. ROARS was designed with that principle in mind. It is understandable that data would be moved around, fed into a scientific workflow and used to produce interesting and important results. The Cooperate Computing Lab at the University of Notre Dame provides users an array of tools that assist them with distributed workflows.

Chapter 6 will give an insight into how ROARS fits into the Cooperative

Computing Lab (CCL) distributed eco-system. In brief, Chapter 5 discusses the integration of ROARS in workflows using abstractions and distributed application building tools including Weaver, All-Pairs, Work Queue and Makeflow.

Chapter 7 summarizes the main contribution of this dissertation. Mainly, this work provides frameworks for a distributed storage system that can accommodate terabytes of data and can also support fast metadata query. The ability to find what you want quickly is crucial to the productivity of the system as a whole. This work takes in a number of difficult and important decisions during the design process of a distributed storage system. Data replication, metadata structure, data provenance [61], and data integrity are the common questions faced by any system architect. This work discusses some of the techniques and principles which are used to make the system more reliable and more resilient against hardware failure. The last chapter also discusses interesting lessons learned through-out the design and the implementation of ROARS so the system can be improved in future works. This work lays a distributed storage framework for researchers in other fields and encourage them to use distributed systems to further advance their study.

CHAPTER 2

RELATED WORK

A storage system for scientific data needs to satisfy dual goals. A scientific data repository is required to provide both scalable fault-tolerant data storage and efficient querying of the rich domain-specific metadata. Unfortunately, traditional local filesystems, network filesystem distributed filesystems and databases fail to meet both of these requirements simultaneously.

Decades ago, scientist often stored data in a local hard drive and share data through floppy disc or simple remote access protocols such as HTTP[24] or FTP[51]. However, as data grows sharply and the demand for data collaboration increases, local filesystems become a bottleneck very quickly. To increase the storage and sharing capability, scientists look to take advantage of network filesystem such as NFS [56] and AFS [31]. Network filesystems make sharing data easy, still, they also becomes a bottleneck when data is accessed repeatedly by multiple users simultaneously. Moreover, because of the lack of data replication, when an AFS or a NFS storage node goes down, the data is not accessible until the node is back online.

While most distributed filesystems such as the Hadoop filesystem provide robust scalable, and fault tolerant data archiving, they fail to adequately provide for efficient rich metadata operations. In contrast, database systems provide efficient querying capabilities, but fail to match the workflow of scientific researchers. The next sections exam several storage systems and their properties and discuss why they could not satisfy the dual-goals requirement for a scientific repository.

2.1 Local Filesystem

There are a wide variety of UNIX Local Filesystems such as: Ext2, Ext3, Ext4, XFS, JFS, ReiserFS, ZFS [16], [34], [39], [7], [10], [53],[2]. This subsection briefly discusses Ext2, Ext3 and ZFS.

Ext2 or the Second Extended File System was developed to correct some of the problems of the First Extended File System (Ext). Ext2 introduced several new filesystem features which have become the standard for many future Linux file systems. Ext2 supports files with 255 characters in the file name. The maximum file system size is 4TB. The maximum file size is 2 GB. A file is associated with an inode. An inode contains attributes of the file such as type, size, access rights and most important, pointers to data blocks. The block size is chosen when the filesystem is created, and can be 1024, 2048 or 4096 bytes. Depending on the size of the file, an inode's pointers may point to direct blocks or indirect blocks, which can point to other indirect blocks. Choosing a big block size will reduce the number of I/O requests but will increase the amount of wasted space due to block fragmentation. Symbolic links are stored in the inode itself, thus a symbolic link does not use any data blocks; the link operation is fast because it reads the information directly from the inode.

An Ext2 file system is divided into a number of block groups. One of the advantages of Ext2 is that it provides several redundant copies of critical file system information. Each block group holds a copy of this information (such as superblock and file system descriptors), followed by a block bitmap, inode bitmap, inode table and data blocks. In case of the system crashing, super block and file system descriptors can easily be restored. Ext2 also implements some performance optimization techniques, such as readahead: when a block is read, several contiguous blocks are also read. In addition, when data are written to a block, up to 8 adjacent blocks will be pre-allocated.

Ext3 (Third Extended Filesystem) is the third interation of Ext. Ext3 was introduced in 1999 [75] and soon became the default file system for many popular Linux distributions [74]. Being both forward and backward compatible with Ext2 is one reason that led to the success and of de adoption of Ext3 [76]. An Ext2 file system can be mounted as Ext3 and vice versa. Ext3 inherited all the great features from Ext2 and also added a journal to increase the level of data consistency. There are three levels of data integrity in Ext3. Journal mode forces both the metadata and the contents of data blocks to be written to the journal before changes are committed. This approach imposes a hit on performance because the data are written twice, however, improves reliability tremendously. Ordered mode is the default mode, which force data blocks to be written to the disk before the metadata are changed in the journal. Writeback mode only keeps metadata in the journal. This approach is fast but high risk because the metadata are committed before actual data are written to disk.

ZFS was developed by Sun in 2002 to address shortcomings of other Unix filesystems. The design of ZFS focuses on simplifying multi-disk management and detecting data error [11]. ZFS allows multi disks to join together in a virtual storage pool. A ZFS filesystem is decoupled from physical storage disks and can be expanded or shrunk on the fly. ZFS can support storage-pool up to 256 quadrillion zettabytes (ZB). In ZFS, data is stored in blocks and checksum of each block is kept in a parent indirect block (pointer block) using 64-bit Fletcher checksum [25]. Pointer blocks are also checksummed and the checksum is store in the next level pointer block. This checksum mechanism propagates all the way to the top of the filesystem. When data is written to a block, a new checksum is calculated and an update is made to the pointer block. Whenever a block is read, the checksum is calculated, then compared with the checksum on record to detect data corruption. Although modern disk drives have built-in error detection mechanisms, a fraction of errors is still gone undetected by either by the disk or the operating system. Checksum also allows ZFS to repair corrupted data automatically if ZFS is configured with redundancy (mirrored or RAID). ZFS provides a tool called scrub to check and repair corrupted data block. Unlike other tools like fsck, system administrator can run scrub in the background. There is no need to take a whole filesystem offline to run integrity check. Scrub can work on a working filesystem.

Local filesystems can provide a safe storage space for raw scientific data. Read and write data to a local filesystem is convenient and fast because it does not have to take in account of external factors such as network bandwidth and network latency. Data can be read as fast a HDD can spin. Users can use RAID [48] to to increase local filesystem storage capacity and add ability to recover from data corruption. However, to get good performance, users may only process, analyze and run scientific workloads using only local machines. In addition, sharing data is naturally difficult without a network connection.

2.2 Distributed Filesystem

In order to facilitate sharing of scientific data, scientific researchers usually employ various network filesystems such as NFS [56] or AFS [31] to provide data distribution and concurrent access.

NFS was developed by Sun Microsystems in 1983. It has gone on to become the most common distributed filesystem in the Linux/UNIX world. It was developed with four primary goals in mind [57]. First of all, NFS is machine and OS independent. The protocols in NFS are simple so that they can theoretically be implemented on any kind of machine, not just UNIX machines. Secondly, NFS is stateless. The NFS server does not maintain any state between Remote Procedure Calls (RPC). This way, if the client or server crashes, no state needs to be recovered. Thirdly, NFS is transparent. To applications, an NFS filesystem appears like a local hard drive mounted on VFS. Files can be accessed using regular pathnames, and no extra work is needed to retrieve data over the network. Lastly, NFS is designed for performance. The original idea was to make an NFS filesystem have comparable speed to a local disk on a SCSI interface.

The NFS protocol is implemented using synchronous RPCs. Essentially, the client makes a request to the kernel using the standard OS API, which then sends an RPC request to the server. The client then blocks until the server processes the request and sends back another RPC with the reply.

The server in the NFS protocol does not maintain any state regarding its interactions with the client. This means that the client must send all contextual information needed for a request each time it sends one. The reason for this is to aid in trivial crash recovery. If the server crashes, the client can simply repeat requests until it receives the data it needs. If the client crashes, the server needs to do no work at all, and the client can be in charge of its own crash recovery without having to interact with the server. In newer version of NFS, statelessness is not maintained as stringently as when it was originally implemented in order to implement things like better cache coherency protocols and file locking [35].

To maintain consistency when multiple clients are using the same file, the NFS protocol requires clients to use a method called close-to-open consistency. Basically, this means that the client must commit any changes to the file when it closes, and that when a client opens a file it must always retrieve file attributes from the server. By using this method, clients know that their data will at least be consistent between opening and closing.

The Andrew File System (AFS) is a network file system developed by Carnegie Mellon University. AFS supports various popular operating systems such as UNIX, Windows, and Mac OS. Like NFS, AFS allows users to access files across the network as if they are in a local storage device. One of advantage AFS has over NFS is security. By using Kerberos authentication, users not only can access their files, but can also share their files with other users or other groups. There are two components of AFS: Venus and Vice [62]. Venus runs on the client machine and makes requests to the AFS server on behalf of the clients. Vice runs on the server side, and serves requests from Venus. When a file is opened, Venus makes a request to fetch the whole file from the server. This local copy is called a snapshot. Any file operation such as a read or write is done on the snapshot.

Network filesytems provides an easy and convenient way for scientists to share data across the network. However they have drawbacks. First of all, network communication does not scale very well in term of data throughput. Performance of data access is bounded by the filesystem's data consistency policy, physical network bandwidth and network latency. Users who request the same file or dataset at the same times could put a strain on the file server as well the network. More importantly, systems like AFS and NFS do not provide any means to deal with hardware failure. No matter how often the system is backup, only one copy of the data is available for all users. If a fileserver goes down, all the files in the server are not accessible until the fileserver is restored.

To get scalable and fault tolerant data storage, scientists may look into distributed storage systems such as Ceph [80] or Hadoop [29]. Most of the data in these filesystems are organized into sets of directories and files along with associated metadata. Since some of these filesystems perform automatic data replication, they not only provide fault-tolerant data access but also the ability to scale the system. Therefore, in regards to the need for scalable, fault-tolerant data storage, current distributed storage systems adequately meet this requirement.

Google developed GFS to handle an enormous amount of data. GFS is designed to store very large files, which are regularly generated by the Google Search Engine. Files are divided into chunks of 64MB, similar to clusters in a traditional local filesystem. Chunks are replicated and stored in multiple *Chunkservers*. The number of data replications varies. High demand data have more replicated chunks than low demand data. A single *Masterserver* manages the GFS namespace, mapping files to chunks and enforcing file access control. Data modification is appended at the end of file rather than being overwritten. Applications issue a read request through the *Masterserver*. The *Masterserver* then passes the location of the chunk to the application. The application then accesses the chunk directly from the *Chunkserver*.

HDFS is an open source implementation of GFS by Apache Software Foundation. HDFS is a distributed filesystem written in Java. It is designed to run on commodity hardware to store big files that traditional local filesystems do not support. The HDFS architecture includes a *Namenode* and multiple *Datanodes*. The role of the *Namenode* is to manage the HDFS namespace while the *Datanodes* are in charge of storing actual data. Namenodes can be in the same Local Area Network (LAN) or they can span in multiple data centers. In HDFS, files are broken into chunks of a fixed size. The default chunk size is 64MB but it can be manually changed. For fault tolerance, data files are replicated at the chunk level. *Datanodes* communicate to the *Namenode* through a *Heartbeat* and *BlockReport* in order to maintain load-balancing [13]. HDFS employs a locality awareness read policy to improve data read performance. A read request for a chunk will be served by the same rack of Namenodes where the request originates. If the chunk is not stored in the same rack as the reader, the request will be served by the local data center before trying any remote chunk.

Where filesystems such as GFS and HDFS still fail, however, is in providing an efficient means of performing rich metadata queries. Since filesystems do not provide a direct means to perform these metadata operations, export processes usually involve a complex set of *ad hoc* scripts, which tend to be error prone, inflexible, and unreliable. More importantly, these manual searches through the data repository are also time consuming since all of the metadata in the repository must be analyzed for each export. Although some distributed systems such as Hadoop provide programming tools such as MapReduce [19] to facilitate searching through large datasets in a reliable and scalable manner, these full repository searches are still costly and time consuming since each experimental run will have to scan the repository and extract the particular data files required by the user.

Moreover, even with the presence of these programming tools, it is still not possible to dynamically organize and group subsets of the data repository based on the metadata in a persistent manner, making it difficult to export reusable snapshots of particular datasets.

2.3 Databases

When you want to query a tremendous amount of structured data, naturally you turn to a database management system (DBMS). Basic DBMS operations include inserting data, updating data, and querying for data. Users interact with the DBMS by issuing commands and expect to receive the result back shortly. Highly active commercial systems like the NYSE stock exchange or Amazon shopping website can handle thousands to millions of queries per second. A DBMS is also used to analyze data. However, with such a high volume of queries, performing analysis of tasks on a DBMS is not ideal. Data analysis can take hours or days and thus can bring the DBMS system to a crawl. Therefore, in the 1990s, the DBMS landscape saw an evolution from the local database to warehouse. For big database systems, data often is separated into two systems. The operational database hosts live data and serves insert, update, and query requests. A larger data warehouse [21] hosts archive data in snapshots. Data is periodically archived from the operational database to the data warehouse.

With the explosion of social networks in late 2000's there has been another evolution from traditional DBMS to NoSQL [36]. Unlike DMBS, NoSQL does not use SQL structure to perform queries. There is no fixed schema; data is denoted and stored in a key-value format instead of using a highly structured table. Major Internet companies like Google, Facebook and Twitter have different challenges in managing their users' data. First of all, the data does not have a strong structure. For example Facebook users' photos can be tagged and associated with any type of imaginable metadata. It is not feasible to modify the database schema when a new metadata is added.

Adding a new field to the database schema will affect every single record, which can take days to complete given the sheer amount of data these companies deal with. Secondly there is a requirement for instant gratification. When users update their status or post new photos, they expect to see the result right the way. Traditional DBMS could not maintain and provide real-time information out of large volumes of data update. A NoSQL system such as MongoDB, BaseX, SimpleDB, Apache CouchDB [26, 30, 50, 59] fit this type of workload better than a traditional relational DBMS. One of the drawbacks of NoSQL is that it has limited support to store raw data files. For example, MongoDB imposes a limit on file size of 4MB. In order to store large data objects, users will need to use GridFS [41] GridFS is system built on top of MongoDB. GridFS breaks up large files into chunks of 4MB and stitches them back together per users' requests.

Another common approach to managing scientific data in a database is to go the route of projects such as the Sloan Digital Sky Survey [66]. That is, rather than opt for a "flat file" data access pattern used in filesystems, the scientific data is collected and organized directly in a large distributed database such as MonetDB [32] or Vertica [78]. Besides providing efficient query capabilities, such systems also provide advanced data analysis tools to examine and probe the data. However, these systems remain undesirable to many scientific researchers.

The first problem with database systems is that in order to use them the data must be organized in a highly structured explicit schema. From our experience, it is rarely the case that the scientific researchers know the exact nature of their data *a priori* or what attributes are relevant or necessary. Because scientific data tends to be semi-structured rather than highly structured, this requirement of a full explicit schema imposes a barrier to the adoption of database systems and explains why most research groups opt for filesystem based storage systems which fit their organic and evolving method of data collection.

Most importantly, database systems are not ideal for scientific data repositories because they do not fit into the workflow commonly used by scientific researchers. In projects such as the Sloan Digital Sky Survey and Sequoia 2000 [65], the scientific data is directly stored in database tables and the database system is used as a data processing and analysis engine to query and search through the data. For scientific projects such as these, the recent work outlined by Stonebraker et. al [64] is a more suitable storage system for these high-structured scientific repositories.

In most fields of scientific research, however, it is not feasible or realistic to put the raw scientific data directly into the database and use the database as an execution engine. Rather, in fields such as biological computing, for instance, genome sequence data is generally stored in large flat files and analyzed using highly optimized tools such as BLAST [5] on distributed systems such as Condor [73]. Although it may be possible to stuff the genome data in a high-end database and use the database engine to execute BLAST as a UDF (user defined function), this goes against the common practices of most researchers and diverts from their normal workflow. Therefore, using a database as a scientific data repository moves the scientists away from their domains of expertise and their familiar tools to the realm of database optimization and management, which is not desirable for many scientific researchers.

Because of these limitations, traditional distributed filesystems and databases are not desirable for scientific data repositories which require both large scalable storage and efficient rich metadata operations. Although distributed systems provide robust and scalable data storage, they do not provide direct metadata querying capabilities. In contrast, databases do provide the necessary metadata querying capabilities, but fail to fit into the workflow of research scientists.

The purpose of ROARS is to address these shortcomings by constructing a hybrid system that leverages the strengths of both distributed filesystems and relational databases to provide fault-tolerant scalable data storage and efficient rich metadata manipulation. This hybrid design is similar to SDM [46] which also utilizes a database together with a file system. The design of SDM highly optimizes for n-dimensional arrays type data. Moreover, SDM uses multiple disks support high throughput I/O for MPI [22], while ROARS uses a distributed active storage cluster. Another example of a filesystem-database combination is HEDC [63]. HEDC is implemented on a single large enterprise-class machine rather than an array of storage nodes. iRODS [79] and its predecessor, the Storage Resource Broker [9], supports tagged searchable metadata implemented as a vertical schema. ROARS manages metadata with horizontal schema pointing to files and replicas which allows for the full expressiveness of SQL to be applied.

ROARS merges these two different storage concepts into a hybrid distributed storage system where the filesystem is augmented and enriched with database capabilities. Next chapters will demonstrate that ROARS is capable of handling raw data storage, metadata query, and metadata provenance gracefully. ROARS is also a stable system that provides a high level of data availability and data integrity through data replication. ROARS is robust enough to tolerate system failures and provide mechanisms for recovery, backup, and restoration. Because data is replicated and distributed intelligently across Storage Nodes, ROARS naturally fits into distributed framework such as All-Pairs [42], Makeflow [82], and Weaver [15]. Thus, ROARS enables scientific researchers to continue using their familiar workflow and applications.

CHAPTER 3

ARCHITECTURE AND IMPLEMENTATION

3.1 System Design

ROARS is designed to store millions to billions of individual objects, each typically measured in megabytes or gigabytes. Each object contains both binary data and structured metadata that describes the binary data. Because ROARS is designed for the long-term preservation of scientific data, data objects are write-once, read-many (WORM), but the associated metadata can be updated by logging. The system can be accessed with an SQL-like interface and also by a filesystem-like interface.

3.1.1 Data Model

A ROARS system stores a number of named **collections**. Each collection consists of a number of unordered **objects**. Each object consists of the two following components:

1. **Binary Data:** Each data object corresponds to a single discrete binary file that is stored on a filesystem. This object is usually an opaque file such as a `TIFF`(image), a `AVI`(video), a `WAV`(sound) or `PDF`(document), meaning that the system does not extract any information from the file other than the basic filesystem attributes such as filesize and creation date.

2. **Structured Metadata:** Associated with each data object is a set of metadata items that describes or annotates the raw data object with domain-specific properties and values. This information is stored in plain text as rows of (NAME, TYPE, VALUE, OWNER, TIME) tuples as shown in the example of a metadata record here:

```
NAME         TYPE    VALUE        OWNER  TIME
recordingid  string  nd3R22829    hbui   1253373461
filename     string  nd5M12.wav   hbui   1253373461
format       string  wav          hbui   1253373461
ingested     date    08/03/2010   hbui   1253373461
subjectid    string  nd1S04388    hbui   1253373461
comment      text    Spring Co.   hbui   1253373461
state        string  problem      dthain 1254049876
problemtype  number  34           dthain 1254049876
state        string  fixed        hbui   1254050851
```

In the above example, each tuple contains fields for NAME, TYPE and VALUE, which define the name of the object's property, the type, and its value. Currently supported types include string, number, date, and text, with no declared limits on field length. This data model is schema-free: the user does not declare any property of a collection, and an object may have any number of properties. In practice, a given collection is likely to have objects with similar metadata, so an implementation of ROARS may reasonably optimize for that case.

In addition to NAME, TYPE and VALUE fields, each metadata entry also contains fields for OWNER and TIME. This is to provide provenance information and complete history of the metadata. OWNER denotes who changed the value of the metadata while `TIME` represents the time stamp (in UNIX epoch time) when the change took place. Rather than overwriting metadata entries when a field is updated, new values are simply appended to the end of the record. In the example above, the `state` value was initially set to `problem` by one user and then later to `fixed` by another. By doing so, the latest value for a particular field will always be the last entry found in the record. This transactional metadata log is critical to scientific researchers who often need to keep track of not only the metadata, but how it is updated and transformed over time. These additional fields enable the users to track who made the updates, when the updates occurred, and what the new values are.

This data model fits in with the write-once-read-many nature of most scientific data. The discrete data files are rarely if ever updated and often contain data to be processed by highly optimized domain-specific applications. The metadata, however, may change or evolve over time and is used to organize and query the data sets.

3.1.2 User Interface

Users may interact with the system using either a command-line tool or a filesystem interface. The command line interface supports the following operations:

```
SCREEN  <coll> FROM  <dir>
IMPORT  <coll> FROM  <dir>
QUERY   <coll> WHERE <expr>
EXPORT  <coll> WHERE <expr> INTO <dir> [AS <pattern>]
VIEW    <coll> WHERE <expr> AS <pattern>
```

```
DELETE  <coll> WHERE <expr>
```

Before data is ingested into ROARS. It is recommended that users use `SCREEN` to examine data and metadata for consistency purpose. The `SCREEN` operation scans a local directory containing objects and evaluates metadata before `IMPORT` is called. The `IMPORT` operation loads a local directory containing objects and metadata into a specific collection in the repository. With `IMPORT`, ROARS creates replicas sequentially. `QUERY` retrieves the metadata for each object matching a given expression. `EXPORT` retrieves both the data and metadata for each object and stores the result on the local disk. `VIEW` creates a materialized view on the local disk of all objects matching the given expression, using the specific pattern for the path name. `DELETE` marks data objects and the corresponding metadata as deleted. The system administrator may permanently delete them if desired.

Because metadata schema can evolve overtime, it is important to make sure that the metadata `IMPORT` is about to ingest into ROARS is consistent with the current metadata schema. For each metadata attribute, the `SCREEN` checks for its name, type and length. `SCREEN` then compares this information to information from the internal metadata schema. If `SCREEN` detects any discrepancy, it notifies the user and gives an option to correct the schema. For example, `SCREEN` can warn the user that an attribute does not exist in the current schema and ask the user to expand the schema to accommodate the new attribute. `SCREEN` also examines the actual data objects and alerts the user about missing raw data.

Ordinary applications may also view ROARS as a read-only filesystem, using either FUSE [1] (a user/kernel filesystem driver) or Parrot [70] (a ptrace-based interposition agent). Individual objects and their corresponding metadata can be accessed via their unique file identifiers using absolute paths:

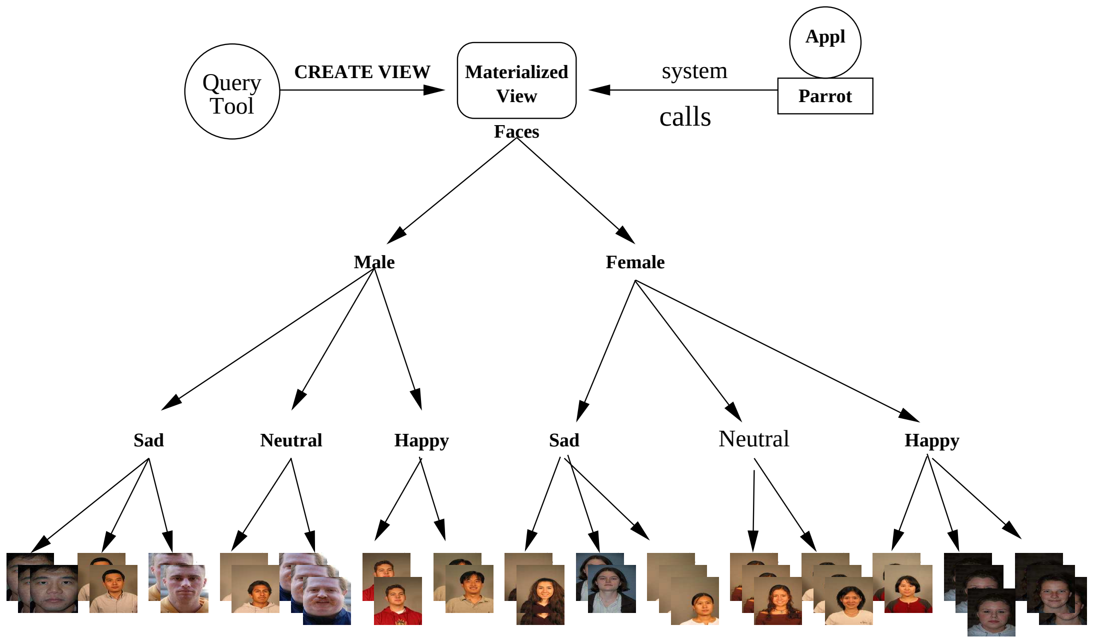


Figure 3.1. Example Materialized View

```
/roars/mdsname/fileid/3417
/roars/mdsname/fileid/3417.meta
```

Of course, it would be impractical to list and navigate a flat directory consisting of millions of files. Instead, most users find it effective to explore the repository using the `QUERY` command to retrieve metadata, then use `VIEW` to deposit a smaller materialized view for direct use. A materialized view consists of a directory tree where the leaves are symbolic links pointing to absolute paths in the repository. For example, Figure 3.1 shows a view generated by the following command:

```
VIEW faces WHERE true AS "gender/emotion/fileid.type"
```

Because the materialized view is stored in the normal local filesystem, it can be kept indefinitely, shared with other users, sent along with batch jobs, or packed up into an archive file and emailed to other users. Authentication can be obtained using a ticket system [23] that grants access to a dataset within a time period. The creating user manages their own disk space and is responsible for cleanup at the appropriate time. The ability to generate materialized views that provide third party applications a robust and scalable filesystem interface to the data objects is a distinguishing feature of ROARS. Rather than force users to integrate their domain-specific tools into a database execution engine or wrap it in a distributed programming abstraction, ROARS enables scientific researchers to continue using their familiar scripts and tools. We will continue and expand on the discussion of using ROARS' `VIEWS` in a large scale distributed application in a later chapter.

3.1.3 System Management

Management of a large storage cluster requires some care. Adding or removing storage nodes may require movement of data, which itself may be a long-running and fault-prone task. To this end, ROARS provides a management interface which separates the logical addition and removal of nodes from data migration, which can be performed at leisure. Our current implementation of ROARS includes the following management operations:

```
LIST NODES
ADD NODE     <nodename> <groupid>
REMOVE NODE  <nodename>
ABANDON NODE <nodename>
MIGRATE DATA
AUDIT DATAi
REPAIR DATA
```

`LIST NODES` queries the MDS for the list of available storage nodes, showing the current state, capacity, and usage. `ADD NODE` will add a storage node to the system and make it available as a target for newly ingested data. `REMOVE NODE` will put an existing storage node in the 'removing' state, but has no immediate effect on the data on that node. A removed node is no longer a target for `IMPORT` and is not preferred for servicing reads. In the case where a physical failure renders migration impossible, `ABANDON NODE` is used to immediately remove the corresponding node and replica records from the system. Finally, `MIGRATE DATA` queries for nodes in the *removing* state holding files with too few replicas. It incrementally migrates or replicates the data to nodes with available space as needed. When a storage node in the removed state no longer contains replicas, it is deleted from the MDS.

A major system reconfiguration – such as replacing one storage cluster with another – can be achieved by calling `ADD NODE` to configure the new nodes, `REMOVE NODE` to mark the old nodes as no longer needed, and then `MIGRATE DATA` to begin the process of moving data. Note that because `ADD NODE` and `REMOVE NODE` only interact with the MDS, it does not matter whether the server is online or offline. If `MIGRATE DATA` finds a server offline, it simply moves on to other available work.

`AUDIT DATA` is used to check the data integrity of the entire system. This command checks all servers for basic health and then queries the MDS to ensure that every file has sufficient replicas, that replicas are distributed across groups, and that there are no replicas located on removed or abandoned servers. Finally, all data objects are checksummed and compared against the value in the MDS. Checksumming is performed in parallel locally at each storage node, making it feasible to check the integrity of a very large archive in time proportional to the

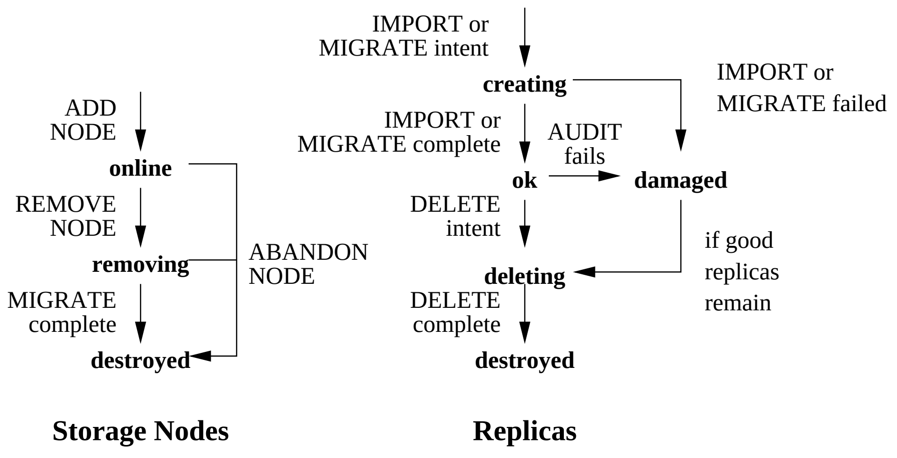


Figure 3.2. State Machines for Storage Nodes and Replicas

size of the largest storage node. If problems exist, they are reported to the MDS and, where possible, repair is done by replicating good replicas. In extreme cases where no good replicas remain, repair is not possible, and damaged replicas are left in place (and indicated as damaged) for manual examination and recovery.

The time of the last good checksum for each replica is stored in the MDS. This data has several uses: tools can focus on the oldest replicas first, management operations can be done incrementally, and the physical wear of auditing on the system can be throttled by specifying a minimum time between checksums.

3.2 Implementation

Figure 3.3 shows the basic architecture of ROARS. To support the discrete object data model and the data operations previously outlined, ROARS utilizes a hybrid approach to construct scientific data repositories. Multiple *storage nodes* are used for storing both the data **and** metadata in archival format. A *metadata server* (MDS) indexes all of the metadata on the storage server, along with the

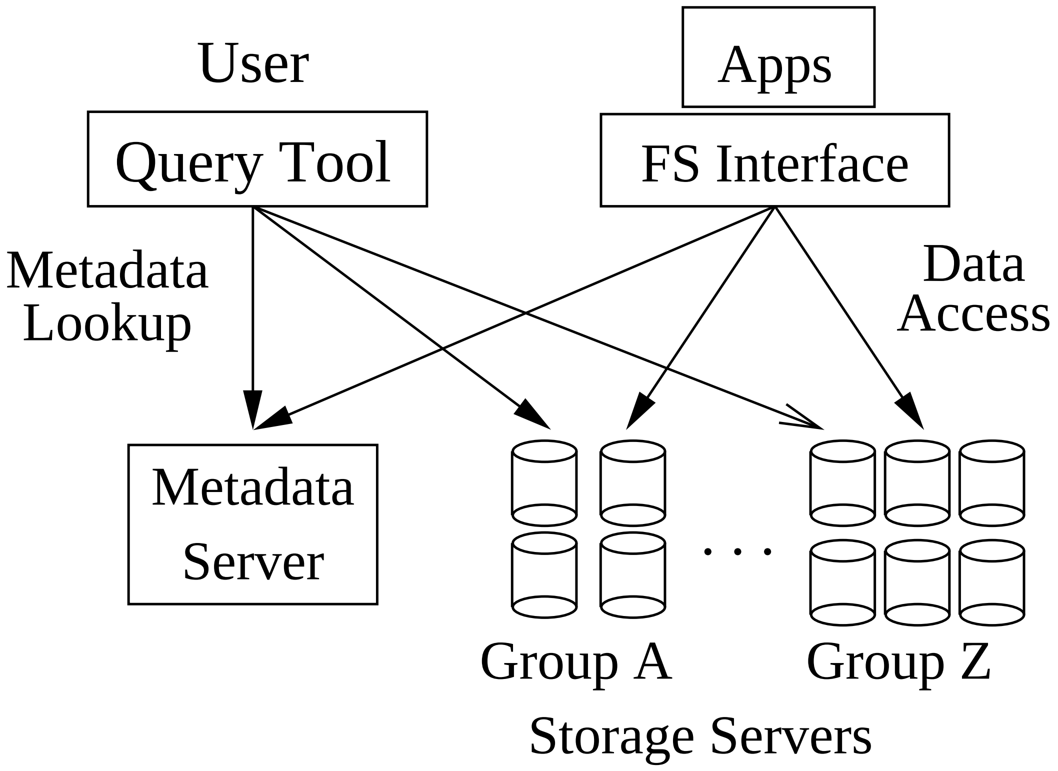


Figure 3.3. ROARS Architecture

location of each replicated object. The MDS serves as the primary name and entry point to an instance of ROARS.

The decision to employ both a database and a cluster of storage servers comes from the observation that while one type of system meets the requirements of one of the components of a scientific data repository, it is not adequate at the other type. For instance, while it is possible to record both the metadata and raw data in a database, the performance would generally be poor and difficult to scale, especially to the level required for large scale distributed experiments nor would it fit in with the workflow normally used by research scientists. Moreover, the distinct advantage of using a database, its transactional nature, is hardly utilized in a scientific repository because the data is mostly write-once-read-many, and thus rarely needs atomic updating. From our experience, during the lifetime of the repository, metadata may changed once or twice while the raw data stays untouched. Besides the scalability disadvantages, keeping raw data in a database poses bigger challenges on everyday maintenance and failure recovery. So although, a database would provide good metadata querying capabilities, it would not be able to satisfy the requirement for large scale data storage.

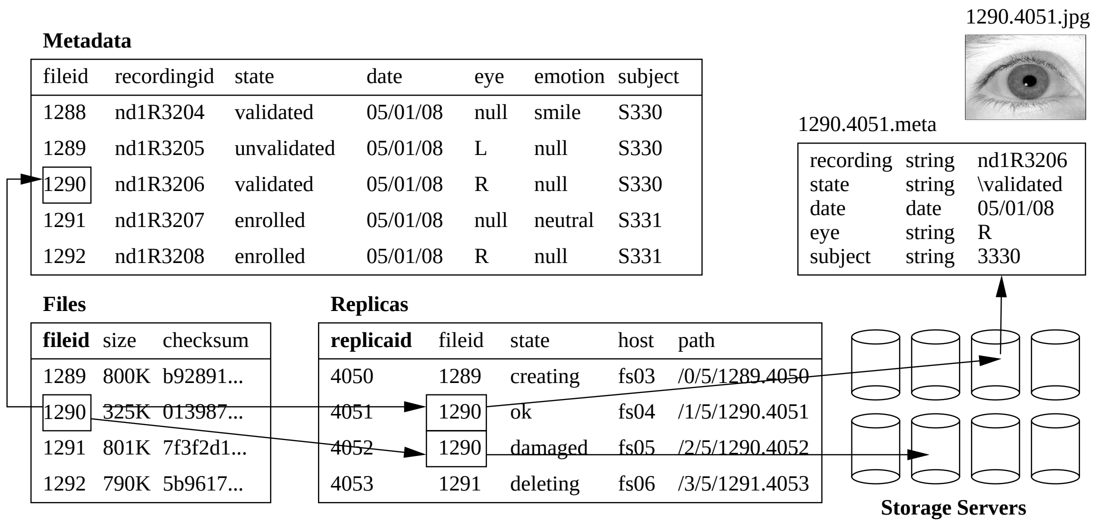
On the other hand, a distributed storage system, even with a clever file naming scheme, is also not adequate for scientific repositories. Such distributed storage systems provide scalable high performance I/O but provide limited support for rich metadata operations. Metadata operations generally devolve into full dataset scans or searches using fragile and *ad hoc* scripts. Although there are possible tricks and techniques for improving metadata availability in the filesystem, these all fall short of the efficiency required for a scientific repository. For instance, while it is possible to encode particular attributes in the file name, it is still inflexible and inefficient, particularly for data that belong to many different categories. Fast access to metadata remains nearly impossible, because parsing thousands or millions filenames is the same if not worse than writing a cumbersome script to parse collections of metadata text files.

The hybrid design of ROARS takes the best aspects from both databases and distributed filesystems and combines them to provide rich metadata capabilities and robust scalable storage. To meet the storage requirement, ROARS replicates the data objects along with their associated metadata across multiple storage nodes. Like in traditional distributed systems, this use of data replications allows for scalable streaming read access and fault tolerance. In order to provide fast metadata query operations, the metadata information is persistently cached upon importing the data objects into the repository in a traditional database server. Queries and operations on the data objects access this cache for fast and efficient storage operations and metadata operations.

Although, ROARS storage organization is similar to the one used in the Google Fileystem [27], and Hadoop [29], where simple Data Nodes store raw data and a single Name Node maintains the metadata. ROARS architecture differs in a few important ways however. First, rather than striping the data as blocks across multiple storage nodes as done in Hadoop and the Google Filesystem, ROARS store discrete whole data files on the storage nodes. While this prevents us from being able to support extremely large file sizes, this is not an important feature since most scientific data collections tend to be many small files, rather than a few extremely large ones. Moreover, the use of whole data files greatly simplifies recovery and enables failure independence. Likewise, the use of a database server as the metadata cache enables us to provide sophisticated and efficient metadata queries. While Google Filesystem and Hadoop are restricted to basic filesystem type metadata, ROARS can handle queries that work on constraints on domain-specific metadata information, allowing researchers to search and organize their data in terms familiar to their research focus.

3.2.1 MDS Structure

We employ a relational database to implement the main functionality of the MDS. The database contains three primary tables: a metadata table, a file table and a replica table. The metadata table stores the most recent values for all items in a collection, indexed for efficient lookup. Each entry in the metadata table



Metadata

fileid	recordingid	state	date	eye	emotion	subject
1288	nd1R3204	validated	05/01/08	null	smile	S330
1289	nd1R3205	unvalidated	05/01/08	L	null	S330
1290	nd1R3206	validated	05/01/08	R	null	S330
1291	nd1R3207	enrolled	05/01/08	null	neutral	S331
1292	nd1R3208	enrolled	05/01/08	R	null	S331

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	/0/5/1289.4050
4051	1290	ok	fs04	/1/5/1290.4051
4052	1290	damaged	fs05	/2/5/1290.4052
4053	1291	deleting	fs06	/3/5/1291.4053

1290.4051.meta

recording	string	nd1R3206
state	string	\validated
date	date	05/01/08
eye	string	R
subject	string	3330

Figure 3.4. MDS Structure

points to a unique `fileid` in the file table. The file table plays the same role an inode table in a traditional Unix file system does for ROARS and holds the essential information about raw data files, such as `size`, `checksum`, and `create time`. ROARS utilizes this information not only to keep track of files but also to emulate system calls such as `stat`. For any given `fileid`, there can be multiple replica entries in the replica table, which tracks the location and state of each replica of a file. Figure 3.4 gives an example of the relationship between the metadata, file, and replica tables. In this configuration, each file is given a unique `fileid` in file table. In the replica table, the `fileid` may occur multiple times, with each row representing a separate replica location in the storage cluster. Accessing a file then involves looking up the `fileid`, finding the set of associated replica locations, and then selecting a storage node.

As can be seen, this database organization provides both the ability to query files based on domain specific metadata and the ability to provide scalable data distribution and fault-tolerant operation through the use of replicas. Some of the additional fields such as `lastcheck`, `state`, and `checksum` are used by high level data access operations provided by ROARS to maintain the integrity of the system. These operations will be discussed in later subsections.

A few complications regarding the metadata are worth noting. First, the ROARS abstract data model has no schema nor limits on field length. ROARS maps all metadata fields to a relational database table. ROARS supports new fields by adding new columns or expanding field widths as needed. This could be highly inefficient for very sparse data, but is adequate for the common case where items in a collection share a number of properties.

Second, the metadata table only contains the most recent values for each tuple in a record. The complete history including the OWNER and TIME elements described in section 2.1 is stored in a distinct *metadata log* table. Additionally, these information is written to the metadata file next to each raw data object in the storage nodes. This provides the complete history of the repository when it is necessary to audit for scientific integrity. Each metadata change is written to the database intermediately. However, the change may not be reflected at the storage node simultaneously. ROARS could write changes to both database and storage servers atomically; however, because of the latency discrepancy between a database update transaction and a disk write transaction, writing changes to both is lagged and bounded by slow disk speed. Especially when there are mass metadata changes during the enrollment process, writing thousands of small transactions to disk can take minutes to hours.

Third, any changes to metadata must be reflected in several places: the metadata table, the metadata log, and each of the distributed metadata files. This is accomplished by treating the metadata log as a roll-forward recovery log. All updates are applied to the log first and marked as 'incomplete' until they are appended to each of the distributed metadata files.

3.2.2 Storage Nodes

ROARS uses the Chirp [72] user level filesystem to implement the software component of each storage node. A storage node is typically a conventional server with large local disks, organized into a cluster. Storage nodes are divided into different storage groups based on locality, and given a `groupid`. This approach is consistent with the *structure* principle that Maccormic et al. proposed with the Kinesis system [37]. In such a system, storage servers are grouped into different segments which are likely to be failure-independent. Thus, failure in one segment would not catastrophically affect the system as a whole. An `IMPORT` deliberately places replicas in different storage groups to achieve both load balancing and failure independence. This approach is similar to *rack awareness* in Hadoop. By convention, if a data object was named `X.jpg`, then the associated metadata file would be named `X.meta` and both of these files are replicated across the storage nodes in each of the Storage Groups.

By replicating the raw data across the network, ROARS provides scalable, high throughput data access for distributed applications. Moreover, because each storage group has at least one copy of the data file, distributed applications can easily take advantage of data locality with ROARS. Applications using the filesystem interface are directed to the closest replica, preferring one on the same node, otherwise in the same storage group if possible.

3.3 ROARS as a Storage System for Daily Use and Long Term Data Preservation

3.3.1 Robustness

The ROARS architecture is robust to a wide variety of failures in a number of dimensions, including server or network outage, server loss, data corruption, and interruption of write and administrative operations. The system design assumes that errors are due to failures or accidents, but does not go to the expense of protecting against Byzantine failures, as in in LOCKSS [55].

Data integrity is achieved by checksumming all file objects on storage nodes and by recording this in the MDS. (Integrity of the MDS can be accomplished via internal hierarchical checksums of the tables, as in ZFS [12]. Data integrity is verified by a periodic `AUDIT` process as described above. Damaged replicas are automatically deleted if a majority of replicas are in agreement with the checksum stored in the MDS. If a majority of replicas agree upon a checksum, but this does not equal that stored in the MDS, ROARS assumes the MDS is corrupted, and with manual approval, will update the MDS to correspond to the majority view.

Replication is the primary defense against server and network outage. Files are replicated three times by default. During a read operation by the query client or the filesystem client, if a replica is not reachable due to server outage or hardware failure, ROARS will randomly try other replicas until a user-specified timeout is reached. As mentioned earlier, storage nodes are organized into groups based on locality. By placing replicas in distinct groups, the likelihood of availability is improved.

ROARS employs complete replicas of each data object and corresponding metadata file to protect against server loss. In the event of multiple simultaneous hardware failures from a fire, flood, etc., any individual storage unit that can be recovered contains a usable fragment of the data and its corresponding metadata. If the metadata server itself is lost, the entire contents of the metadata table, metadata log, file table, and replica table can be reconstructed from the metadata files on the distributed storage nodes, albeit at some expense. From this perspective, the MDS is an important cache of the metadata, but not the authoritative copy.

All operations that write to the archive, including `IMPORT`, `MIGRATE DATA`, and `AUDIT DATA` are carefully designed to move file servers and replicas through the state transitions shown in Figure 3.2. The common concept is that major actions are accomplished via two phase commit: a state transition marks an intent to execute and the intention is executed before completing the next action. If an administrative command crashes or is forcibly killed, the next invocation observes the previously recorded intent, and continues. This makes all operations robust to system failures or accidental cancellation. It also give the system operator flexibility to spread out long-running operations. For example, one might accomplish a complex migration by running it incrementally for two hours every night during a period of low usage. Additionally, both the command line client and the filesystem client take server and replica states into account, so that imports and reads can continue while the system is in flux.

3.3.2 Reliability and Availability

ROARS must be able to preserve data and support data recovery when disaster strikes. First, it must be *reliable*: once imported, data in the system should survive the expected rate of hardware failures and migrate automatically as new hardware is provisioned. Second, it must also be *available*. Acquisition of data occurs on

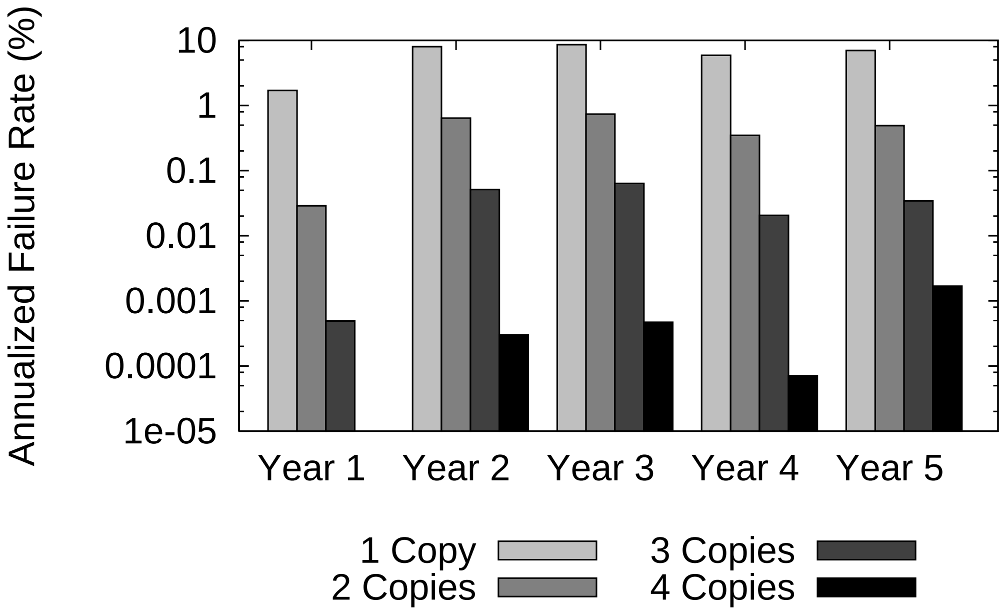


Figure 3.5. Expected Failure Rate for Replicated Data

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.

Figure 3.5 shows the expected probability of data loss due to disk failure based on the values observed by Google [49], 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, ROARS should plan to provision new equipment

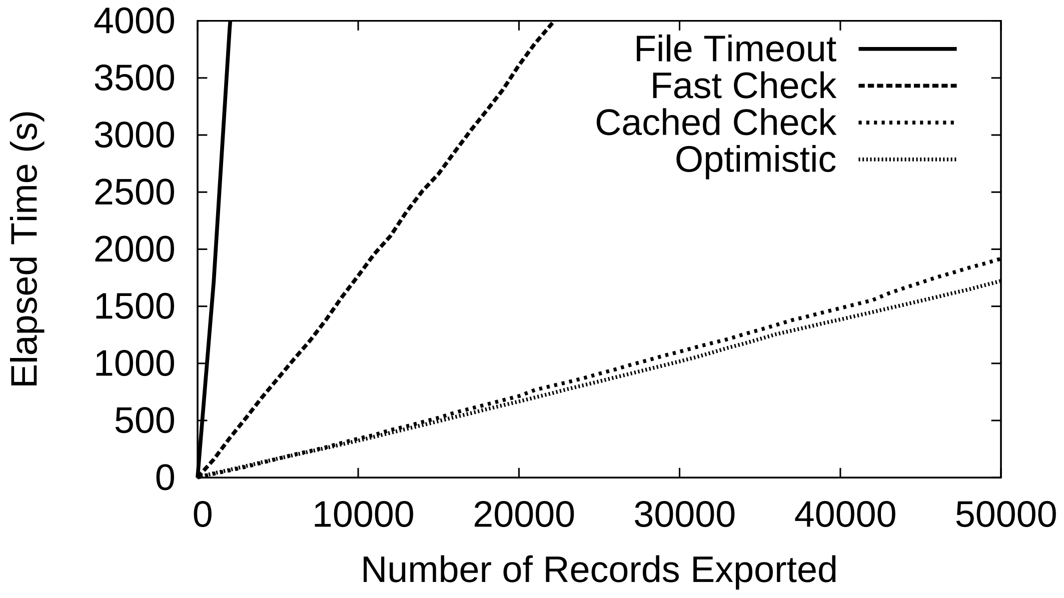


Figure 3.6. Performance of Transparent Failover Techniques

3.3.3 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 the entire workload. To avoid this problem, ROARS probes for server health using an inexpensive test 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 3.6 demonstrates this by comparing the performance of several variations of transparent failover while exporting 50,000 data objects, 300KB each. The "Optimistic" case has all 16 servers operating and downloading 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 an additional check.

3.3.4 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.

3.3.5 Asynchronous Audit and Repair

An important aspect of preserving data for the long haul is providing the end user with an independent mean 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, ROARS was bulit to 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 is 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 all active storage units.

When the repository is scaled up to a million recordings, 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` check for corrupted replicas, repairs the system by making new replicas and deleting bad copies.

3.3.6 Active Storage

ROARS is also capable of executing programs internally, co-locating the computation with the data that it requires. This technique is known as *active storage* [54]. In ROARS, an active storage job is dispatch to a specific file server containing the input files where it is run in an *identity box* [68] to prevent it from harming the archive.

Active storage is frequently used in ROARS to provide transcoding from one data format to another. For example, a large MPEG format animation might be converted down to a 10-frame low resolution GIF animation to use as a preview image on a web site. A given web page might show tens or hundreds of thumbnails that must be transcoded and displayed simultaneously. With active storage, we can harness the parallelism of the cluster to deliver the result faster.

Table 3.3.6 shows the performance of transcoding various kinds of images using the active storage facilites of ROARS. Each line shows the turnaround time (in seconds) to convert 50 images of the given type. The 'Local' line shows the time to complete the conversions sequentially using ROARS as an ordinary file system. The 'Remote' lines show the turnaround time using the indicated number of active storage servers. As can be seen the active storage facility does not help when applied to small still images, but offers a significant speedup when applied to large videos with significant processing cost.

TABLE 3.1

TRANSCODING IN ACTIVE STORAGE

	Iris Still	**Face Still**	**Iris Video**	**Face Video**
Method	(300KB)	(1MB)	(5MB)	(50MB)
Local	10	18	106	187
Remote x2	80	45	150	134
Remote x4	23	26	57	79
Remote x8	22	16	58	70
Remote x16	12	12	18	33
Remote x32	12	17	16	17

CHAPTER 4

SYSTEM EVALUATION

4.1 System Environment

To evaluate the performance and operational characteristics of ROARS, we ran a number of experiments on a testbed cluster consisting of 22 data storage nodes. They are servers with 32GB RAM, twelve 2TB SATA disks and two 8-core Intel Xeon E5620 CPUs, all connected via a dedicated Gigabit Ethernet switch. ROARS was deployed with 22 Chirp servers running on those 22 nodes, and an MDS with the MySQL database on a head node. A number read and write experiments were also ran on Hadoop filesystem and the results were compare to ROARS' performance results. Hadoop filesystem was deployed with 22 Hadoop filesystem Datanodes running on the same 22 data storage nodes and the Hadoop filesystem Namenode running on the cluster head node. With Hadoop filesystem, we kept the usual Hadoop defaults such as employing a 64 MB chunk size and a replication factor of three.

The following experimental results test the performance of ROARS and demonstrate its capabilities while performing a variety of storage system activities such as importing data, exporting materialized views and migrating replicas. These experiments also include micro-benchmarks of traditional filesystem operations to determine the latency of common system calls, and concurrent access benchmarks that demonstrate how well the system scales in read performance throughput. For these latter performance tests, we compare ROARS's performance to Hadoop, which is an often cited as an alternative to distributed data archiving. At the end, we include operational results that demonstrate the operational robustness of ROARS.

4.2 Basic Data Storage Operations

The purpose of these benchmark experiments is to measure ROARS' performance on daily operations of a storage system. These operations include `SCREEN`: exam data and metadata before ingesting into the system , `IMPORT`: ingest data into ROARS, `EXPORT`: get data and metadata out of ROARS, `VIEWS`: create a materialized view with metadata, `QUERY`: only get metadata, and `DELETE`: remove data from ROARS. Of the six operations, `IMPORT`, `EXPORT`, and `DELETE` interact directly with the storage nodes, while `SCREEN`, `VIEW`, and `QUERY` do not.

Figure 4.1 shows the runtime of each of the key operations on 10,000 data objects total of 17.4GB data, with triple replication. Most operations require multiple transactions against the database and the storage cluster. `IMPORT` operates at the lowest speed because it has to create three replicas for each data file. Moreover, the number of database queries is by far the most comparing to other operations. Table 4.2 shows the number of database queries for each operations per data file.

UPDATE queries are extra expensive comparing to other types of query because of ROARS' metadata logging mechanism described in Chapter 2. Each update essentially means another insert into the log table which decreases the performance of `IMPORT` and `DELETE` operations. `SCREEN` is significantly faster than

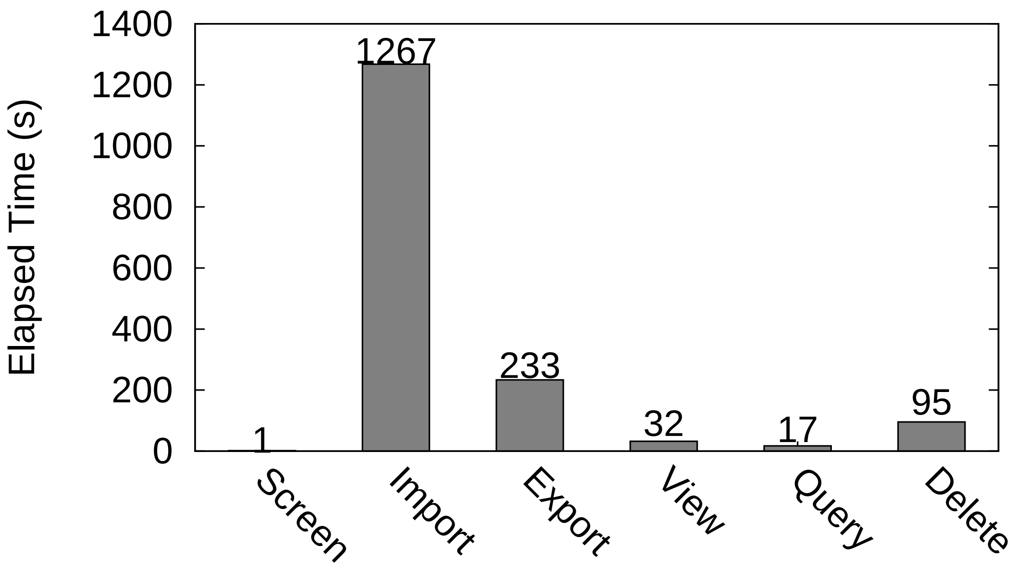


Figure 4.1. Screen, Import, Export, View, Query, Delete Performance
10,000 files 17.9GB of data

all other commands because it has the least number of database query and its merely stat data files on local storage instead of transferring data files across the network multiple times sequentially like `EXPORT`. `QUERY` is also fast because it only needs to query the metadata from the database and it does not actually fetch any data file. `VIEW` is very similar to `QUERY` because it does not interact with the storage nodes. However `VIEWS` creates symbolic link files on local hard drive for each object, thus the performance differs from `QUERY` Figure 4.2 shows the runtime of the same set of operations, this time with 10,000 small data objects of about 300KB each. As expected, the performance of `SCREEN`, `QUERY`, and `DELETE` does not depend on the total size of the data, their performance only depends on the number of objects. `QUERY` took longer in this experiment because the number of metadata for this dataset is almost double the number of metadata for the first

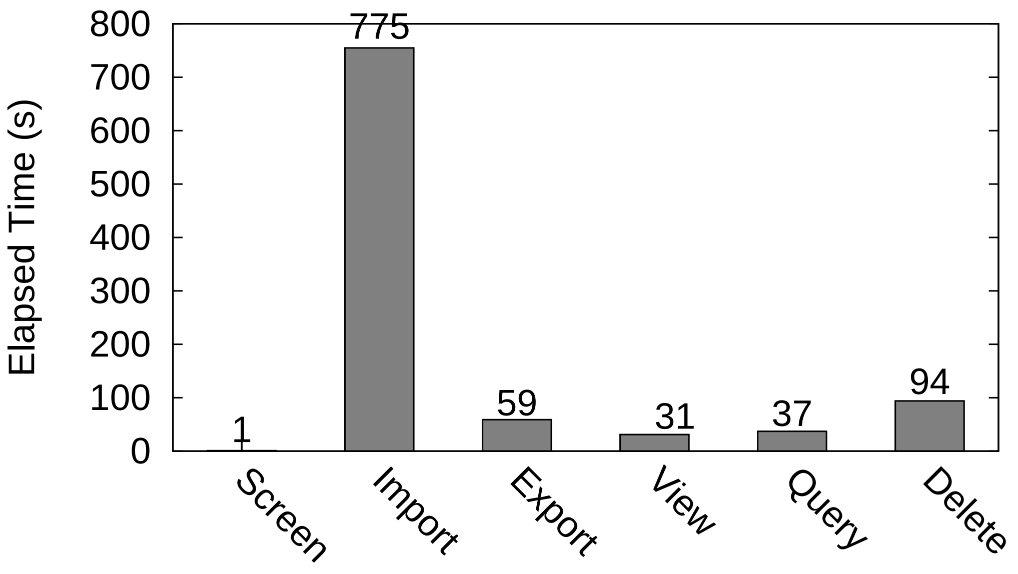


Figure 4.2. Screen, Import, Export, View, Query, Delete Performance, 10,000 files 3.0GB of data

experiment. The runtime of `SCREEN` and `DELETE` are almost identical for both experiments. The experiment results show that ROARS achieve good performance for daily use of data ingestion and data export.

4.3 ROARS vs. Hadoop: Data Import and Metadata Query

These set of experiments were designed to measure the performance of data import and metadata query for both ROARS and Hadoop. Since ROARS and Hadoop filesystem replicate data differently, the data import performance are not expected to be identical. As described in Chapter 2, ROARS replicate data in a sequential manner while Haddop replicates data in a parallel tree structure. On another hand, ROARS supports for metadata query through database, and Haddop does not have any built-in support for metadata thus users usually rely

TABLE 4.1

NUMBER OF DATABASE QUERIES PER OPERATION

QUERY	**SCREEN**	**IMPORT**	**EXPORT**	**VIEW**	**QUERY**	**DELETE**
SELECT	1	1	1	1	1	1
INSERT	0	5	0	0	0	0
UPDATE	0	7	0	0	0	5
DELETE	0	0	0	0	0	5

on Map Reduce for any type of metadata query.

4.3.1 Data Import

The following test measured the performance of importing large datasets into both Hadoop and ROARS. For this data import experiment, the test were divided into several sets of data objects. Each set consists of number of fixed size files, ranging from 1KB to 1GB. To perform the experiment, we imported the data from a local disk to the distributed systems. In the case of Hadoop this simply involved copying the data from the local machine to Hadoop filesystem. For ROARS, we used the `IMPORT` operation.

Figure 4.3 shows the data import performance for Hadoop and ROARS for several sets of data. The graph shows the throughput as the file sizes increase. The maximum theoretical throughput on a gigabit link is 128MB/s, and the maximum achievable by a TCP connection is closer to 100MB/s, depending on the variant used. Both ROARS and Hadoop filesystem achieve significantly less than that, due to the overheads of interacting with the central metadata server, and the creation

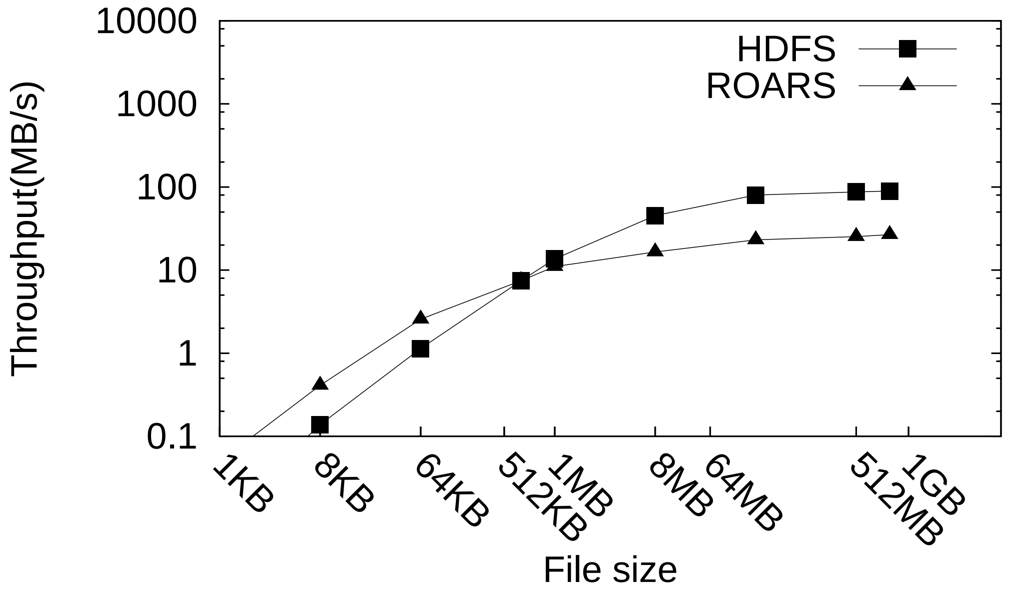


Figure 4.3. Import Performance for ROARS and Hadoop filesystem

of multiple file replicas. The overhead of interacting with the MDS is higher in ROARS, due to the multiple state transitions shown in Figure 3.2. The overhead of creating replicas is higher in ROARS, because it transfers and verifies each replica separately, whereas Hadoop filesystem sets up a data forwarding pipeline, and only communicates with the primary replica. In both systems, higher throughput is achieved with larger files. For the purposes of long term data preservation with a write-once, read-many model, this is an acceptable trade-off to achieve high integrity transactions.

4.3.2 Metadata Query

In this benchmark, we studied the cost of performing a metadata query. As previously noted, one of the advantages of ROARS over distributed systems such

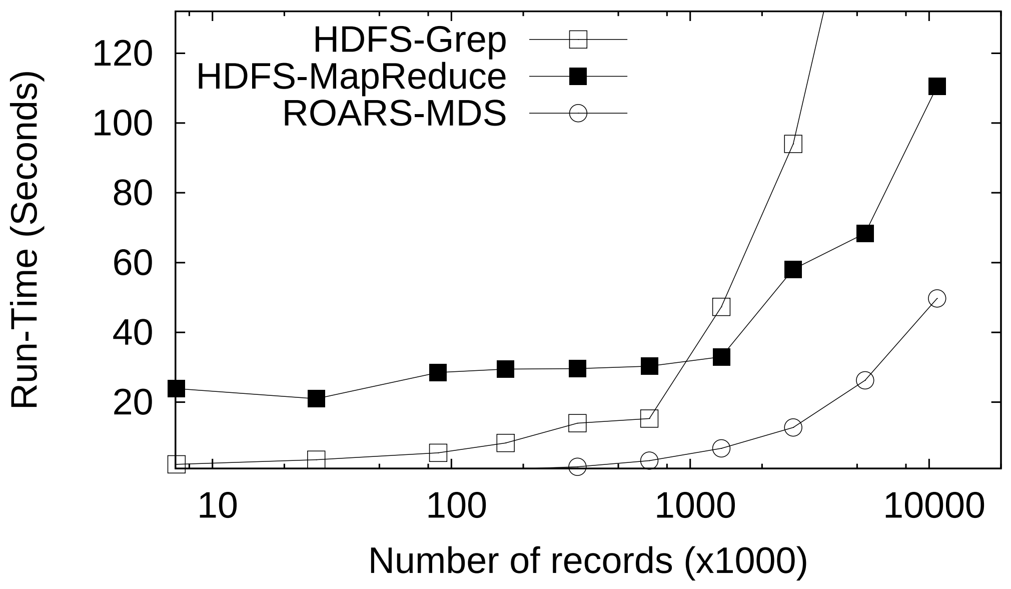


Figure 4.4. Query Performance

as Hadoop is that it provides a means of quickly searching and manipulating the metadata in the repository. For this experiment, we created multiple metadata databases of increasing size and performed a query that looks for objects of a particular type.

As a baseline reference, we performed a custom *grep* of the database records on a single node accessing Hadoop filesystem, which is normally what happens in rudimentary scientific data collections. For Hadoop, we stored all the metadata in a single file in `NAME`, `TYPE`, `VALUE` tuple format we described in Section 2.1. For each object, the metadata takes up approximately 1.3KB in storage. We started with We queried the metadata by executing the custom script using MapReduce [19]. For ROARS, we queried the metadata using `QUERY` which internally uses the MySQL execution engine.

Figure 4.4 clearly shows that ROARS takes full advantage of the database query capabilities properly and is much faster than either MapReduce or standard *grepping*. Evidently, as the metadata database increases in size, the *grep* performance degrades quickly. The same is true for the `QUERY` operation. Hadoop, however, at first retains a steady running time, regardless of the size of the database (up to 2.7M rows or 3.6 GB). After that, Hadoop Map Reduce runtime only increases linearly. This is because the MapReduce version was able to take advantage of multiple compute nodes and thus scale up its performance. Unfortunately, due to the overhead incurred in setting up the computation and organizing the MapReduce execution, the Hadoop query had a high startup cost and thus was slower than the MDS. Futhermore, the standard *grep* and MDS queries were performed on a single node, and thus did not benefit from scaling. That said, the ROARS query was still faster than Hadoop, even when the database reached 345M data objects(427 GB of data).

4.4 ROARS Data Read Performance

4.4.1 Filesystem Access

ROARS provides a read-only filesystem interface for conventional applications, consisting primarily of the system calls `stat`, `open`, `read`, and `close`. Hadoop filesystem provides similar functionality through the library `libhdfs`. To evaluate these side by side, we implemented equivalent modules in Parrot for a single Chirp server, ROARS, and Hadoop filesystem. In addition, we provide a variant of the ROARS module which caches recently used filesystem data and metadata. To test the latency of these common filesystem functions, we constructed a simple benchmark which performs repeated `stat`s, `open`s, `read`s, and `close`s on a single file.

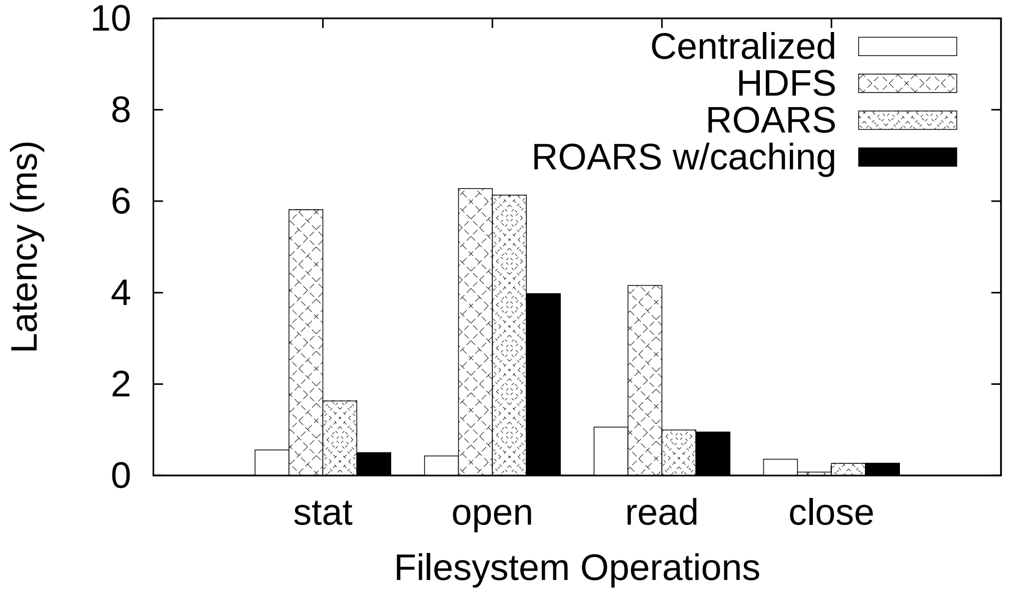


Figure 4.5. Latency of Filesystem Operations

Figure 4.5 shows the latency of each operations on a centralized server, Hadoop filesystem, and ROARS. As can be seen, ROARS provides comparable latency to the centralized server, and in the case of `stat`, `open`, and `read`, lower latency than Hadoop filesystem. Since all file access also pass through Parrot, there is some interposition overhead for each system call. However, since all of the storage systems were accessed though the same Parrot adapter, this additional overhead is same for all of the systems and thus does not affect the relative latencies.

These results show that there is overhead to communicating with the MDS for metadata, the latencies provided by the ROARS system remain comparable to Hadoop filesystem. Moreover, because of the write-once nature of the data, these queries can be cached for significant performance gains. With this small optimization, operations such as `stat` and `open` are significantly faster with ROARS than with Hadoop filesystem.

4.4.2 ROARS Concurrent Access

To determine the scalability of ROARS in comparison to a centralized network server (running Chirp) and Hadoop filesystem, we exported two different datasets to each of the systems and performed a test that read all of the data in each set. In the case of ROARS, we used a materialized view with symbolic links to take advantage of the data replication features of the system, while for the centralized network server and Hadoop filesystem, we exported the data directory to each of those systems. We ran our test program using the Condor [73] distributed batch system running on the same cluster with 1 - 32 concurrent jobs reading data from each system.

We set up the experiment using a 32 storage nodes with the following setup. Nodes are commodity servers with dual-core Intel 2.4 GHz CPUs, 4GB of RAM, and 750GB SATA-II disks, all connected via a dedicated Gigabit Ethernet switch. ROARS was deployed with 32 Chirp servers running on the 32 cluster data nodes, and an MDS with the MySQL database on the cluster head node. Hadoop filesystem was deployed with 32 Hadoop filesystem Datanodes running on the 32 cluster data nodes, and the Hadoop filesystem Namenode running on the cluster head node.

Figure 4.6 and Figure 6.2.2 show the performance results of all three systems for both datasets. In Figure 4.6, the clients read 10,000 320KB files, while in Figure 6.2.2 1,000 5MB files were read. In both graphs, the overall aggregate throughput for both Hadoop filesystem and ROARS increases with an increasing number of concurrent clients, while the traditional file server levels off after around 8 clients. This is because the central file server is limited to a maximum upload rate of about 120MB/s, which it reaches after 8 concurrent readers. ROARS

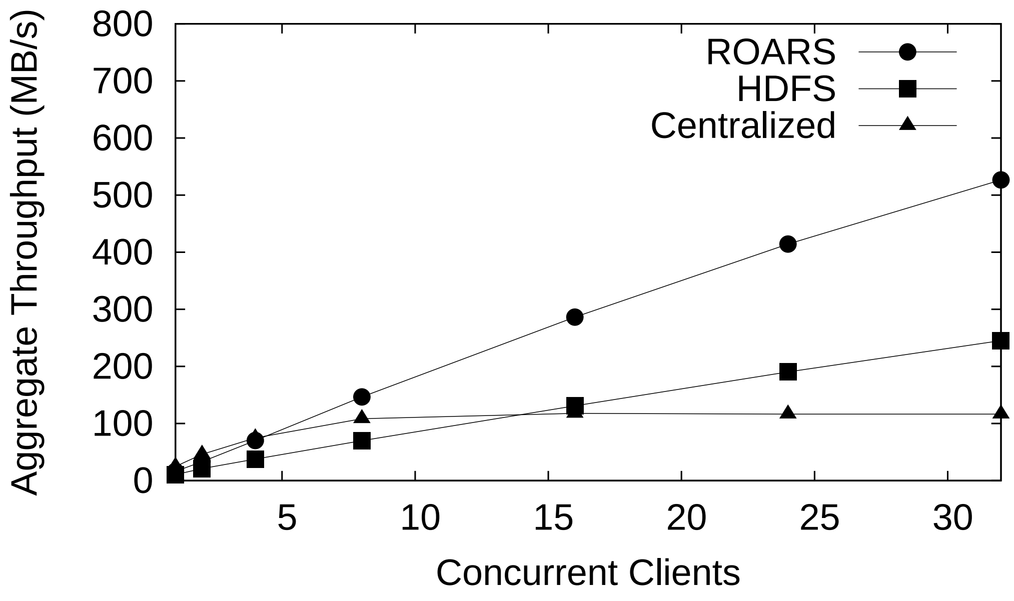


Figure 4.6. Concurrent Access Performance (10K x 320KB)

and Hadoop filesystem, however, use replicas to enable reading from multiple machines, and thus scale with the number of readers. As with the case of importing data, these read tests also show that accessing larger files is much more efficient in both ROARS and Hadoop filesystem than working on smaller files.

While both ROARS and Hadoop filesystem achieve improved aggregate performance over the traditional file server, ROARS outperforms Hadoop filesystem by a factor of 2. In the case of the small files, ROARS was able to achieve an aggregate throughput of 526.66 MB/s, while Hadoop filesystem only reached 245.23 MB/s. For the larger test, ROARS reached 1030.94 MBS/s and Hadoop filesystem 581.88 MB/s. There are several reasons for this difference. First, ROARS has less overhead in setting up the data transfers than Hadoop filesystem as indicated in the micro-operations benchmarks. Such overhead limits the number of concurrent

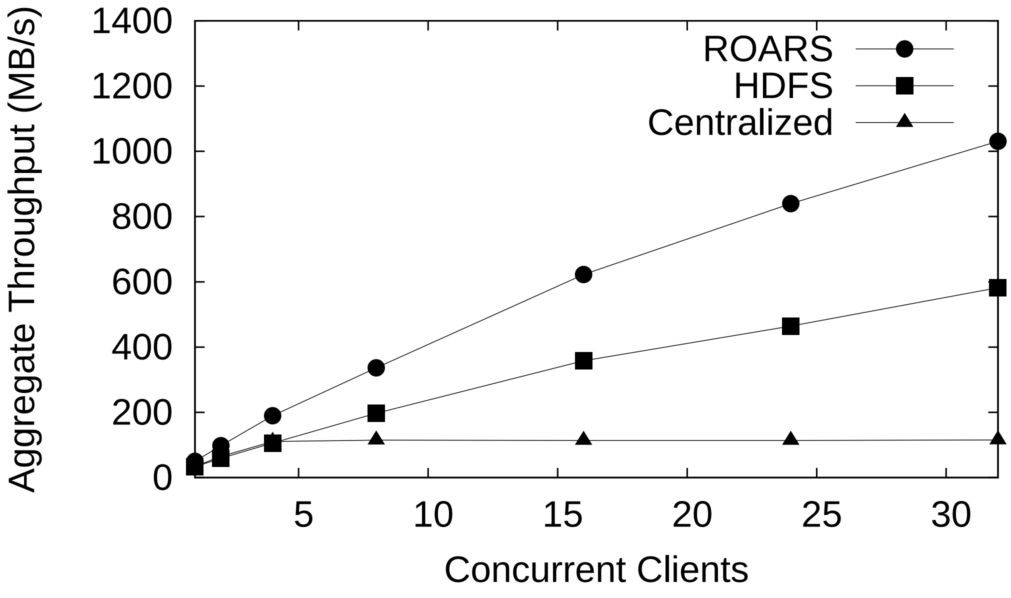


Figure 4.7. Concurrent Access Performance (1K x 5MB)

data transfers and thus aggregate throughput. Another cause for the performance difference is the behavior of the storage nodes. In Hadoop filesystem, each block is checksummed and there is some additional overhead to maintain data integrity, while in ROARS, data integrity is only enforced during high level operations such as `IMPORT`, `MIGRATE`, and `AUDIT`. Since the storage nodes in ROARS are simple network file servers, no checksumming is performed during a read operation, while in Hadoop filesystem data integrity is enforced throughout, even during reads.

4.5 Integrity Check & Recovery

In ROARS, the `AUDIT` command is used to perform an integrity check. As we have mentioned, the file table keeps records of a data file's size, checksum, and the last checked date. `AUDIT` uses this information to detect suspect replicas and

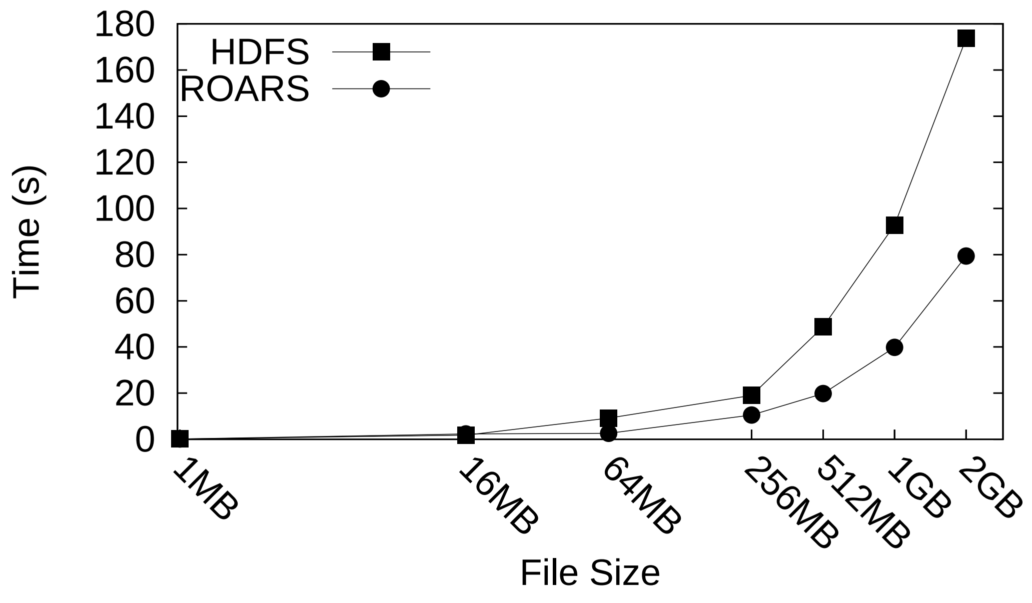


Figure 4.8. Cost of Calculating Checksums

replace them. At the lowest level, `AUDIT` checks the size of the replicas to make sure it is the same as the file table entries indicate. This type of check is not expensive to perform, but it is also not reliable. A replica could have a number of bytes modified, but remains the same size. A better way to check a replica's integrity is to compute the checksum of the replica, and compare it to the value in file table. This is expensive because the process will need to read in the whole replica to compute the checksum.

Figure 4.8 shows the cost of computing checksums in both ROARS and Hadoop filesystem. As file size increases, the time required to perform a checksum also increases for both systems. However, when the file size is bigger than a Hadoop filesystem block size (64MB), ROARS begins to outperform Hadoop filesystem because the latter incurs additional overhead in selecting a new block and setting up a new transaction. Moreover, ROARS lets storage nodes perform checksum remotely where the data file is stored while for Hadoop filesystem this data must be streamed locally before an operation can be performed.

Verifying data integrity is an essential component of maintaining a long-term archive with many stakeholders. If verification requires moving all data to an external party, then it can only be done in time proportional to the sum of the archive. To make this process feasible on a regular basis, ROARS uses the active storage facility to run the checksums directly on each storage node, then runs each storage node in parallel. In this way, a complete system audit can be performed in time proportional to the capacity of the largest node.

To start, we measure the performance of each audit method on 50,000 images object of about 300KB each. Table 5.4 show the initial result.

We also compared the performance of an external sequential audit against a parallel/active-storage audit on a production ROARS deployment of 90,000 files totalling 497GB. The sequential implementation completed in **4.2 hours**, averaging 32.5MB of data verified per second. The parallel implementation completed in **19.6 min**, for a speedup of 13x, which is imperfect due to the Amdahl overhead of the MDS operations, but still significantly faster. If we consider much larger storage systems, say 100 storage nodes of 1 TB each, a sequential integrity check would take months and be practically infeasible, while a complete parallel check could be scheduled into system downtime and completed in hours.

We measure the performance of each audit method on 50,000 images object of about 300KB each:

TABLE 4.2

AUDIT RUNTIME FOR 50,000 DATA OBJECTS, 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

4.6 Metadata Logging

Metadata can be modified through out the life cycle of a data object, starts with data import, continues with data update and ends with data delete. At each phase, metadata change is written to the database intermediately. However, those changes have not been reflected at the storage level yet. ROARS could write changes to both database and storage servers at the same time. However, because of the time discrepancy between a database update transaction and a disk write transaction, writing changes to both is lagged and bounded by slow disk speed. Especially when there are mass metadata changes during the enrollment process, writing thousands of small transactions to disk can take minutes to hours.

In order to maintain data consistency, ROARS logs all metadata changes in a log table. The log table keeps track of what has been changed, who made the change, and when the change was made. However, logging changes come with a cost. Figure 4.9 graphs shows the average performance import,update and delete operations on 100,000 metadata records with and without metadata logging. Obviously, metadata logging feature does affect import, update and delete metadata performance. Each update of metadata will result to at least one insert

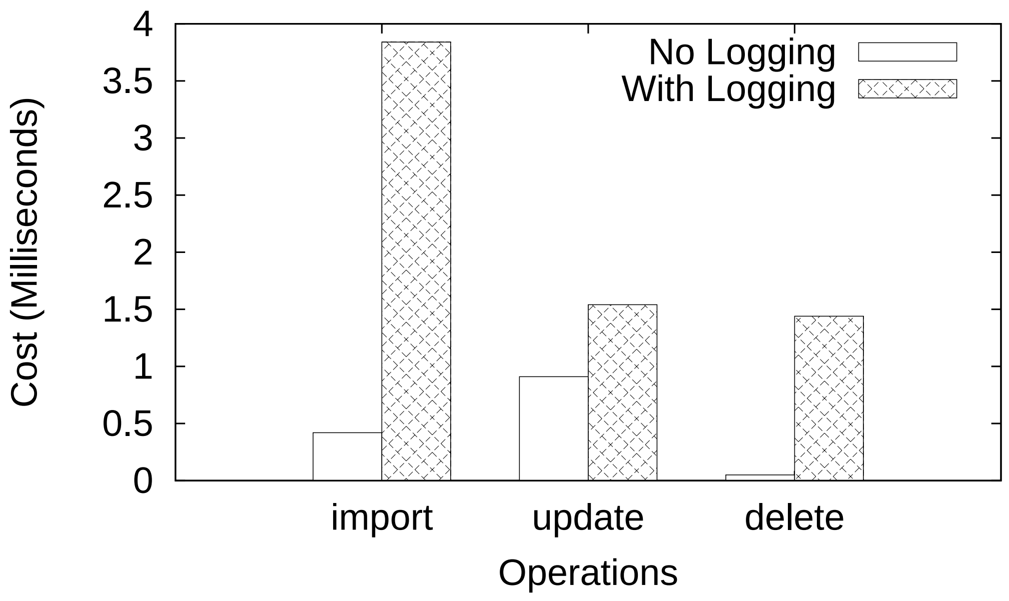


Figure 4.9. Metadata Logging

into a metadata log table. The performance penalty is about 40 percent. While import and delete result in multiple inserts into the log table.

4.7 Dynamic Data Migration

In early section, I showed that hardware failure is unavoidable in a distributed environment. Hard drive goes bad all the time. New storage node is added to the systems to either increase the storage capability or to replace a failing node. Either way, when a new storage node is added, data need to be moved to a new node, new replicas will be created to replace the damage one or to maintain the load balancing among group.

To demonstrate the data migration and fault tolerance features of ROARS, we set up a migration experiment as follows. We added 16 new storage nodes to our current system, and we started a `MIGRATE` process to spawn new replicas. Starting with 30 active storage nodes, we intentionally turned off a number of storage nodes during `MIGRATE` process. After some time, we turn some storage nodes back on, leaving the others inactive.

By dropping storage nodes from the system, we wanted to demonstrate that ROARS still could be functional even when hardware failure occurs. Figure 4.10 demonstrates that ROARS remained operational during the `MIGRATE` process. As expected, the performance throughput takes a dip as number of active Storage Nodes decreases. The decrease in performance is because when ROARS contacts an inactive storage node, it would fail to obtain the necessary replica for copying. Within a global timeout, ROARS will retry to connect to the same storage node and then move on to the next available Node. Because storage nodes remain inactive, the ROARS continues to endure more and more timeouts. That leads to the decrease of system throughput. While the experiment was progressing, we added a number of storage nodes back to the system. As soon as number of nodes came back online, we see the increase in system throughput.

Although, throughput performance decreases slightly when there are only two inactive storage nodes, throughput takes a more significant hit when there is a larger number of inactive storage nodes. There are ways to reduce this negative effect on performance. First, ROARS can dynamically shorten the global timeout, effectively cutting down retry time. Or better yet, ROARS can detect inactive storage nodes after a number of failed attempts, and blacklist them, thus avoiding picking replicas from inactive Nodes in the future.

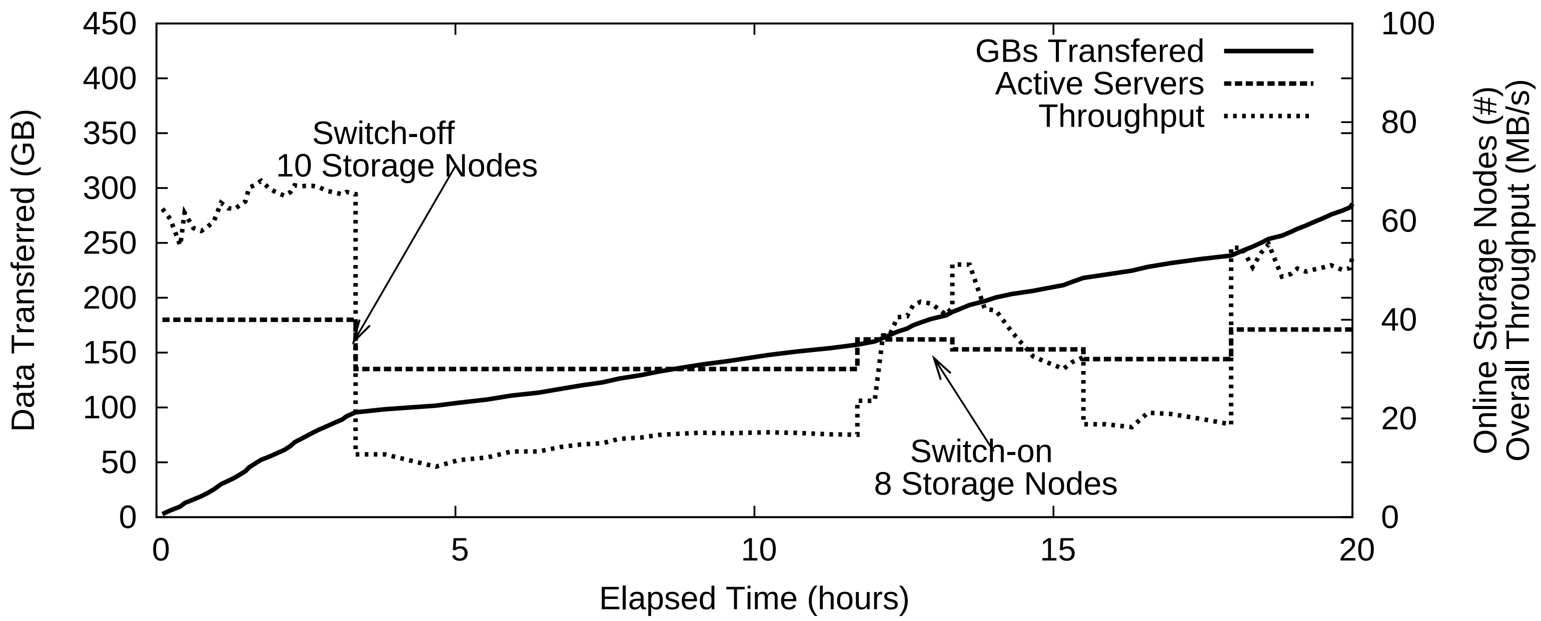


Figure 4.10: Dynamic Data Migration

CHAPTER 5

ROARS AND BIOMETRICS DATA

The last chapter shows that ROARS provides scalable, fault-tolerant data storage with metadata support for scientific data. Users can utilize ROARS to store, manage, update, and query for data easily using a useful set of tools. `IMPORT` and `EXPORT` performance are adequate for daily data ingestion and data query from a number of concurrent users. ROARS can not only manage scientific data, but also present data to users in a more effective way which previously was not possible.

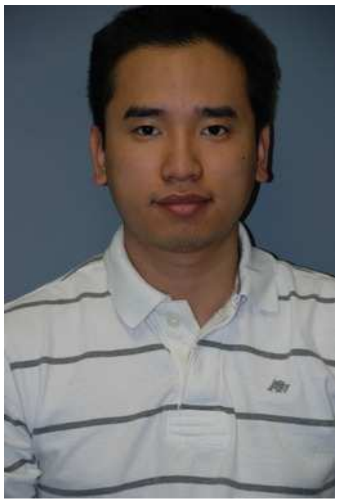
ROARS has been used to manage a large biometrics data repository (BXGrid) at Notre Dame. We have developed a step by step model to capture, ingest, validate, and prepare data for biometrics research. During the life-time of biometrics data, there are many hidden errors which can be introduced into the data. Those errors can affect the overall quality of data, and thus can skew the results of biometrics research. ROARS helps researchers improve data quality and sub-sequentially the quality of their research. ROARS provides data replication, automated data validation, and metadata provenance which are necessary and crucial to improve the quality and reliability of biometrics data.

5.1 Biometrics Research

Biometrics researchers study human body characteristics in the context of identification. Scientists develop algorithms to identify and confirm a human identity by comparing those characteristics with a known set and using a measurement of a physical trait. There have been a number of studies detailing the effectiveness of using human body characteristics such as fingerprint [52], hand [33], iris [18], and face [83] to identify or to verify an identity claim. However, questions remain about how to improve speed and reliability of the identification process. Nowadays, with the popularity of cloud computing [71], biometrics researchers have a very powerful tool to study the correctness and effectiveness of their biometrics recognition algorithms.

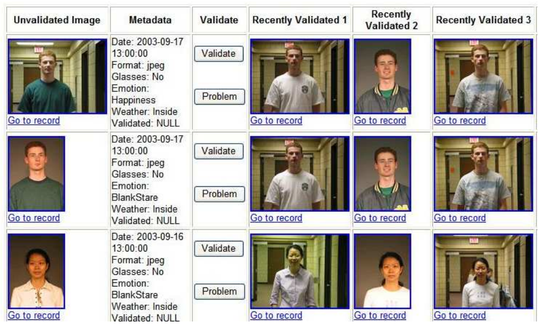
The Computer Vision Research Lab (CVRL) at the University of Notre Dame collects hundreds of gigabytes of biometrics data every semester from students, staffs, and faculties. Data is ingested and maintained in the BXGrid system for internal use to study newly developed algorithms. Data is also exported and shipped to the National Institute of Standards and Technology (NIST) to enter into a national research database. Examples of biometrics data are iris, face (still images and movies), and 3D face scans. Every image, movie, and scan collected is considered a **recording**. There are a number of metadata attributes attached to a recording. The metadata is gathered during the acquisition of each recording. Figure 5.1 shows an example of a face recording.

BXGrid is tailored from ROARS to support storage of biometrics data and metadata and to facilitate large scale experiments using distributed tools available from the Cooporate Computing Lab. Most end users interact with the system through the web portal, which allows for interactive browsing, data export in



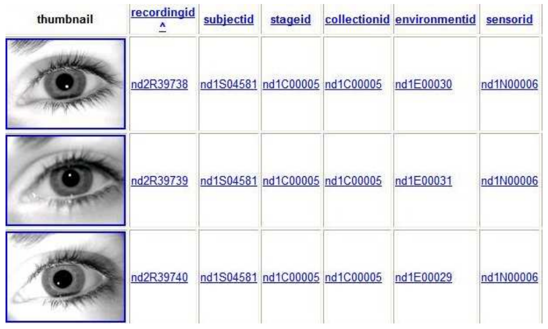
id	numeric	64427
recordingid	string	nd4R91445
shotid	string	2009-084-004-neutral.NEF
sequenceid	string	05432d373
date	string	2009-03-25 00:00:00
format	string	nef
subjectid	string	nd1S05432
glasses	string	No
source1	string	Retrospectively
emotion	string	BlankStare
source2	string	Given
stageid	string	nd4T00014
weather	string	Inside
collectionid	string	nd1C00031
environmentid	string	nd1E00069
sensorid	string	nd1N00012
illuminantid1	string	nd1I00010
state	string	enrolled
fileid	numeric	430941
by_user	string	slagree
lastcheck	string	2009-04-09 09:49:11
date_added	string	2009-03-31 10:00:40
added_by	string	dwright2
temp_collectionid	string	1238508120
YOB	numeric	1982
gender	string	Male
race	string	Asian

Figure 5.1. Sample Face Image and Metadata



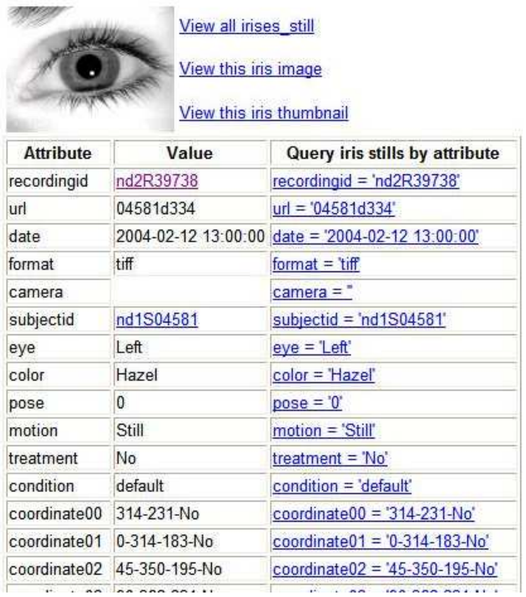
Unvalidated Image	Metadata	Validate	Recently Validated 1	Recently Validated 2	Recently Validated 3
Go to record	Date: 2003-09-17 13:00:00 Format: jpeg Glasses: No Emotion: Happiness Weather: Inside Validated: NULL	Validate Problem	Go to record	Go to record	Go to record
Go to record	Date: 2003-09-17 13:00:00 Format: jpeg Glasses: No Emotion: BlankStare Weather: Inside Validated: NULL	Validate Problem	Go to record	Go to record	Go to record
Go to record	Date: 2003-09-16 13:00:00 Format: jpeg Glasses: No Emotion: BlankStare Weather: Inside Validated: NULL	Validate Problem	Go to record	Go to record	Go to record

(a) **Validate Page)**



thumbnail	recordingid ^	subjectid	stageid	collectionid	environmentid	sensorid
	nd2R39738	nd1S04581	nd1C00005	nd1C00005	nd1E00030	nd1N00006
	nd2R39739	nd1S04581	nd1C00005	nd1C00005	nd1E00031	nd1N00006
	nd2R39740	nd1S04581	nd1C00005	nd1C00005	nd1E00029	nd1N00006

(b) **Browse Page)**



View all irises_still

View this iris image

View this iris thumbnail

Attribute	Value	Query iris stills by attribute
recordingid	nd2R39738	recordingid = 'nd2R39738'
url	04581d334	url = '04581d334'
date	2004-02-12 13:00:00	date = '2004-02-12 13:00:00'
format	tiff	format = 'tiff'
camera		camera = ''
subjectid	nd1S04581	subjectid = 'nd1S04581'
eye	Left	eye = 'Left'
color	Hazel	color = 'Hazel'
pose	0	pose = '0'
motion	Still	motion = 'Still'
treatment	No	treatment = 'No'
condition	default	condition = 'default'
coordinate00	314-231-No	coordinate00 = '314-231-No'
coordinate01	0-314-183-No	coordinate01 = '0-314-183-No'
coordinate02	45-350-195-No	coordinate02 = '45-350-195-No'

(c) **Record Detail Page)**

Figure 5.2: Examples of the Web Portal Interface

various forms, dataset management, and system administration. Figure 5.2 shows examples of BXGrid's portal pages.

5.2 Overview of Acquisition

The CVRL collects data bi-weekly during Fall and Spring semesters. Each acquisition involves several lab technicians and employs a number of biometrics sensors. Acquisition needs to be carried out as quickly as possible according to a plan to ensure the quality of data collected and the correctness of derived metadata. Acquisition usually includes a number of stations. Each station requires one or more lab technicians to monitor and capture data as subjects proceed through. A station uses one or multiple sensors with different lighting conditions. A sensor can produce more than one recording. A recording can be a picture, a movie, or a 3D scan.

5.2.1 Acquisition Setup

The first step of any acquisition session is to set up the stations based on an acquisition specification. The job of the setup technician is to follow the specification in order to determine the placement of sensors (camera, camcorder, scanner) and illuminant sources. In addition, the specification provides the position of subjects and number of recordings captured per subject per sensor. After setting up the station according to the specification, technicians perform a mock acquisition to make sure the equipment functions properly, and then eliminate any remaining problems observed.

5.2.2 Data Acquisition

As subjects start an acquisition session, each of them is given a **session id**. This **session id** is used to synchronize each captured recording and its metadata, such as subjectid, stageid, eye color, etc. Subjects go through a number of stations, recordings are captured at each station, and metadata is recorded. During the acquisition, technicians capture these data and act as the first quality screening gate. They make sure that eyes are open during iris acquisition, faces are unobstructed during face acquisition, and so on. They will initiate re-acquisition if deemed necessary.

During acquisition, metadata is captured along with each recording. Metadata includes lighting conditions, sensor specifications, relative position of subject to sensor and lighting (e.g. subject 6' away from camera and illuminant 8' above ground, 6' directly in front of subject). Other metadata contains personal information regarding the subject, such as eye color, race, and age. Another set of metadata is a recording of specifications such as format, resolution, and length (for video).

5.2.3 Pre-ingestion Assembly of Data

After acquisition, there are several types of recordings that need to be processed before ingesting. HD video needs to be clipped by subject, renamed, then transcoded to MPEG format. BMP images need to be converted to TIFF format. Iris videos need to be clipped by subject and eye (left,right), then transcoded to MPEG format and renamed. Data and metadata need to be gathered and synchronized before ingesting into a distributed storage system. While computer controlled sensors have the session id built into the recording's filename, manually operated sensor recordings need to be renamed. The new filename includes session id, date, description of a recording (regarding either quality – high, low – or activity classification – still, movement, etc.)

The next step is to collate metadata from various sources into a spreadsheet that links it to the correct recording. Some data comes from subject registration, e.g., eye color, glasses, age; some is environment-dependent e.g., sensor id (sensor information), illuminant id (lighting information). Metadata is then converted to name value pair format and is ready to be ingested. The name-value-pair format is similar to the metadata shown in Figure 5.1. Metadata name, type (numeric or string), and value are separated by tabs, while recordings are separated by an empty line.

5.2.4 Data Ingestion and Data Storage

After being prepared, data is ingested into BXGrid by invoking an `IMPORT` command. BXGrid automatically replicates data and associated metadata across multiple storage servers. BXGrid provides data redundancy to assure data quality and data integrity. Data information such as size of file and checksum are kept internally inside BXGrid.

5.2.5 Data Validation

The acquisition process clearly leaves a lot of room for error. With so many people working with such a large number of images, mistakes are not only probable but inevitable. In order to find these errors and combat their permanent entry into the repository, all image records have a `state` attribute. A newly imported record is initially in the `unvalidated` state. For an image 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. If the technician identifies an error in the metadata, such as an incorrect subject, or a left eye labelled as a right eye, 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 workers.

A second level of approval is required before an image 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 number.

5.2.6 Data Enrollment

The final step in processing a recording is to enroll it. Once a record is enrolled, it should not be edited or changed in any way. During the enrollment process, a record is associated with a **collectionid** and given a **recordingid** that is used to identify the image in any subsequent research and ensuing publication. Another unique metadata named **sequenceid** is assigned to each recording. The **sequenceid** is used internally at Notre Dame by the CVRL. Additional metadata that must be kept internally for bookkeeping purposes are **shotid**: original file-

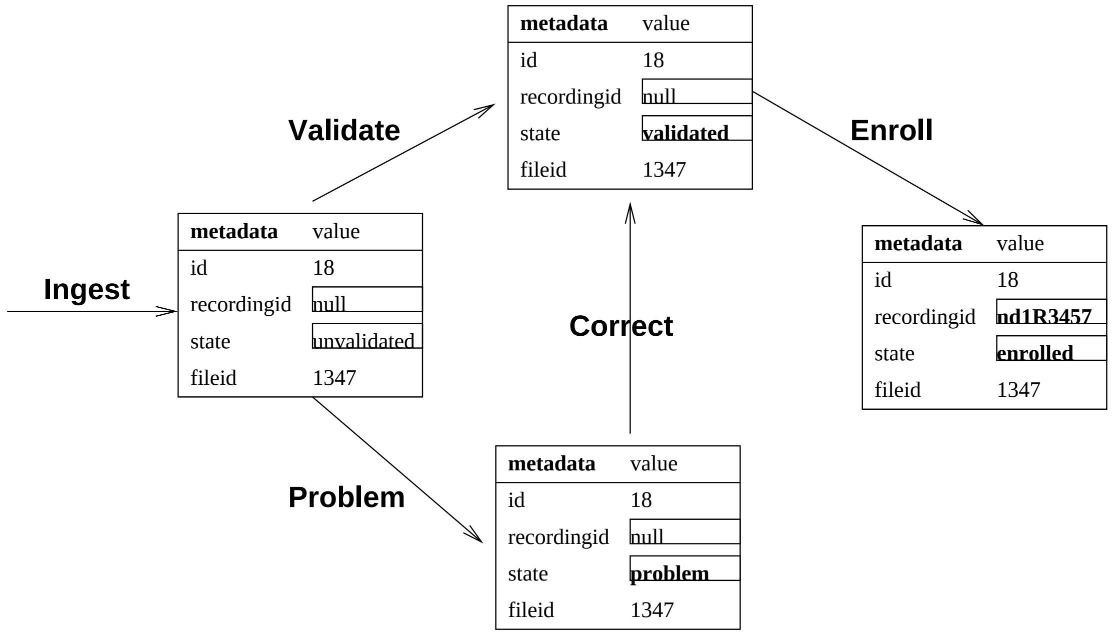


Figure 5.3. Data Life Cycle. Metadata changes during validation process. New metadata is assigned when data is enrolled.

name, and **batchid**: unique number for a collection session. BXGrid supplies the structure for creating a collection in a format that is consistent with a US government Document Type Definition Reference Document. It provides a template for naming the collection and allows the user to specify the type of data and the acquisition dates to be included with a few simple buttons. Once the user verifies her choices, BXGrid generates the **recordingid** for each of the included images and adds the collection to the collections table. Figure 5.3 shows the life cycle of a recording.

5.3 Improve Biometrics Data Quality

Biometrics, like many modern science and engineering research fields, is data-driven. Data enters the research enterprise through sensors and is processed, yielding derivative data sets, some of which feed comparisons that are used to evaluate the sensing technology, the steps in the processing pipelines leading to the comparisons, and the comparison techniques. Such evaluations must be performed with statistical rigor, which drives the collection of data to support the conclusions reached. Management of this data is a demanding task and the data sets' integrity must be assured through appropriate management and validation techniques. The use of ROARS to store and maintain the integrity of data, coupled with web services and portals that allow crowdsourced evaluation work and data access, is an ideal management strategy for large data sets such as those used in biometrics.

5.3.1 Issues That Can Affect Data Quality

Creating and maintaining a large repository of biometrics data can be challenging in many ways. One hundred thousand data files can add up to terabytes of data. Because of the size of the repository and the fault-prone nature of both humans and computers, data quality can be affected throughout the life cycle of data. Error can be introduced into data at any time during pre-acquisition, during acquisition, during ingestion and after ingestion. Depending on the nature of the errors, solutions to correct errors can be recapturing data, modifying metadata, or removing data completely.

During acquisition, equipment can malfunction (e.g. a camera does not take a picture, the flash does not trigger). Other errors can be due to carelessness of lab technicians (e.g. camera has a wrong zoom setting, unnoticed blinking

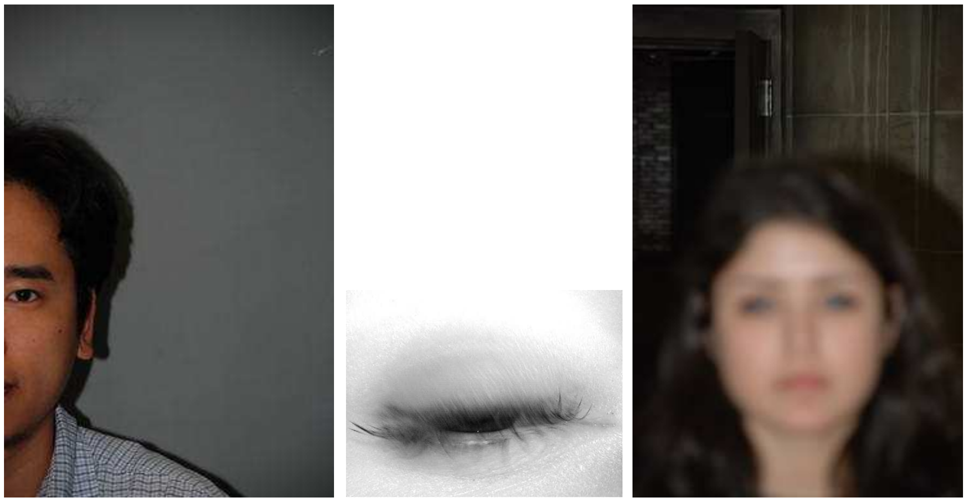

(a) Wrong camera position. (b) Blinking eye. (c) Out of focus camera.

Figure 5.4. Example of problem recordings.

eyes at the time of data capture). Another error occurs when subjects get out of order during acquisition. Figure 5.4 shows some of the problem recordings. Because each acquisition usually includes a number of stations, a subject jumping the station line will cause a string of mislabeled data. This proves to be costly when data is enrolled and used in experiments because it can inadvertently affect experiment results. Mistakes during acquisition can be easily corrected if the lab technician pays attention during operation and identifies the mistakes. Once a mistake is identified, steps are carried out to correct the mistake, ranging from logging the discrepancy to retaking a picture or a movie.

After acquisition, the lab technician uses various tools to prepare the data for the ingestion process. Data collected during acquisition is copied into local storage for pre-processing purposes. A script is used to rename the default filename to a more meaningful one. Data, such as video, will be edited. Problems may arise when the renaming script does not perform as intended or when video cutting fails. Mistakes during this stage can be eliminated by carefully processing data and also by maintaining a stable, working set of tools.

When data is ready to ingest, the lab technician invokes a `SCREEN` then an `IMPORT` command to ingest data into the repositories. Each ingestion is assigned a **batchid**. The batchid is very useful for keeping track of each data acquisition, and also for correcting mistakes when mistakes are made. Ingestion can fail unexpectedly due to malfunctioning hardware or power outage. When ingestion is interrupted, the lab technician can invoke the same `IMPORT` command to resume the ingestion. `IMPORT` command will automatically start where the last `IMPORT` command left off. `IMPORT` also has built-in redundancy detection. When a batch is ingested twice, `IMPORT` will ignore already ingested data. When a batch needs to be deleted due to error, the lab technician can identify batchid and invoke `DELETE` to erase the batch from the repository.

The last step to assure data quality before enrollment is the validation process. Lab technicians validate data using a web portal. The web portal allows the technician to identify poor quality data by displaying data and comparing data from the same subject. Common metadata mistakes are mislabeling, such as left eye to right eye and vice versa, subject wrongly marked as wearing glasses, and data assigned to the wrong subject. By providing a comparison view between unvalidated data and already validated data from the same subject, lab technicians have a better chance of detecting these types of mistakes and correcting them accordingly. Figure 6.9 shows an example of a validation page.

Data quality plays a very important role in the success of an experiment. Data and metadata have to match correctly. Wrongly matched data and metadata can alter the result of an experiment. BXGrid employs a number of mechanisms

TABLE 5.1: SUMMARY OF PROBLEMS AND SOLUTIONS

Stage	Problem	Solution
Acquisition	Equipment malfunctions	Discard image/movie, reset, or replace equipment
Acquisition	Subject jumps out of order	Lab technicians detect and correct the order
Ingestion	Ingestion is interrupted	Re-run ingestion command
Validation	Incorrect metadata	Lab technicians correct metadata using web portal
Validation	Length of validation process	Automated Data Validation
Validation	Metadata inconsistency	Two phases of metadata update, database then flush to storage
Archival	Hardware failure	Replicate and store metadata in three storage servers
Archival	Data inconsistency	Audit and Repair process
Archival	Validate/Enroll errors	Revert using metadata log
Archival	Loss of database	Recover by scanning metadata from fileservers

to assure the correctness, consistency, and availability of data. Table 5.3.1 lists the problems we have identified and steps we take to minimize or eliminate data quality problems.

5.4 Recent Data on Failure Rates and Recovery Mechanisms

Hardware failure is not uncommon, especially hard drive failure. A hard drive can fail because it is exposed to extreme conditions, such as heat, humidity, water, shock, etc. It also can fail due to use or aging [49], [58]. Google [49] published a study on commodity hard drive failure rate in 2007. Although the annualized failure rates are higher than those reported by hard drive manufacturers, given the scope and size of a Google disk farm, the number provided on hard drive failure rate is deemed to be accurate. According to Google, 2 percent of disks fail within a year, but the annualized failure rate jumps to 8 percent over two years and 9 percent in the first three years. The study shows that in order to sustain data through hard disk failure, we should plan to backup, replicate and audit data more often, and we should plan to provision new hard drives to replace old ones that are prone to failure.

Hardware failure is unavoidable for a production system like BXGrid. BXGrid employs as many as 41 file servers, and after a year of operation, some of them have already suffered hardware failure. Most common failures are bad hard drives and bad SATA controller boards. In the case of a bad hard drive, a new hard drive is added to replace the bad one, and all data on the drive is lost. In the case of a bad SATA controller board, data is intact and recoverable with a new controller board. As recent studies on hard drive failure show, system administrators need to run data audit and repair frequently. However, as the amount of data grows,

TABLE 5.2

AUDIT AND REPAIR TIMELINE

Period	**Elapsed Time**	**Files Checked**	**Suspect**
1	24 hours	80,000	16,244
2	24 hours	80,000	15,153
3	48 hours	160,000	1,227
4	16 hours	60,000	9,381
5	32 hours	160,000	0
6	28 hours	160,000	0

it is not feasible to perform auditing on the whole system every day. Thus we have been running audit and repair only during night time when BXGrid usage is minimal. In order to test BXGrid's ability to recover from hard drive failure, we intentionally removed several hard drives from the storage cluster. We ran BXGrid audit and repair incrementally to detect and replace missing replicas.

Table 5.4 shows the length of each audit and repair run, the number of audited files and the number of repaired replicas when hardware failure was deliberately introduced into BXGrid. During the recovery process, BXGrid remained in operational mode, and was accessible by multiple users performing regular tasks, such as import, export, validate, enroll, etc.

During December 2009, four storage servers suffered from hard drive failures. After identifying problem servers, `REPAIR` was invoked to spawn new replicas. These replicas replaced those from problem servers and kept the number of replicas for each file at three. The repair process took just over five hours to replace an estimated 26,000 missing replicas, a total 250GB of data. The repair process took significantly less time than the audit process because auditing involves expensive checksum calculations. Repair process throughput is mainly bounded by the speed of network links between storage servers, a mixture of gigabit and 100Mbps network.

5.5 Current status of BXGrid

At the time of writing, BXGRid has been in use as the archival service for a biometrics research group at the University of Notre Dame for over three years. BXGrid is used to curate data which is transmitted to the National Institute of Standards and Technology for evaluation of biometric technologies by the federal government. Approximately 60GB of new data is acquired in the lab on a bi-weekly basis, while collections on legacy storage devices are gradually being imported into the system.

Figure 5.5 shows the growth of BXGrid over time from 2008 to 2009. 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.

BXGrid currently contains 853,004 recordings totalling 14.1TB of data, spread across 40 storage nodes. Figure 5.6 shows that the filesize distribution in BXGrid. The repository is dominated by small and medium size files because the majority of the files are iris and face images. Only a small portion of BXGrid consists of bigger video and 3D files.

The data model fits ROARS perfectly because the raw data never changes

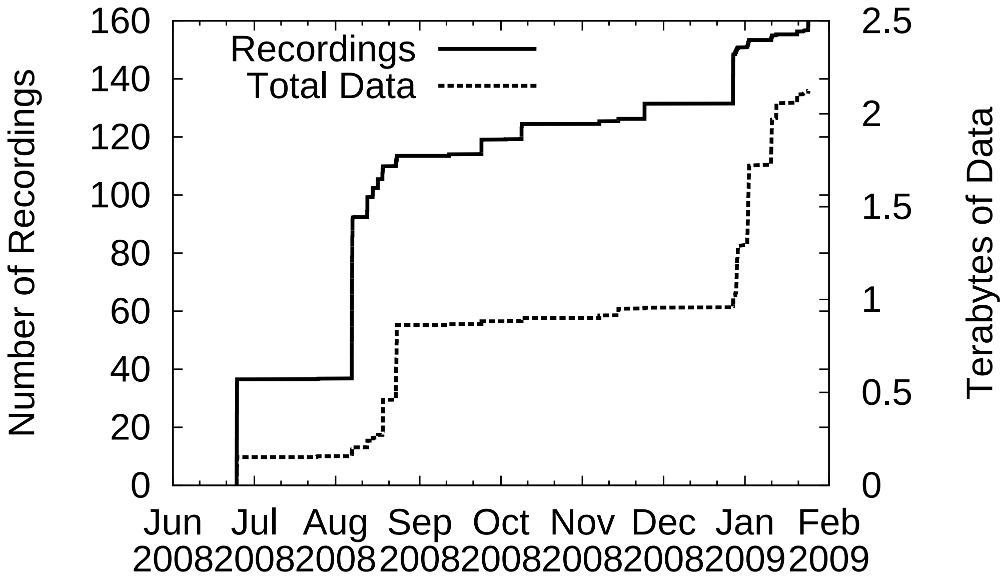


Figure 5.5. System Growth Jul 2008 - Jan 2009

after its initial ingestion. However the metadata can change or more precisely will change throughout the biometrics team's validation and verification process. When a *recording* is first ingested, it is marked as *unvalidated.* The state, which is a part of *recording* metadata, can be changed to *validated* or *problem* during a validation process. A *recording* is deemed to be *problem* if its metadata is mislabeled or the *recording* itself is unusable. In the case where its metadata is mislabeled (e.g. right iris is flagged as left iris), the metadata can be modified and the state of the *recording* is set to *validated.* At the end of this whole process, the state of the *recording* changes to *enrolled*, and a **collectionid** is assigned. **collectionid** differs from **batchid** because it is a unique number usually representing a semester worth of data.

In the last 6 months, there has been 1,685,509 entries inserted into the log table. So far, 48 users has modified 21 types of metadata. More than half of the

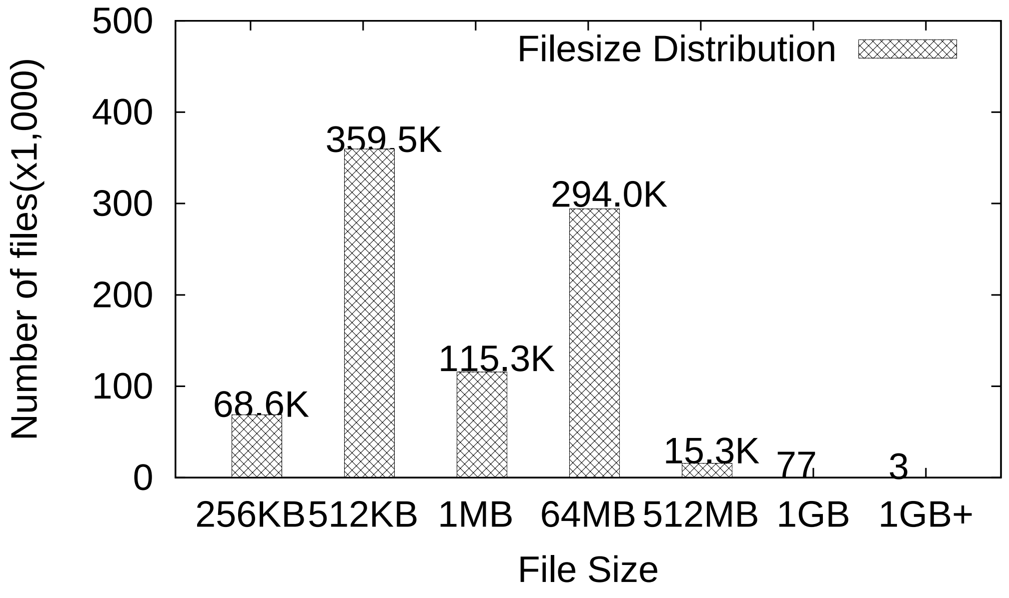


Figure 5.6. Filesize Distribution in BXGrid

total metadata changes were related to state changes, and they were made during validation and enorllment process. The rest of metadata changes concentrated on a few metadata: lighting condition, weather condition and yaw angle of face images.

In May of 2011. We upgraded the storage cluster for BXGrid. We removed 32 aging storage nodes from the storage pool and we added 32 new storage nodes. Each storage node consists of 32GB RAM, twelve 2TB SATA disks and two 8-core Intel Xeon E5620 CPUs. All of them are equipped with Gigabit Ethernet. We safely removed the old nodes from the system and migrated the data to the new nodes. Figure 5.7 shows the entire migration process. It took 40 hours to move approximate 5TB to the new nodes.

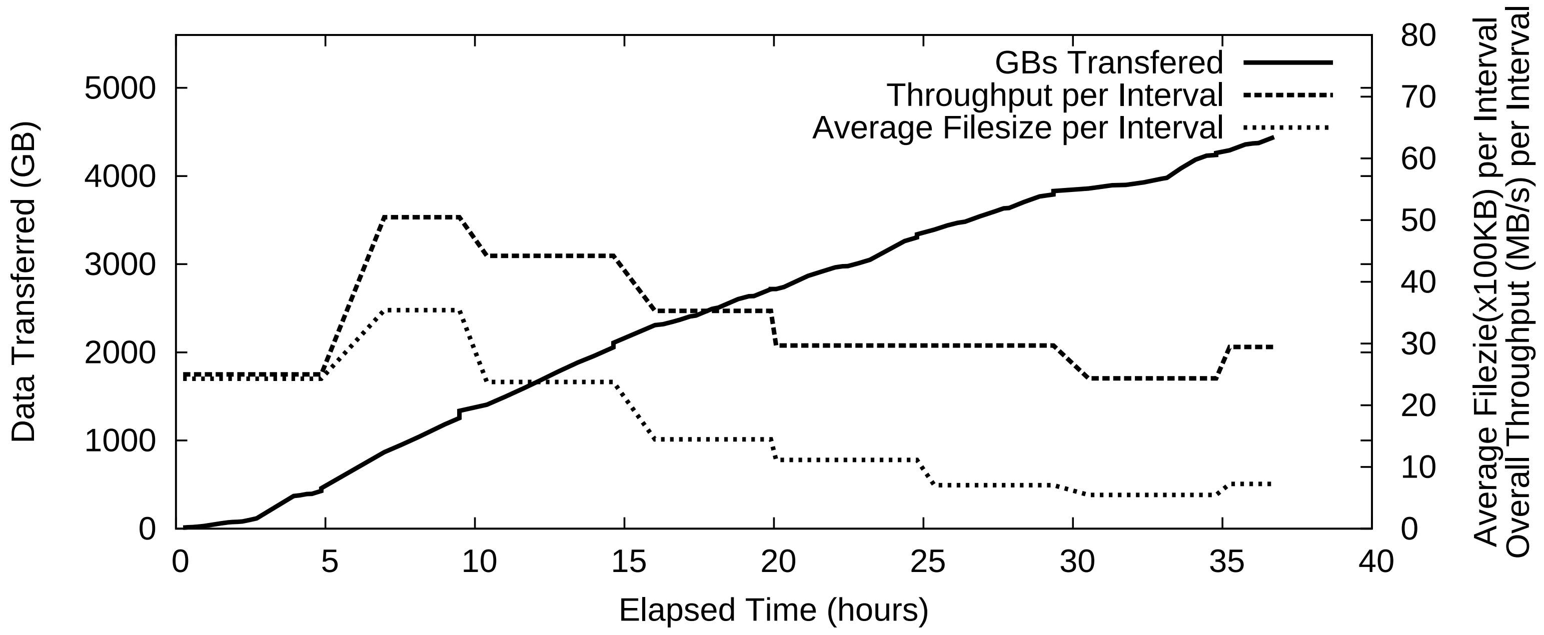


Figure 5.7: BXGrid Data Migration To A New Cluster

CHAPTER 6

ROARS INTEGRATION WITH WORKFLOWS

Chapter 5 demonstrates ROARS' usefulness as a biometrics data repository. In addition to provididing safe storage for biometrics data, ROARS also helps researchers speedup their research by taking advantage of the distributed nature of ROARS. In order to demonstrate the ability of ROARS to integrate with a number of abstractions and scientific workflows [77], this chapter will give a number of examples of abstractions and workflows which take advantage of ROARS in the context of biometrics research.

6.1 Distributed Computing Tools

The Cooperative Computing Lab at the University of Notre Dame provides a number of tools to help users from other disciplines to harness the power of large distributed systems.

Work Queue [82] is a scalable and robust master/worker framework, which provides an API for users to write their own distributed application. Users can define and submit tasks to a worker queue. Tasks are sent to and executed on any available worker machines. After finishing the assigned task, the worker reports the result to a master and asks for another task. The role of the master is to distribute tasks and manage the results.

All-Pairs [42] is an abstraction which takes in two sets of objects, A and B, and performs a function F on any pair of objects (a,b) such that a belongs to set A and b belongs to set B. Users provide set A, set B and function F. The All-Pairs abstraction executes the work load in a distributed manner and handles automatically other details such as fault tolerance, data movement, etc Users do not have to be a distributed system expert to run All-Pairs workloads.

Makeflow [4] is a workflow engine that assists users with executing large and complex scientific workflows in number of distributed environments such as cluster, clouds, and grids. Users can use Makeflow to execute their application using supported distributed frameworks such as Work Queue or All-Pairs.

Weaver [15] is a Python-based workflow compiler for distributed applications. Weaver supports several common distributed computing patterns. The result of an application compiled by Weaver is workflow described in Makeflow format.

6.2 Abstractions for Biometrics Research

Motivated by the advice of Gray [28], 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 6.1:

- **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, C)** : Apply convert function C to all members of set S, yielding the output of C attached to the same metadata as the input. This

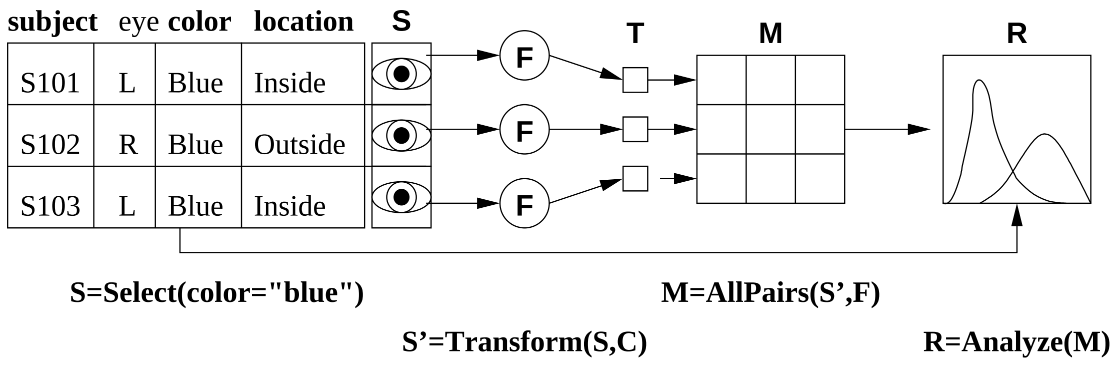


Figure 6.1: Workflow Abstractions for Biometrics

abstraction is typically used to convert file types, or to reduce an image into a feature space such as an iris template, an iris code or a face geometry.

- **All-Pairs(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.

- **Analyze(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 a histogram or an ROC curve.

6.2.1 Select Abstraction

The first step in experimentation is to select a dataset. Select abstraction is equivalent to a `EXPORT` request for both data and metadata. Because most users are not SQL experts, the primary method of selecting data is to compose 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

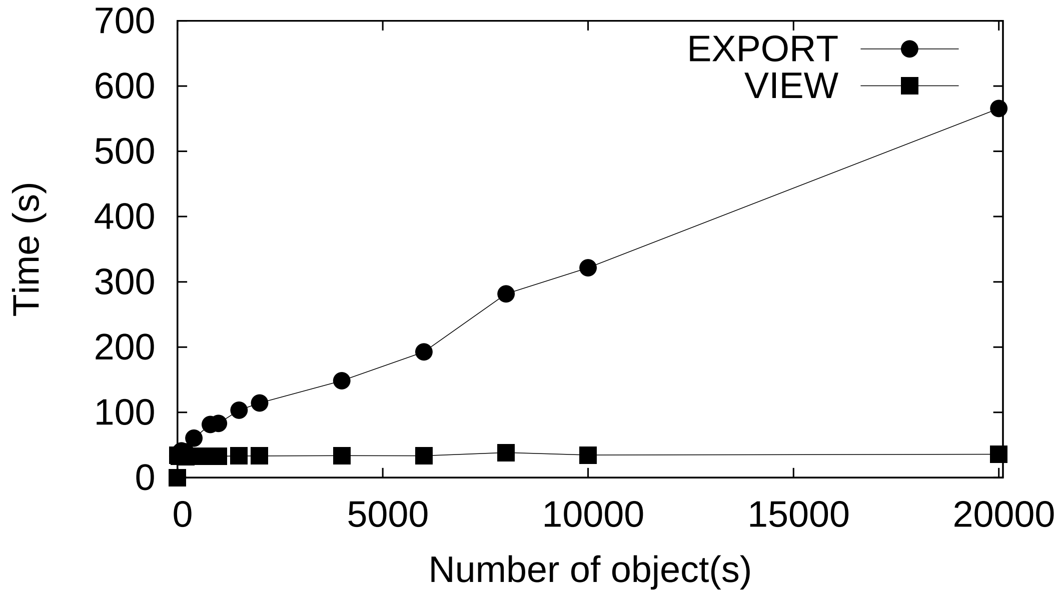


Figure 6.2: Export and View performance

through a text interface, view the results graphically, and then save the results for other users.

As described in Chapter 3, there are multiple ways for users to get data out of BXGrid. They can use `EXPORT` to download actual data objects with metadata to local storage. Then they can choose to process data locally. They also can have data distributed and analyzed remotely. If users choose to run experiments on data in a distributed workflow, data has to move twice thus it is not optimal. First, data is moved from BXGrid to local storage, and then once again from local storage to remote nodes.

In order to be more efficient, users can use `VIEW` to create a materialized view of the dataset on local storage. They can run experiments on the data using either FUSE or Parrot. If they choose to run their experiment remotely in a distributed manner, they can send a Chirp ticket along with the materialized view. The remote jobs will use the Chirp ticket [23] to gain the access to the actual data on BXGrid's storage nodes. Instead of sending actual data which could be Gigabytes in total, users only need to send a set of symbolic links which point to the location of the data in BXGrid's storage nodes. By using `VIEW`, the data only needs to be moved once from BXGrid to the remote job's location. Figure 6.2 shows the cost of `EXPORT` and `VIEWS` for various datasets. As the datasets get bigger, `EXPORT` performance grows linearly with dataset's size while `VIEWS`'s runtime stays at constant. It is because `EXPORT` transfers data from BXGrid's storage node to local storage while `VIEWS` only creates symbolic links to the data.

6.2.2 Transform Abstraction

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 local storage or on a distributed system, 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, or a function converts images and videos to thumbnails for web pages. The result will inherit information such as: left eye, subjectid, environmentid, etc. from the original iris image.

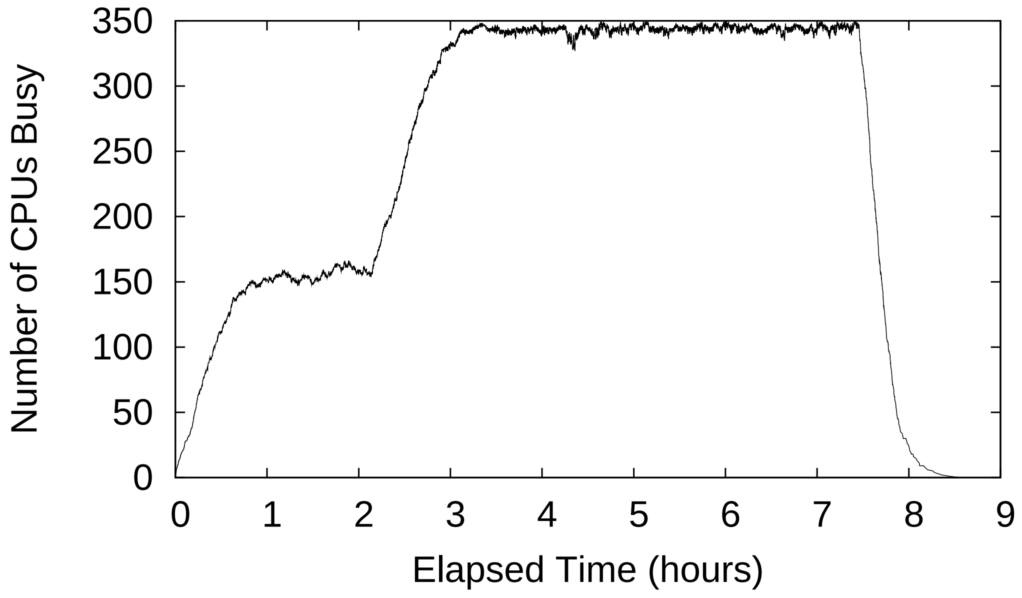


Figure 6.3: All-Pairs on 4000 Faces

6.2.3 All-Pairs Abstraction

All-Pairs abstraction helps users perform a large-scale comparison. The user uploads or chooses an existing comparison function and a saved data set. This task is very computation intensive and requires dispatch to a computational grid. Details of the implementation of All-Pairs is described in an earlier paper [43] 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, 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 [73]. Figure 6.3 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 on a active storage cluster.

6.2.4 Analyze Abstraction

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 in 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.

Given Select, Transform, All-Pairs, and Analyze abstractions as an interface to the repository, new workloads can be constructed to solve interesting problems and answer research questions in biometrics.

6.3 Biometrics Workflow

A workflow is a series of tasks that are executed in order to achieve a final result. In science experiments, a workflow is important because it defines the specification of the experiment. In other words, workflows convey the blueprint of the whole experiment and include all necessary steps to achieve the final goal. A workflow usually is represented by a directed acyclic graphs (DAG). Figure 6.4 shows a simple biometrics workflow.

This workflow compares two iris images using a function F. Note that before

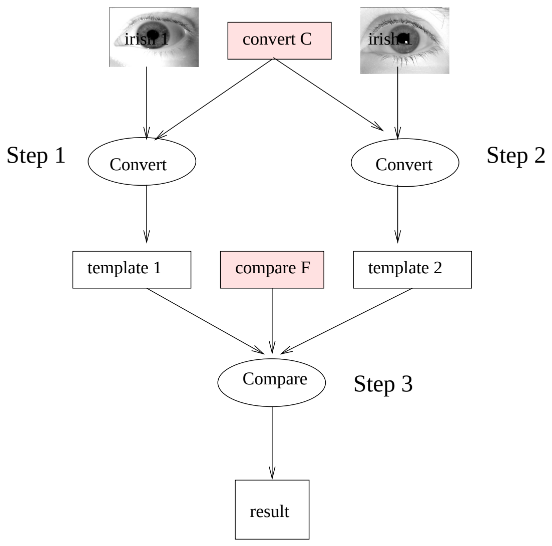


Figure 6.4: DAG Workflow for comparing two irises

```
template1: iris1.tiff convertC                    //Rule #1
        ./convertC iris1.tiff template1

template2: iris2.tiff convertC                    //Rule #2
        ./convertC iris2.tiff template2

result: template1 template2 compareF              //Rule #3
        ./compareF template1 template2 > result
```

Figure 6.5. Makeflow code that creates two iris template and compare the templates

executing function F, the two iris images need to be transformed into a template format. For simplicity, let us assume that all steps, 1, 2 and 3, take 1 second to complete. If the workflow is executed sequentially, it would take 3 seconds to complete all tasks (2 seconds for running convert function C twice and 1 second for running compare function F once). It is possible to finish the workflow in 2 seconds if the images are converted to templates concurrently. With a more complicated workflows, for example a workflow that compares thousands or millions of irises, if tasks can be executed in parallel, the runtime of the whole experiment will decrease significantly.

There are a number of workflow management systems [84], [3], [20], [40], [47] have been developed to assist scientists with running distributed workflows. At Notre Dame, the Cooperative Computing Lab has developed Makeflow, a workflow management system that uses the traditional Make language syntax to express tasks and their dependencies. The advantage of Makeflow is that it is simple and portable. Makeflow's workflow can be executed across multiple execution engines including Condor, SGE, HDFS, and more. Figure 6.6 shows the architecture of Makeflow [4]. In this chapter, example workflows are executed by Makeflow on Local, Condor and WorkQueue. Figure 6.5 is an example of the Makeflow code representing the workflow in figure 6.4. Since there is no dependency between the first two rules, they can be executed concurrently. The last rule needs the output of the first two rules to complete.

6.3.1 BXGrid Transcode

BXGrid website helps users visualized biometrics data more easily. Images and videos are transcoded into smaller thumbnail size still images or animated GIFs before they are displayed on the website for users' viewing. Figure 6.7 shows an example of a browser page for iris images. Each page can have up to 100 images. If all 100 images need to be transcoded, the browser page will take a long time to load, which is unacceptable for users to wait. Moreover, when users validate data, there may be up to 600 images to transcode per page. Although the transcoded results are only generated once and kept in a cache, newly ingested images will

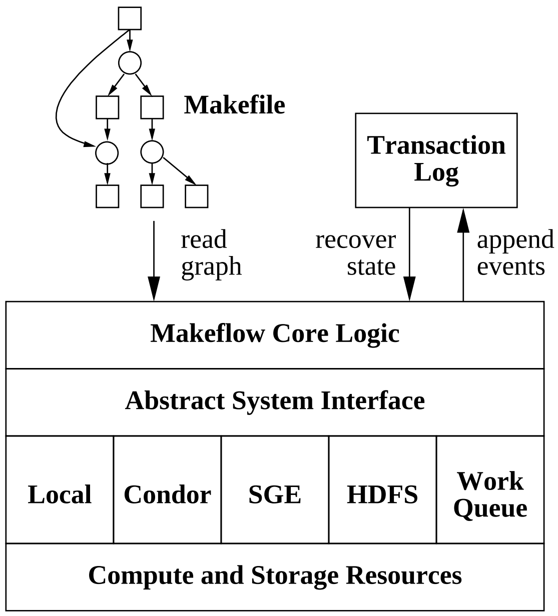


Figure 6.6: Makeflow Architecture[4]

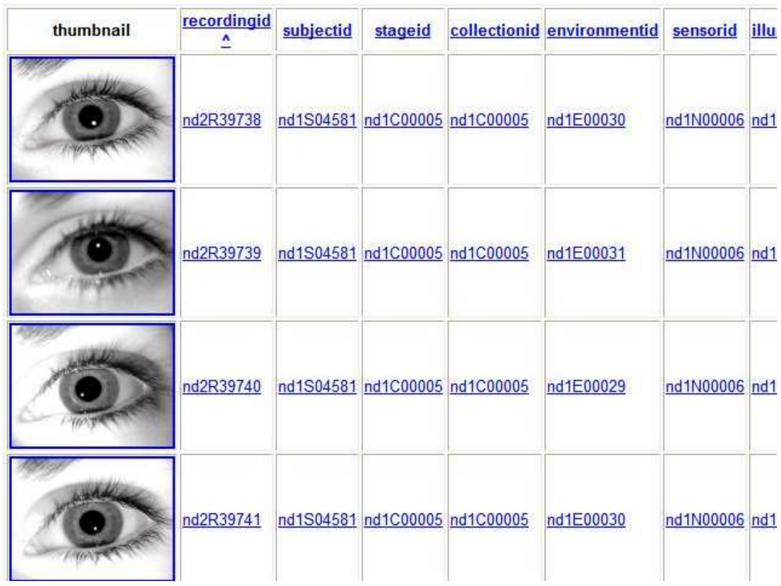
have no thumbnail. Browsing new data without already generated thumbnails will hinder users' overall experience.

In order to provide users with a more positive browsing experience, a transcode workflow was created to pre-generate the thumbnails for all images and videos. The workflow can be summarized as follow:

Question: How to select all new data from the last workflow execution and transcode them, and store the result in the cache?

```
S = Select(D)
T = Transform(S,F)
```

Figure 6.8 shows a Weaver program that compiles to Makeflow rules representing the transcode workflow to query and generate missing thumbnails for BXGrid website. The workflow then are executed using WorkQueue. The initial run, BXGrid transcode completed transcode 85.72 GB of biometrics data in 3.81 Hours



thumbnail	recordingid ^	subjectid	stageid	collectionid	environmentid	sensorid	illu
	nd2R39738	nd1S04581	nd1C00005	nd1C00005	nd1E00030	nd1N00006	nd1
	nd2R39739	nd1S04581	nd1C00005	nd1C00005	nd1E00031	nd1N00006	nd1
	nd2R39740	nd1S04581	nd1C00005	nd1C00005	nd1E00029	nd1N00006	nd1
	nd2R39741	nd1S04581	nd1C00005	nd1C00005	nd1E00030	nd1N00006	nd1

Figure 6.7. BXGrid's Browser Page

```
for file_type, command, query, cache_path in file_types:
   missing = [find_missing_thumbnails(f, cache_path, force=False)
   for f in query(bxgrid)]
      missing = filter(lambda x: x, missing)
      if len(missing) > CHUNK_SIZE:
         with SubNest(file_type, local=True):
            for i, chunk in enumerate(Chunk(missing, CHUNK_SIZE)):
               with SubNest('%s.%04X' % (file_type, i),local=True):
                  GenerateThumbnails(file_type, command, chunk, cache_path)
      else:
         with SubNest(file_type, local=True):
            GenerateThumbnails(file_type, command, missing, cache_path)
```

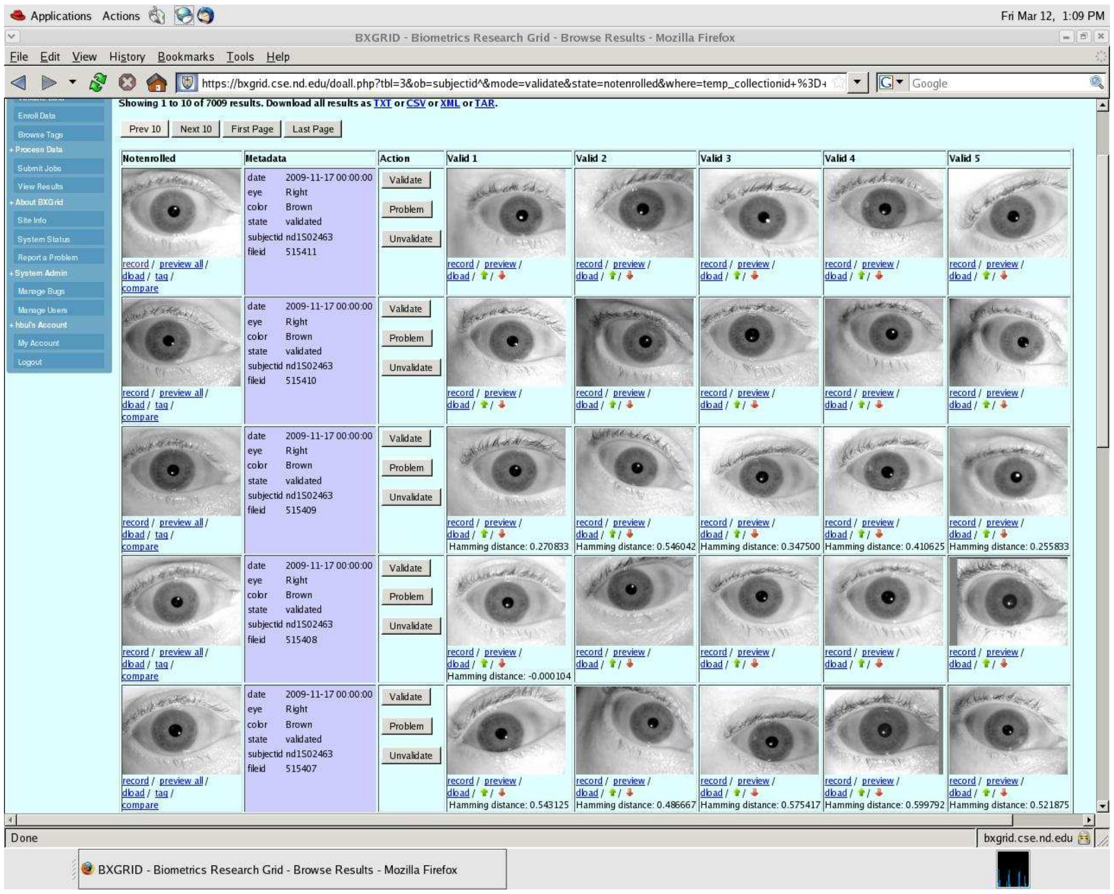
Figure 6.8. Weaver code that compiles to Makeflow rules that generates missing thumbnails

at the average of 22.5 GB/hour. Since then, the transcode makeflow runs automatically at midnight to find and generate thumbnails for newly ingested data.

6.3.2 BXGrid Auto Validation

The data collected through the CVRL acquisitions are used for research into biometric recognition algorithms by institutions throughout the world. Therefore, it is of the utmost importance that it is tagged correctly. The validation function of BXGrid allows for an efficient visual comparison of a newly acquired image against similar images of the same subject while also displaying the relevant metadata tags in a concise format. Records that have no problems are validated. If an error in the metadata or a problem with the image quality is discovered, the record is designated as a "problem" record which can be either eliminated from the data set, or corrected and validated at a later date. The problems encountered fall into two categories: image quality and incorrect metadata. Image quality problems might include blurriness of an image or intended feature not visible (eye closed, part of face cut off, image too dark/light, etc.), which make the image unfit for use in research. The more difficult problems to ferret out and resolve are those that involve incorrect metadata. This can range from an image being tagged as a subject wearing glasses when he/she is not, to an image being tagged with the wrong subject number. The first problem is fairly easy to spot and correct, however when a recording is tagged with a wrong subject, it is more difficult to identify the error, especially with iris images.

During the iris image validation process, lab technicians usually look for mistakes such as closed eye, incorrect subject, etc. Figure 6.9 shows an example of an iris image validation page. An iris image is displayed next to five "good" iris


Applications Actions
Fri Mar 12, 1:09 PM
BXGRID - Biometrics Research Grid - Browse Results - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://bxgrid.cse.nd.edu/doall.php?tbl=3&ob=subjectid^&mode=validate&state=notenrolled&where=temp_collectionid+%3D+
Google
Enroll Data
Browse Tags
+ Process Data
Submit Jobs
View Results
+ About BXGrid
Site Info
System Status
Report a Problem
+ System Admin
Manage Bugs
Manage Users
+ hbui's Account
My Account
Logout
Prev 10
Next 10
First Page
Last Page
Done
bxgrid.cse.nd.edu
BXGRID - Biometrics Research Grid - Browse Results - Mozilla Firefox

Showing 1 to 10 of 7009 results. Download all results as TXT or CSV or XML or TAR.

Notenrolled	Metadata	Action	Valid 1	Valid 2	Valid 3	Valid 4	Valid 5
record / preview all / dload / tag / compare	date 2009-11-17 00:00:00 eye Right color Brown state validated subjectid nd1S02463 fileid 515411	Validate Problem Unvalidate	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓
record / preview all / dload / tag / compare	date 2009-11-17 00:00:00 eye Right color Brown state validated subjectid nd1S02463 fileid 515410	Validate Problem Unvalidate	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓
record / preview all / dload / tag / compare	date 2009-11-17 00:00:00 eye Right color Brown state validated subjectid nd1S02463 fileid 515409	Validate Problem Unvalidate	record / preview / dload / ↑ / ↓ Hamming distance: 0.270833	record / preview / dload / ↑ / ↓ Hamming distance: 0.546042	record / preview / dload / ↑ / ↓ Hamming distance: 0.347500	record / preview / dload / ↑ / ↓ Hamming distance: 0.410625	record / preview / dload / ↑ / ↓ Hamming distance: 0.255833
record / preview all / dload / tag / compare	date 2009-11-17 00:00:00 eye Right color Brown state validated subjectid nd1S02463 fileid 515408	Validate Problem Unvalidate	record / preview / dload / ↑ / ↓ Hamming distance: -0.000104	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓	record / preview / dload / ↑ / ↓
record / preview all / dload / tag / compare	date 2009-11-17 00:00:00 eye Right color Brown state validated subjectid nd1S02463 fileid 515407	Validate Problem Unvalidate	record / preview / dload / ↑ / ↓ Hamming distance: 0.543125	record / preview / dload / ↑ / ↓ Hamming distance: 0.486667	record / preview / dload / ↑ / ↓ Hamming distance: 0.575417	record / preview / dload / ↑ / ↓ Hamming distance: 0.599792	record / preview / dload / ↑ / ↓ Hamming distance: 0.521875

Figure 6.9. Iris Images Validation Page

images from the same subject. A good iris is an iris that has been validated and verified to be correctly linked to that subject. Although there are attributes of a person's eye that could change over time [8], the eyes usually retain their similarities, such as shape and size of the iris. By looking at iris images from the same subject, lab technicians can easily spot an iris that does not belong to the assigned subject. The same process applies to the face validation process.

In order to speed up the iris validation process, a BXGrid auto-validation workflow has been created to take on these challenges. This workflow mimics a technician's actions by comparing the unvalidated image (gallery) to five already validated and verified images from the same subject (probe).

Question: Select all new irises, for everyone of them, answer the question: "Does this iris belong to the person as the metadata claims?"

```
S = Select(D)
For each s in S
  X = Select(subject=x.subject)
   s' = Transform(s',F)
  X' = Transform(X',F)
   All−Pairs(s',X',C)
```

Figure 6.10 shows the workflow for auto validation workload. Based on comparison scores between the gallery image and the probe images, one can deicide to accept the iris identity claim and mark the image as **validated**, or reject the claim and mark it as **problem**. If the system cannot decide, it will be left for the technician to decide later.

To determine the threshold numbers for accepting or rejecting an iris identity claim, the matching algorithm needs to be test against a large set of data. This leads to another biometrics workflow that is used to study matching algorithms.

6.3.3 BXGrid All-Pairs

A typical All-Pairs workflow includes selecting a set of irises, transforming all irises into template format, then applying a comparison function to any pair of templates. Figure 6.11 represents an All-Pairs workflow.

Figure 6.12 show an example of an All-Pairs result matrix. Applying All-Pairs using function F returns Matrix M. Each comparison usually takes a fraction of a

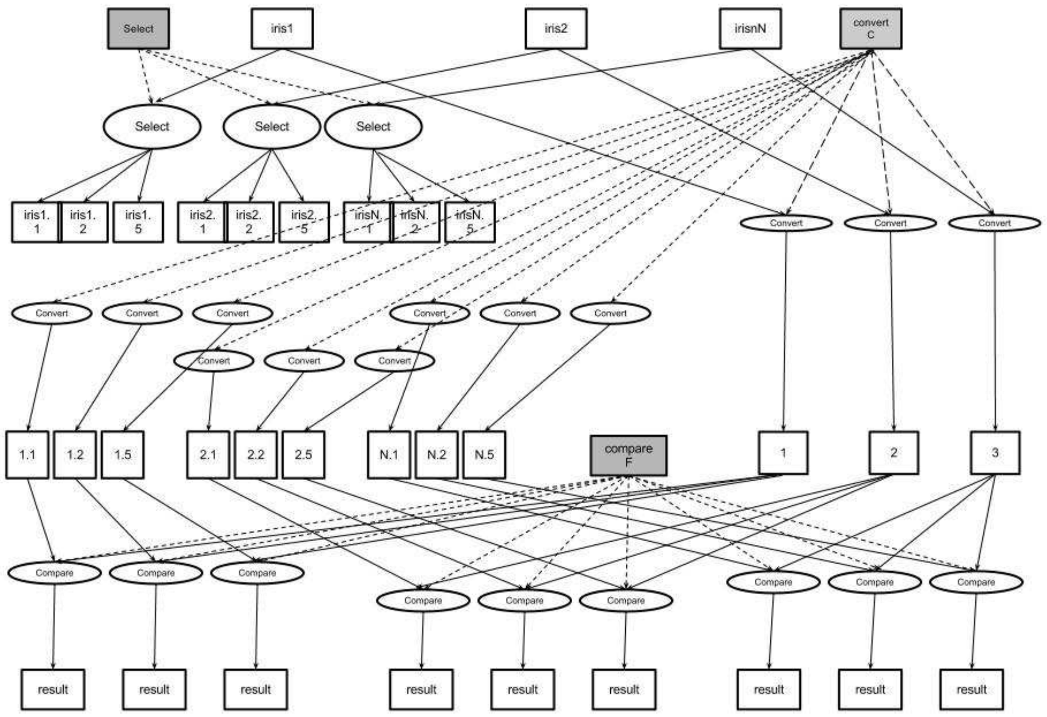


Figure 6.10. Auto Validation Workflow

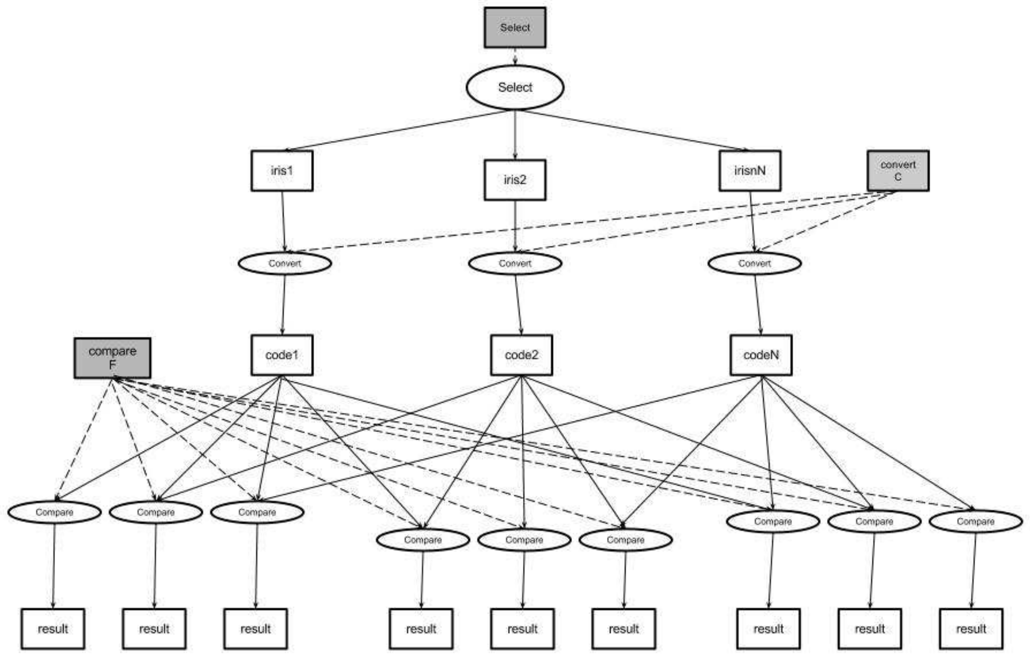


Figure 6.11. All-Pairs Workflow

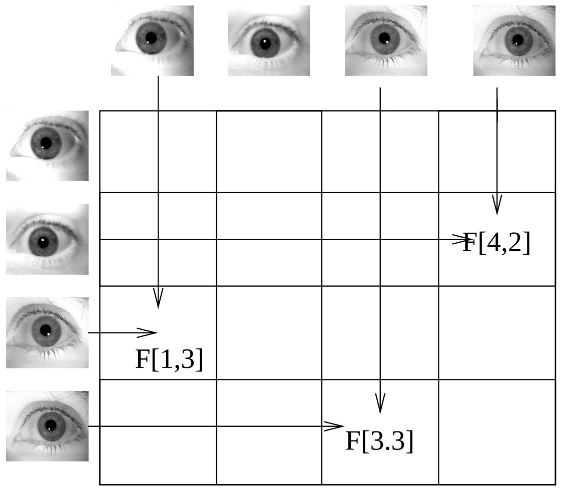


Figure 6.12: All-Pairs Result Matrix M

second to complete. However, applying All-Pairs to dataset of 100 or 1,000 irises needs 10,000 or 1,000,000 comparison respectively. The number of comparisons can grow out of hand very quickly and a single CPU will take hours, even days to finish the workload sequentially.

The CCL has developed two applications to help with running All-Pairs workloads: All-Pairs Multicore [81] and All-Pairs Master [43]. All-Pairs Multicore runs All-Pairs workload locally but uses multiple processes to execute a number of comparisons in parallel. The number of processes running concurrently is usually the number of cores available in the system. All-Pairs Master partitions the result matrix in to a number of sub-matrices, each sub-matrix represents a task. Tasks then can be sent and executed remotely using a number of systems: Work Queue, Condor, and SGE.

Figure 6.13 shows the runtime of an All-Pairs comparison for a All-Pairs Local, All-Pairs Multicore and All-Pairs Master. All-Pairs Local quickly became unfeasible to run because of a long runtime. All-Pairs Multicore performed slightly better. All-Pairs Master did best and finished a 10,000 X 10,000 All-Pairs work-

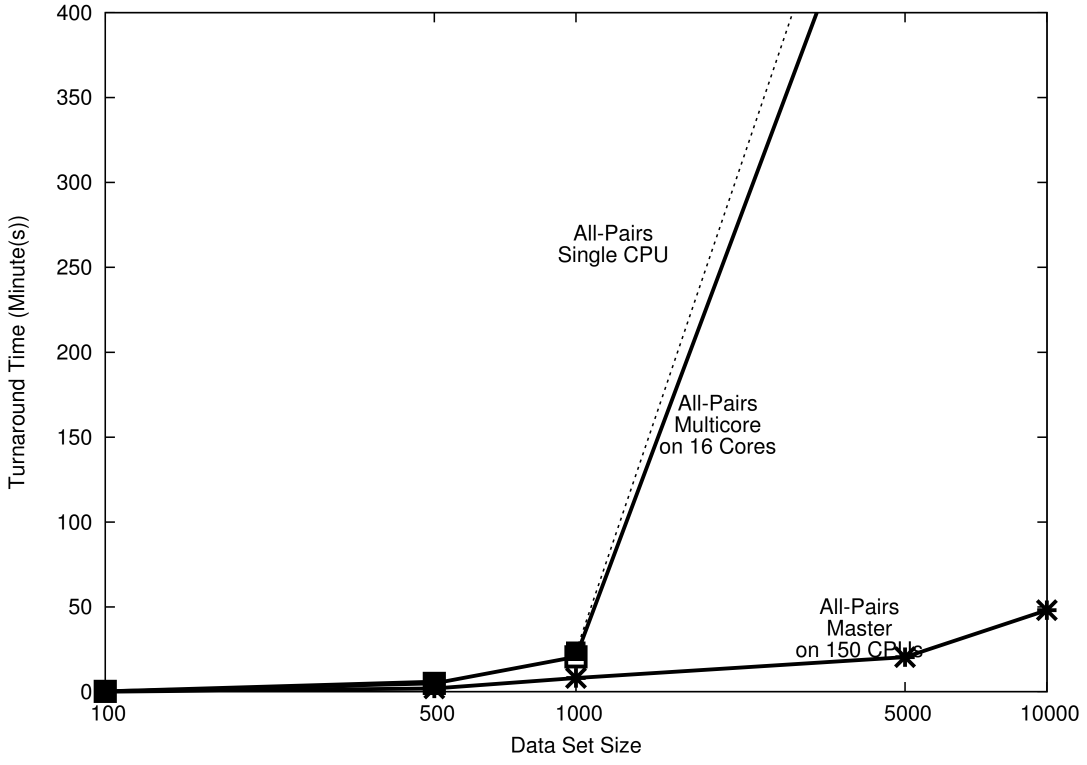


Figure 6.13: All-Pairs runtime(only comparison stage)

load in hours instead of days. The largest All-Pairs experiment was run for 100,000 X 100,000 irises and it took almost 3 days to finish. Figure 6.14 was generated by Condor Log Analyzer [69]. It shows the progress of the 100,000 All-Pairs experiment. Although we requested 500 workers for this workload, the maximum number of jobs running at once was around 300. The number of running jobs fluctuated during the experiment. This is because in a campus grid environment, the resource is allocated based on priority and availability. The total CPU time is 501 hours with 461 hours of Goodput and 40 hours of Badput. Goodput is the time allocation when a remote job spends executing and producing output. Badput is the time remote job spends executing but not producing any output due to application error or early termination.

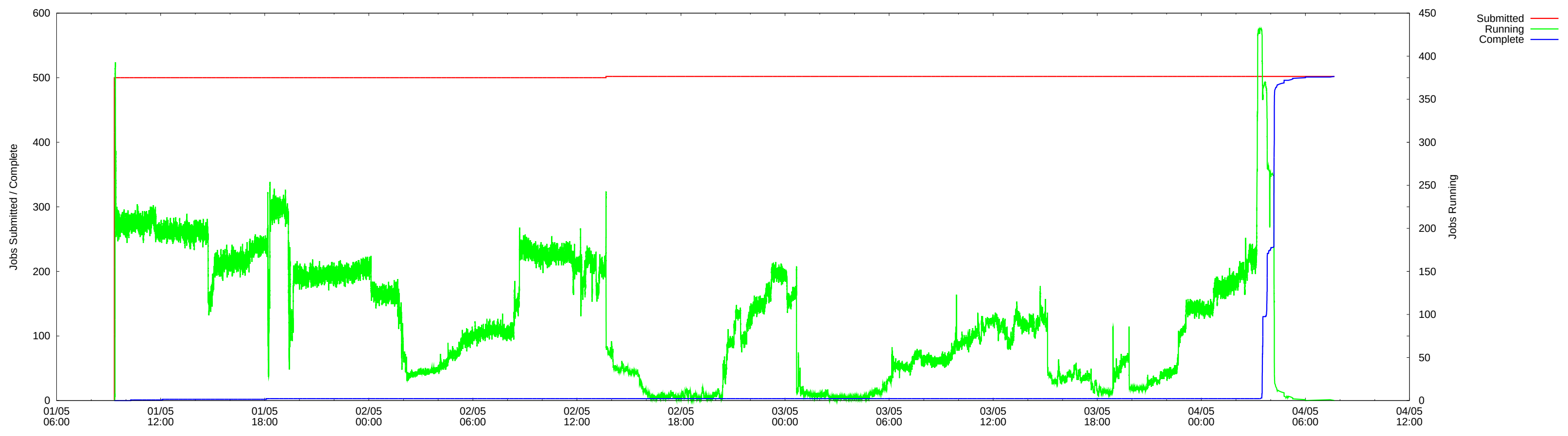


Figure 6.14: 100Kx100K All-Pairs experiment timeline

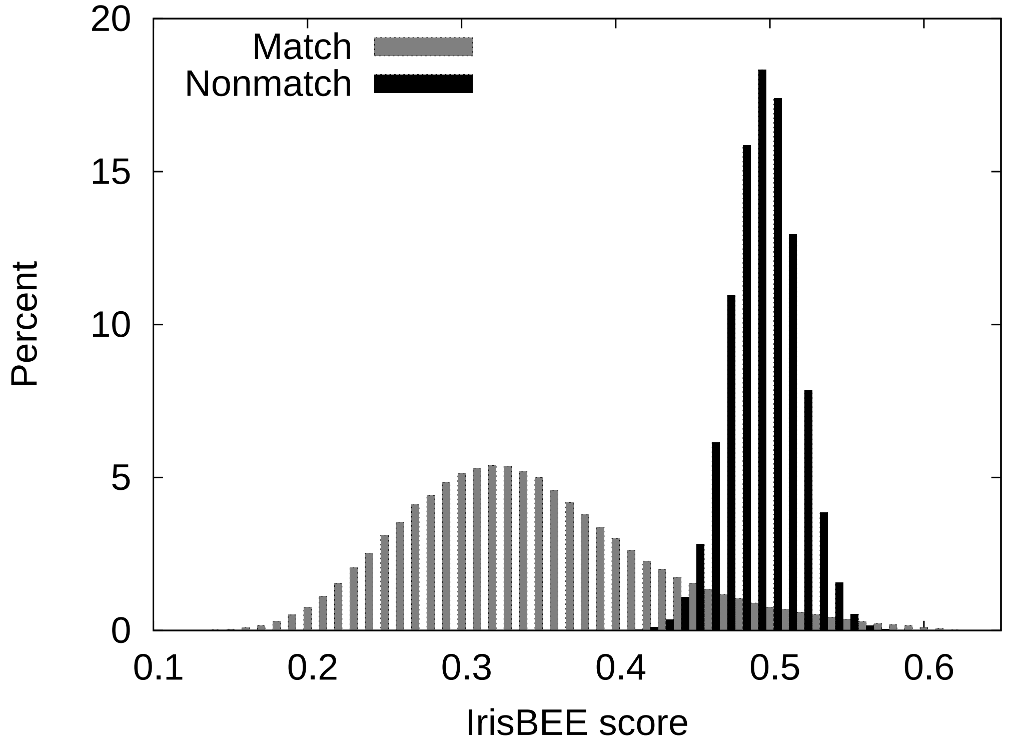


Figure 6.15: Score distribution of 10Kx10K All-Pairs experiment using irisBEE function

With the result matrix from All-Pairs experiments, we can apply Analyze abstraction and draw the conclusion about the threshold numbers.

Question: Given a matcher, how do I pick a threshold for accepting and rejecting identity claim?

```
S  = Select(D)
S' = Transform(S,C)
M  = All-Pairs(S',F)
V = Analyze(M)
```

Figure 6.15 shows a histogram after analyzing the result of an All-Pairs run on 10,000 irises. A match is a comparison between two images of the same iris

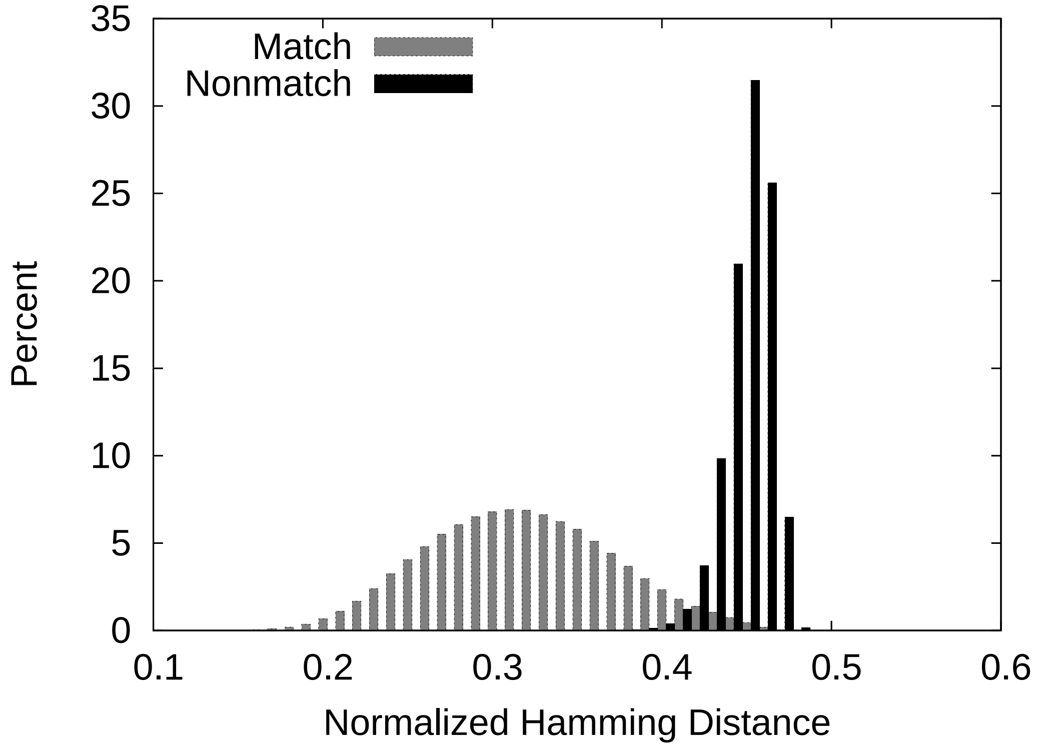


Figure 6.16: Score distribution of 58Kx58K All-Pairs experiment using Hamming distance function[43]

from the same person while a non-match is a comparison between two images of different irises. If there was a perfect matching algorithm, the score of a match comparison would be lesser than the score of a non-match comparison. This particular All-Pairs experiment used irisBEE baseline function [38]. From Figure 6.15, one would conclude that 0.4 is the threshold number to accept a match. However, because there are a number of scores from matches that are greater than the score for non-matches, it is not safe to pick a threshold for rejecting a match.

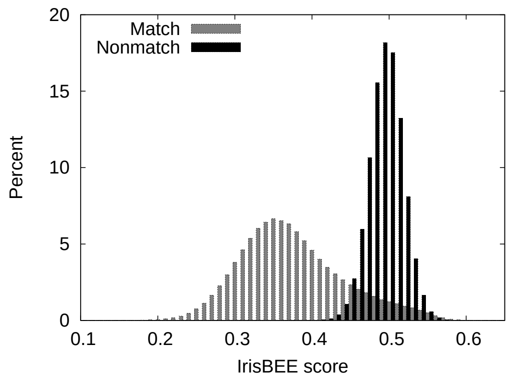
Figure 6.16 shows a histogram after analyzing the result of an All-Pairs run on 58,639 irises [43] from the ICE 2006 [45]. Iris images were first transformed to iris codes [14]. Then, the Hamming distance function [17] was used to compare two iris codes. From Figure 6.16, one would pick 0.4 and 0.475 as the accept and reject threshold number respectively. Thus for the auto-validation workload using Hamming distance function, any score between 0 and 0.4 would yield an accept, any score greater than 0.475 would result in a reject. Any score between 0.4 and 0.475 would be left alone for the technician to decide later.

Beside evaluating comparison algorithm to pick threshold scores, the All-Pairs abstraction and workflow can also explore more interesting biometrics research questions.

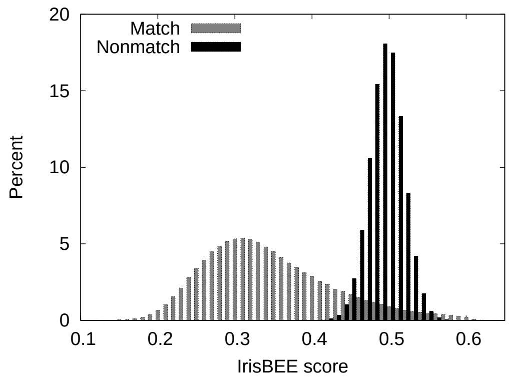
Question: Does matching function M have a demographic bias? To answer this, compute the quality of its matches across several different demographics:

```
foreach demographic D {
  S = Select(D)
  Q[D] = Analyze(All-Pairs(Transform(S,F),M))
}
```

Figure 6.17(a) and Figure 6.17(b) show the result of All-Pairs experiments for Asian subjects and White subjects respectively. The comparison function is the irisBEE baseline. The curves for non-match comparison are similar for both experiments. The score for match irises tilts more to the left. However, there are more matches with a bigger score comparing to non-matches' scores.



(a) Score distribution of Asian subjects' irises



(b) Score distribution of White subjects' irises

Figure 6.17: Histogram of All-Pairs experiments

CHAPTER 7

CONCLUSION

We have showed that ROARS is capable of storing hundreds of thousands of data objects with attached metadata. ROARS provides scalable data access and fast metadata query abilities. ROARS is also robust, fault-tolerant and can handle frequent hardware failure gracefully. Additionally, ROARS can facilitate large scale experiments using abstractions and distributed workflows.

7.1 Impact

The impact of BXGrid on biometrics research activity at Notre Dame has been significant and positive. It has enabled the development of workflows for ingestion, validation, and enrollment that did not exist before BXGrid (all earlier data set constructions were done by hand, by different people, and yielded unstructured piles of customized scripts with variable quality and accuracy). Biometrics group members are not forced to fret about the nuts and bolts of data management as frequently, and can access and use data with the assurance that quality checks have been performed.

7.2 Lessons Learned

Like many engineering projects, ROARS 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. In this section, We will revisit some of the challenges we faced during the development process given the dynamics of distributed environment. The lessons we learned may become useful insights for future projects.

Lesson 1: Get a prototype running right away. It is essential to have working system even if it is only partially working. Having a working system is helpful in many ways. First of all, it takes the system out of its conception to real hardware, real software, and real data. The system is no longer just a blueprint on paper. 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. The prototype system helped us discover our mistakes, pointed us to a right direction before it was too late. 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. However, once we ingested BXGrid with daily collected data, the system became more than just a archival. We understand that other people depend on BXGrid and use BXGrid in their daily research activities. Although the system was still under experimental, we knew that users come to BXGrid with certain expectations. They expect BXGrid to work. Because of that we worked hard and diligently to keep the system operating as smoothly as possible. Working with real scientific data also gets us to understand the data better. This kind of valuable knowledge helps with making the right design decision later on.

Lesson 3: Work closely with your users. Each group brought to the project different experience, terminology, and expectations. By talking to each other, we not only minimize confusion but also re-enforce what we have learned. Users' input is very important, because after all, we build the system for the users, not for us. Although what users want is not always what we can accomplish, healthy discussion is very essential to the success of a project. Users also play a very important role in identifying and reporting bugs. There are bugs that we did not anticipate during the design, implementation, and test process which the users did find and report. Users' contribution to the project does not stop there, their encouragement and thoughtfulness proves to be unmeasurable to the progress of the project.

Lesson 4: 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 5: 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 higher level of privilege, are logged in a distinct area of the system, and do not re-use unique identifiers.

7.3 Future Work

The power of ROARS is not only about managing, exporting data, but also about driving large scale experiments using current scientific abstractions and distributed workflows. The next step is to help researchers analyze and share results in a collaborative environments. Experiments should be re-run easily to confirm the results. Results should be rendered and presented back to the user for visualization. **Share** is the abstraction that takes ROARS to that direction.

Share: ROARS should store 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 opera-

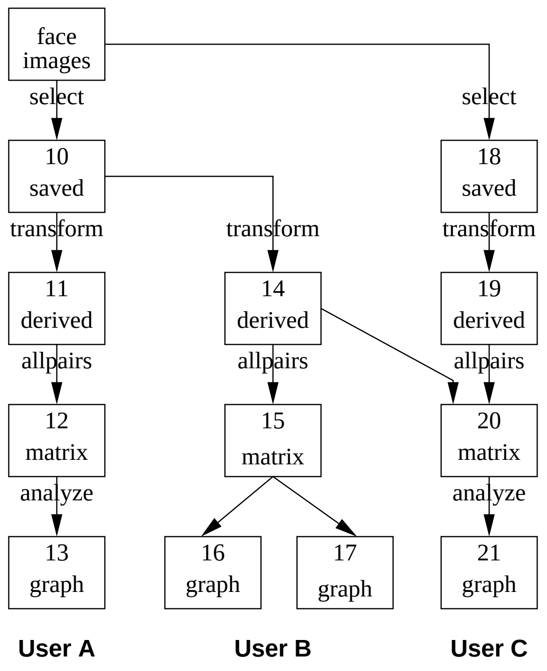


Figure 7.1. Sharing Datasets for Cooperative Discovery

tions (Select, Transform, All-Pairs, and Analyze). Figure 7.1 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 a histogram 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) archive, but apply the same function and produce new results. In this way, experiments can be precisely reproduced and compared.

BIBLIOGRAPHY

1. Filesystem in user space. http://sourceforge.net/projects/fuse.

2. *Solaris ZFS Administration Guide*. Sun Microsystems, Santa Clara, CA (May 1996).

3. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman (2002).

4. M. Albrecht, P. Donnelly, P. Bui and D. Thain, Makeflow: A Portable Abstraction for Cluster, Cloud, and Grid Computing. In *Technical Report CUCS-035-95*.

5. 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).

6. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/ (2009).

7. T. Anderson, M. Dahlin, J. Neefe, D. Pat-terson, D. Roselli and R. Wang, Serverless network file systems. In *ACM Symposium on Operating System Principles* (Dec 1995).

8. S. Baker, K. Bowyer and P. Flynn, Empirical Evidence for Correct Iris Match Score Degradation with Increased Time-Lapse between Gallery and Probe Matches. In *Proceedings of International Conference on Biometrics 2009*, pages 1170–1179 (June 2009).

9. C. Baru, R. Moore, A. Rajasekar and M. Wan, The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada (1998).

10. S. Best and D. Kleikamp, JFS Layout. In *IBM http://jfs.sourceforge.net/project/pub/jfslayout.pdf*.

11. J. Bonwick, M. Ahrens, V. Henson, M. Maybee and M. Shellenbaum, The zettabyte file system. In *Technical Report - Sun Microsystems*.

12. J. Bonwick, M. Ahrens, V. Henson, M. Maybee and M. Shellenbaum, The zettabyte file system. In *Technical Report, Sun Microsystems* (2003).

13. D. Borthakur, HDFS Architecture Guide. In *HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfs_design.pdf*.

14. K. Bowyer, K. Hollingsworth and P. Flynn, Image understanding for iris biometrics: A survey. *Computer Vision and Image Understanding*, 110(2): 281–307 (2007).

15. P. Bui, L. Yu and D. Thain, Weaver: Integrating Distributed Computing Abstractions into Scientific Workflows using Python. In *Challenges of Large Applications in Distributed Environments at ACM HPDC 2010* (2010).

16. R. Card, T. Ts'o and S. Tweedie, Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*.

17. J. Daugman, How iris recognition works. In *University of Cambridge, The Computer Laboratory, Cambridge CB2 3QG, U.K*.

18. J. Daugman, How Iris Recognition Works. *IEEE Trans. on Circuits and Systems for Video Technology*, 14(1): 21–30 (2004).

19. J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation* (2004).

20. 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).

21. B. Devlin, *Data Warehouse: From Architecture to Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996).

22. J. J. Dongarra and D. W. Walker, MPI: A standard message passing interface. *Supercomputer*, pages 56–68 (January 1996).

23. P. Donnelly and D. Thain, Fine-Grained Access Control in the Chirp Distributed File System. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (2012).

24. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, Hypertext transfer protocol (HTTP). Internet Engineering Task Force Request for Comments (RFC) 2616 (June 1999).

25. J. G. Fletcher, An arithmetic checksum for serial transmissions. In *IEEE Transactions on Communications* (1982).

26. A. S. Foundation, The Apache CouchDB project. In *http://couchdb.apache.org* (2012).

27. S. Ghemawat, H. Gobioff and S. Leung, The Google filesystem. In *ACM Symposium on Operating Systems Principles* (2003).

28. 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).

29. Hadoop. http://hadoop.apache.org/ (2007).

30. A. Holupirek, C. Grn and M. H. Scholl, BaseX & DeepFS joint storage for filesystem and database. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (2009).

31. 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).

32. M. Ivanova, N. Nes, R. Goncalves and M. Kersten, Monetdb/sql meets skyserver: the challenges of a scientific database. *Scientific and Statistical Database Management, International Conference on*, 0: 13 (2007).

33. 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).

34. M. K. Johnson, Whitepaper: Red Hat's new journaling file system: ext3 (2011).

35. O. Kirch, Why NFS Sucks. In *Proceedings of the Linux Symposium* (2006).

36. N. Leavitt, Will nosql databases live up to their promise? *Computer*, 43(2): 12–14 (February 2010).

37. J. Maccormick, N. Murphy, V. Ramasubramanian, U. Weder and J. Yang, Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions on Storage*, 4(1) (2009).

38. L. Masek, Recognition of human iris patterns for biometric identi.cation. In *Technical Report, School of Computer Science and Software Engineering, The University of Western Australia, 2003*.

39. A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas and L. Vivier, The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2 (June 2007).

40. P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn and C. Goble, Taverna, reloaded. 6187: 471–481 (2010).

41. MongoDB, GridFS Specification. In *http://www.mongodb.org* (2012).

42. C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn and D. Thain, All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(1): 33–46 (2010).

43. C. Moretti, J. Bulosan, D. Thain and P. Flynn, All-Pairs: An Abstraction for Data Intensive Cloud Computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11 (2008).

44. MySQL: The world's most popular open source database. http://www.mysql.com/ (2012).

45. National Insitute of Standards and Technology, Iris challenge evaluation data http://iris.nist.gov/ice/ (accessed Apr 2008).

46. 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).

47. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat and P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17): 3045–3054 (2004).

48. 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).

49. E. Pinheiro, W.-D. Weber and L. A. Barroso, Failure trends in a large disk drive population. In *USENIX File and Storage Technologies* (2007).

50. E. Plugge, T. Hawkins and P. Membrey, The definitive guide to MongoDB: the noSQL database for cloud and desktop computing. In *Apress, Berkely, CA, USA, 1st edition* (2010).

51. J. Postel, FTP: File transfer protocol specification. Internet Engineering Task Force Request for Comments (RFC) 765 (June 1980).

52. N. Ratha and R. Bolle, *Automatic Fingerprint Recognition Systems*. Springer (2004).

53. H. Reiser, ReiserFS. In *www.namesys.com, 2004.*.

54. E. Riedel, G. A. Gibson and C. Faloutsos, Active storage for large scale data mining and multimedia. In *Very Large Databases (VLDB)* (1998).

55. D. S. Rosenthal, Lockss: Lots of copies keep stuff safe. In *NIST Digital Preservation Interoperability Framework Workshop* (2010).

56. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, Design and implementation of the Sun network filesystem. In *USENIX Summer Technical Conference*, pages 119–130 (1985).

57. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, Design and Implementation of the Sun Network Filesystem. In *Proceedings of USENIX 1985 Summer Conference*, pages 119–130, Portland OR (USA) (1985).

58. B. Schroeder and G. A. Gibson, Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *USENIX File and Storage Technologies* (2007).

59. E. Sciore, SimpleDB: a simple java-based multiuser syst for teaching database internals. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education* (2007).

60. 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).

61. Y. L. Simmhan, B. Plale and D. Gannon, A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3): 31–36 (September 2005).

62. M. Spasojevic and M. Satyanarayanan, An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2) (May 1996).

63. E. Stolte, C. von Praun, G.Alonso and T. Gross, Scientific data repositories . designing for a moving target. In *SIGMOD* (2003).

64. M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger and S. B. Zdonik, Requirements for science data bases and scidb. In *CIDR*, www.crdrdb.org (2009).

65. M. Stonebraker, J. F. T and J. Dozier, An overview of the sequoia 2000 project. In *In Proceedings of the Third International Symposium on Large Spatial Databases*, pages 397–412 (1992).

66. 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).

67. O. Tatebe, N. Soda, Y. Morita, S. Matsuoka and S. Sekiguchi, Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Computing in High Energy Physics (CHEP)* (September 2004).

68. D. Thain, Identity Boxing: A New Technique for Consistent Global Identity. In *IEEE/ACM Supercomputing*, pages 51–61 (2005).

69. D. Thain, D. Cieslak and N. Chawla, Condor Log Analyzer. In *http://condorlog.cse.nd.edu* (2009).

70. D. Thain and M. Livny, Parrot: An Application Environment for Data-Intensive Computing. *Scalable Computing: Practice and Experience*, 6(3): 9–18 (2005).

71. D. Thain and C. Moretti, Abstractions for Cloud Computing with Condor. In S. Ahson and M. Ilyas, editors, *Cloud Computing and Software Services: Theory and Techniques*, pages 153–171, CRC Press (2010).

72. D. Thain, C. Moretti and J. Hemmes, Chirp: A Practical Global Filesystem for Cluster and Grid Computing. *Journal of Grid Computing*, 7(1): 51–72 (2009).

73. 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).

74. T. Y. Ts'o, Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *2002 FREENIX Track Technical Program*.

75. S. Tweedie, Journaling the Linux ext2fs Filesystem. In *Proceedings of the 4th Annual LinuxExpo, Durham, NC*.

76. S. Tweedie, Presentation on EXT3 Journaling Filesystem. In *The Ottawa Linux Symposium 2000* (July 2000).

77. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski and A. Barros, Workflow patterns. *Distributed and Parallel Databases*, 14: 5–51 (2003).

78. Vertica. http://www.vertica.com/ (2009).

79. 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).

80. S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long and C. Maltzahn, Ceph: A scalable, high-performance distributed file system. In *USENIX Operating Systems Design and Implementation* (2006).

81. L. Yu, C. Moretti, S. Emrich, K. Judd and D. Thain, Harnessing Parallelism in Multicore Clusters with the All-Pairs and Wavefront Abstractions. In *IEEE High Performance Distributed Computing*, pages 1–10 (2009).

82. L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd and D. Thain, Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions. *Journal of Cluster Computing*, 13(3): 243–256 (2010).

83. W. Zhao, R. Chellappa, P. Phillips and A. Rosenfeld, Face Recognition: A Literature Survey. *ACM Computing Surveys*, 34(4): 299–458 (2003).

84. 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).

This document was prepared & typeset with $\LaTeX\,2_\varepsilon$, and formatted with NDdiss2_ε *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*