Distrib Parallel Databases (2012) 30:325–350
DOI 10.1007/s10619-012-7103-5

ROARS: a robust object archival system for data intensive scientific computing

Hoang Bui · Peter Bui · Patrick Flynn · Douglas Thain

Published online: 14 August 2012

Abstract 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, scalable storage and efficient rich metadata queries for scientific applications. In this paper, we present the design and implementation of ROARS, focusing primarily on the challenge of maintaining data integrity across long time scales. We evaluate the performance of ROARS on a storage cluster, comparing to the Hadoop distributed file system and a centralized file server. We observe that ROARS has read and write performance that scales with the number of storage nodes, and integrity checking that scales with the size of the largest node. We demonstrate the ability of ROARS to function correctly through multiple system failures and reconfigurations. 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.

Keywords Distributed storage · Distributed system · Archive system

1 Introduction

Recent advances in digital technologies now make it possible for an individual or a small group to create and maintain enormous amounts of data. "Ordinary" researchers in all branches of science operate cameras, digital detectors, and computer simulations that can generate new data as fast as the researcher can pose a hypothesis. This

Communicated by Judy Qiu and Dennis Gannon.

H. Bui (✉) · P. Bui · P. Flynn · D. Thain
University of Notre Dame, 384 Patrick Hall, Notre Dame, IN 46556, USA
e-mail: hbui@nd.edu

increase in the production of data allows the individual to carry out complex studies that were previously only possible with a large staff of lab technicians, computer operators, and system administrators. Of course, such problems are not limited to science. A similar discussion applies to a digital library, to a paperless business, or to a thinly staffed internet startup that finds sudden success.

Unfortunately, this 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.

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 query, 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 it is not highly optimized for this task [25]. 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. However, as a filesystem becomes larger, querying, sorting, and searching can only be done efficiently if they match the chosen parallel structure. As an enterprise 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.

To address this mismatch, we have created ROARS (Rich Object ARchival System), a long-term data archive that combines some features of both the filesystem and database models, while eliminating some of the dangerous flexibility of each. Although there exist a number of designs for scalable storage [3, 4, 12, 13, 15, 30, 36] ROARS occupies an unexplored design point that combines several unusual features that together provide a powerful, scalable, manageable scientific data storage system:

– *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.
– *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.
– *Materialized filesystem views*. Rather than impose a single filesystem hierarchy from the beginning, fast queries may be used to generate materialized 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.
– *Transparent, incremental management*. ROARS does not need to be taken offline even briefly in order to perform an integrity check, add or decommission servers, or to migrate 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.

– *Failure independence*. Each object storage node in the system can fail or even be destroyed independently 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 can be reconstructed by a parallel scan of the object storage.

In our previous work on BXGrid [8], we created a discipline specific data archive tightly integrated with a web portal for biometrics research. ROARS is our "second version" of this concept, which has been decoupled from biometrics, generalized to an abstract data model, and expanded in the areas of execution, management, and fault tolerance. This article is an extension of the work we presented at the ACM Workshop on Data Intensive Distributed Computing in 2010 [7].

This paper is organized as follows. In Sect. 2, we present the abstract data model and user interface to ROARS. In Sect. 3, we describe our implementation of ROARS using a relational database and a storage cluster. In Sect. 4, an operational and performance evaluation of ROARS is presented. In Sect. 5, We discuss our real life experience using ROARS for a biometrics repository. In Sect. 6, we compare ROARS to other scalable storage systems. We conclude with future issues to explore.

2 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.

2.1 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` or `PDF`, meaning that the system does not extract any information from the file other than the basic filesystem attributes.
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 metadata record here:

```
NAME        TYPE    VALUE       OWNER  TIME
recordingid string  nd3R22829   pflynn 1257373461
ingested    date    08/03/2010  pflynn 1257373461
subjectid   string  nd1S04388   pflynn 1257373461
```

```
comment      text    Spring Co.  pflynn 1257373461
state        string  problem     dthain 1254049876
problemtype number   34          dthain 1254049876
state        string  fixed       hbui   1254050851
```

In the example metadata record above, each tuple contains fields for `NAME`, `TYPE`, `VALUE`, which define the name of the object 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 properties 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 the `NAME`, `TYPE`, and `VALUE` fields, each metadata entry also contains a field for `OWNER` and `TIME`. This is to provide provenance information and complete history of the metadata. 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 is 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 data, 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 were.

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.

2.2 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:

```
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>
SCREEN  <coll> FROM  <dir>
```

The `IMPORT` operation loads a local directory containing objects and metadata into a specific collection in the repository. `QUERY` retrieves the metadata for each object matching the given expression. `EXPORT` retrieves both the data and metadata for each object matching the given expression, which are stored on the local disk as pairs of files. `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.

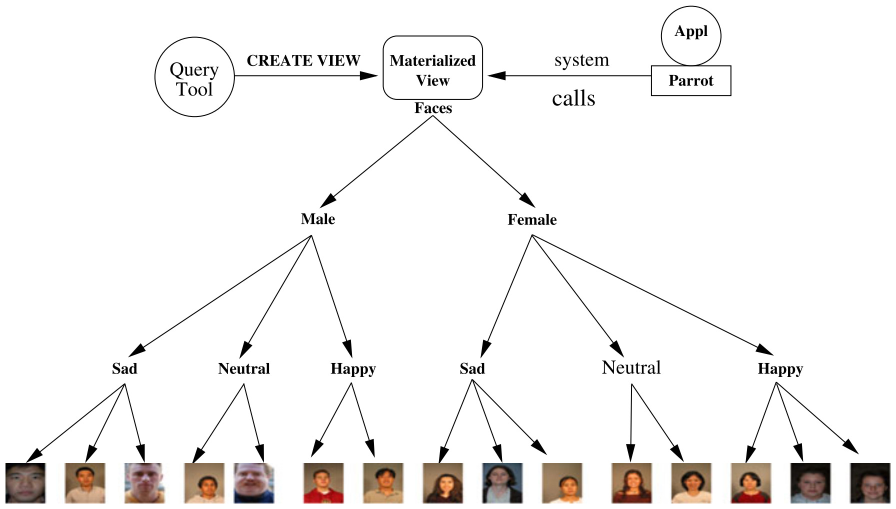


Fig. 1 Example materialized view

Before data is ingested into ROARS. It is recommended that users use `SCREEN` to examen the data and metadata for consistency purpose. The `SCREEN` operation scans a local directory containing objects and evaluates metadata before `IMPORT` is called. 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, and compares these information to the information from the 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 users about missing objects.

Ordinary applications may also view ROARS as a read-only filesystem, using either FUSE [1] (a user/kernel filesystem driver) or Parrot [31] (a ptrace-based interposition agent). Individual objects and their corresponding metadata can be accessed via their unique file identifiers using absolute paths:

```
/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 line tool, 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, Fig. 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. 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 an robust and scalable filesystem interface to the data objects is a distinguishing feature of ROARS. Rather than force users to implant 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.

3 Implementation

Figure 2 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 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 work flow normally used by research scientists. Moreover, the distinct advantage of using a database, which is 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 change 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.

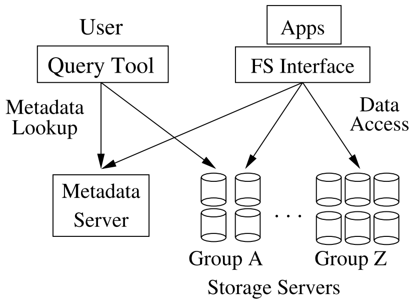


Fig. 2 ROARS architecture

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, which 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.

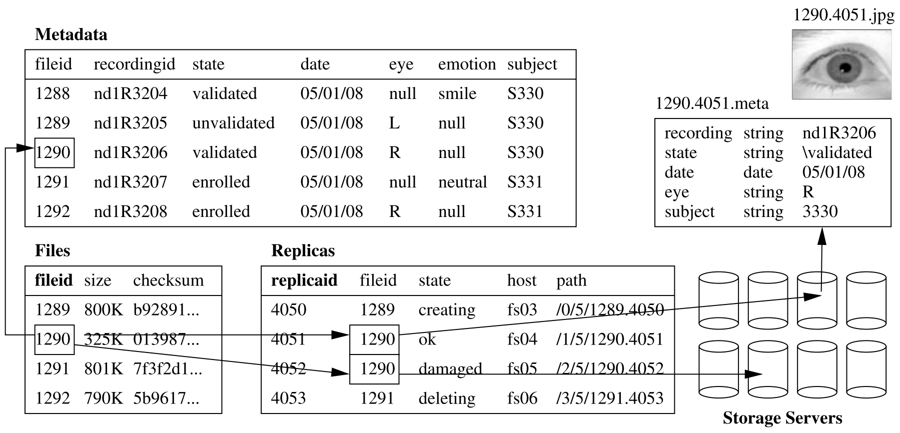
The hybrid design of ROARS takes the best aspects from both databases and distributed filesystems and combines them to provide rich metadata capabilities and robust scalable storage. To meet the storage requirement, ROARS replicates the data objects along with their associated metadata across multiple 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.

3.1 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 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 to not only 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 gives an example of the relationship between the metadata, file, and replica tables. In this configuration, each file is given an 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 and will be discussed in later subsections.

A few complications regarding the metadata are worth noting.



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

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

Fig. 3 MDS structure

First, the ROARS abstract data model has no schema, nor limits on field length. In our current implementation, we map this to a relational database table by adding new columns or expanding field widths as needed on the fly. 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. Future work may address improvements to this implementation. (As we show below, there is some penalty on imports, but read and query performance are sufficient.)

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 Sect. 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.

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 Storage nodes

We use the Chirp [32] 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 Kinesis system [18]. In such 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. 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 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 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 placed 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 and files with too few replicas, and incrementally migrates or replicates 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` and `REMOVE NODE` only interact with the MDS, it doesn’t 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, 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. Then, 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 size of the largest storage node. If problems exist, they are reported, 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.

3.4 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 LOCKSS [22].

Data integrity is achieved by checksumming all file objects on storage nodes, and recording this in the MDS. (Integrity of the MDS can be accomplished via internal hierarchical checksums of the tables, as in ZFS [5], although we have not yet implemented this.) 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 Fig. 4. The common concept is that major actions are

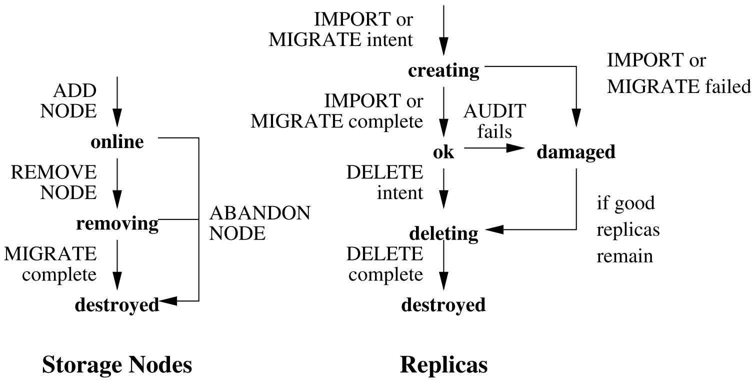


Fig. 4 State machines for storage nodes and replicas

accomplished via two phase commit: a state transition marks an intent to execute, then 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 gives 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.

4 Evaluation

To evaluate the performance and operational characteristics of ROARS, we deployed ROARS, HDFS, and a central fileserver on a testbed cluster consisting of 22 data nodes. They are servers with 32 GB RAM, twelve 2 TB 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 the 22 cluster data nodes, and an MDS with the MySQL database on the cluster head node. HDFS was deployed with 22 HDFS Datanodes running on the 22 cluster data nodes, and the HDFS Namenode running on the cluster head node. For HDFS, we kept the usual Hadoop defaults such as employing a 64 MB chunk size and a replication factor of three. The traditional centralized filesystem consists of a single Chirp server running on a cluster node. Applications access all three systems identically through the filesystem interface by using the Parrot interposition agent.

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. For these latter performance tests, we compare ROARS's performance to that of the traditional network server and Hadoop, which is an often cited alternative to distributed data archiving. At the end, we include operational results that demonstrate the operational robustness of ROARS.

4.1 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 5 shows the runtime of each of the key operations on 10,000 data objects total of 17.4 GB 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 1 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 Chap. 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 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

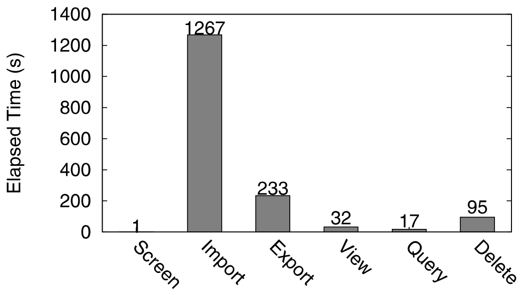


Fig. 5 Screen, import, export, view, query, delete performance 10,000 files 17.9 GB of data

Table 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

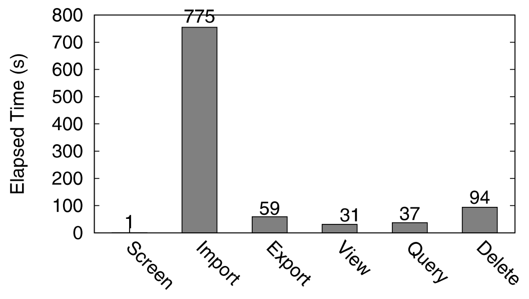


Fig. 6 Screen, import, export, view, query, delete performance, 10,000 files 3.0 GB of data

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` Fig. 6 shows the runtime of the same set of operations, this time with 10,000 small data objects of about 300 KB 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 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.

4.2 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 1 KB to 1 GB. 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 7 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 128 MB/s, and the maximum achievable by a TCP connection is closer to 100 MB/s, depending on the variant used. Both ROARS and HDFS achieve significantly less than that, due to the overheads of interacting with the central metadata server, and the creation of multiple file replicas. The overhead of interacting with the MDS is higher in ROARS, due to the multiple state transitions shown in Fig. 4. The overhead of creating replicas is higher in ROARS, because it transfers and verifies each replica separately, whereas HDFS 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 tradeoff to achieve high integrity transactions.

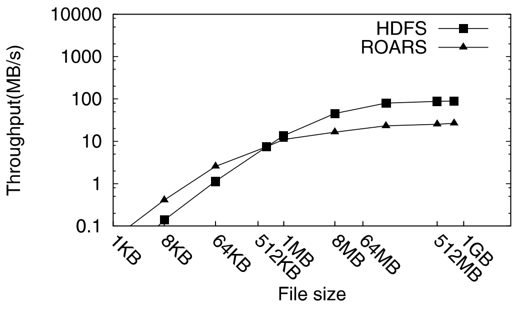


Fig. 7 Import performance

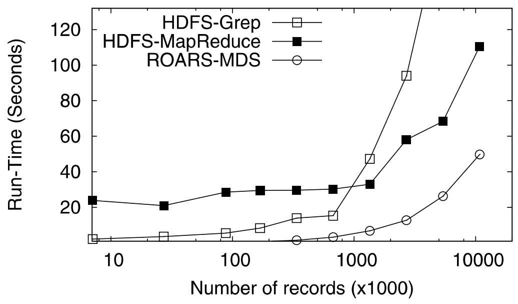


Fig. 8 Query performance

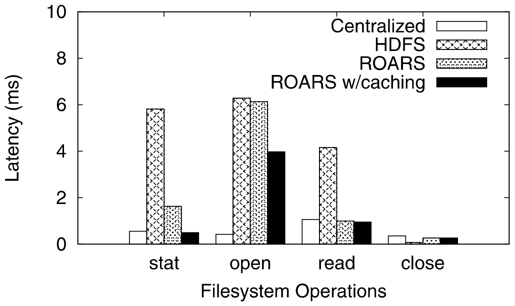
4.3 Metadata query

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 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 HDFS, 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 Sect. 2.1. For each object, the metadata takes up approximately 1.3 KB in storage. We queried the metadata by executing the custom script using MapReduce [9]. For ROARS, we queried the metadata using `QUERY` which internally uses the MySQL execution engine. We did this with indexing on and off to examine its effect on performance.

Figure 8 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 2.7 M 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

Fig. 9 Latency of filesystem operations



query had a high startup cost and thus was slower than the MDS. Furthermore, 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 345 M data objects (475 GB of data).

4.4 Filesystem access

ROARS provides a read-only filesystem interface for conventional applications, consisting primarily of the system calls `stat`, `open`, `read`, and `close`. HDFS 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 HDFS. 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 `stats`, `opens`, `reads`, and `closes` on a single file.

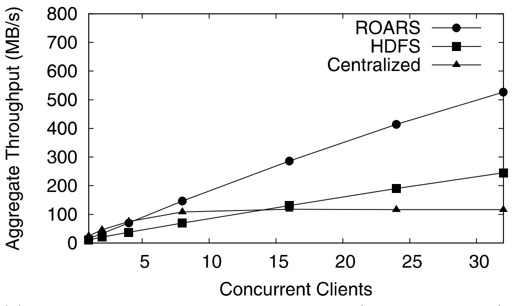
Figure 9 shows the latency of each operation on a centralized server, HDFS, and ROARS. As can be seen, ROARS provides comparable latency to the centralized server, and in the case of `stat`, `open`, and `read`, lower latency than HDFS. Since all file access also pass through Parrot, there is some interposition 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 HDFS. 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 HDFS due to HDFS start-up cost.

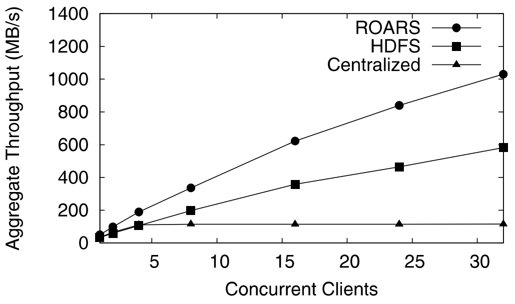
4.5 Concurrent access

To determine the scalability of ROARS in comparison to a traditional file server and HDFS, 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,

Fig. 10 Concurrent read performance



(a) **Concurrent Access Performance (10K x 320KB)**



(b) **Concurrent Access Performance (1K x 5MB)**

while for the traditional filesystem and HDFS, we exported the data directory to each of those systems. We ran our test program using the Condor [33] 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, 4 GB of RAM, and 750 GB 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. HDFS was deployed with 32 HDFS Datanodes running on the 32 cluster data nodes, and the HDFS Namenode running on the cluster head node.

Figure 10 shows the performance results of all three systems for both datasets. In Fig. 10(a), the clients read 10,000 320 KB files, while in Fig. 10(b) 1,000 5 MB files were read. In both graphs, the overall aggregate throughput for both HDFS 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 120 MB/s, which it reaches after 8 concurrent readers. ROARS and HDFS, 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 HDFS than working on smaller files.

While both ROARS and HDFS achieve improved aggregate performance over the traditional file server, ROARS outperforms HDFS 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 HDFS only reached 245.23 MB/s. For the larger test, ROARS reached 1030.94 MBS/s and HDFS 581.88 MB/s. There are several reasons for this difference. First, ROARS has less overhead in setting up the data transfers than HDFS as indicated in the micro-operations benchmarks. Such overhead limits the number of concurrent data transfers and thus aggregate throughput. Another cause for the performance difference is the behavior of the storage nodes. In HDFS, 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 HDFS data integrity is enforced throughout, even during reads.

4.6 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 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 11 shows the cost of computing checksums in both ROARS and HDFS. 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 HDFS block size (64 MB), ROARS begins to outperform HDFS 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 HDFS 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.

We compared the performance of an external sequential audit against a parallel/ active-storage audit on a production ROARS deployment of 90,000 files totalling 497 GB. The sequential implementation completed in *4.2 hours*, averaging 32.5 MB 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.

Fig. 11 Cost of calculating checksums

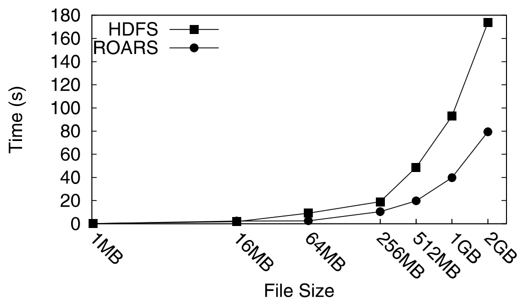
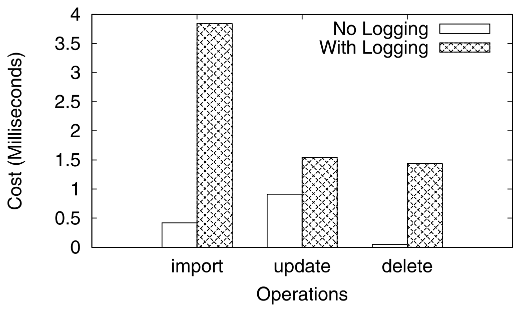


Fig. 12 Metadata logging



4.7 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 12 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 into a metadata log table. The performance penalty is about 40 percent. While import and delete result in multiple inserts into the log table.

4.8 Dynamic data migration

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

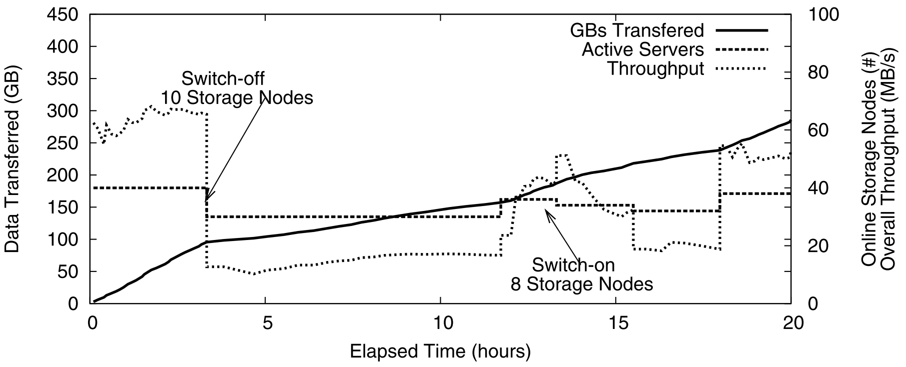


Fig. 13 Dynamic data migration

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 13 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.

5 Experience with ROARS

At the time of writing, ROARS has been in use as the archival service for a biometrics research group at the University of Notre Dame for over three years. We called the system BXGrid. 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.

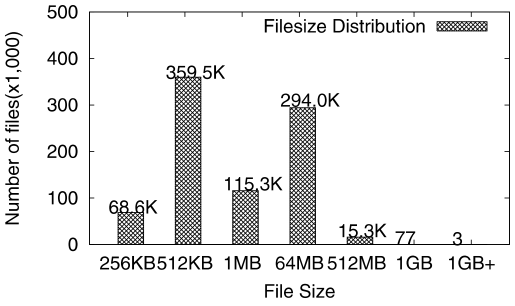


Fig. 14 Filesize distribution in BXGrid

BXGrid currently contains 853,004 recordings totalling 14.1 TB of data, spread across 40 storage nodes. Figure 14 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.

Approximately 60 GB 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. This data includes pictures and videos of faces and irises, along with 3D face scans. Each picture or video file is denoted by a *recording* which is associated with metadata such as: the subject in the picture, lighting conditions, camera used to take the picture. Additional metadata that must be kept internally for bookkeeping purposes are *ShotID*: original filename, and *BatchID*: unique number for a collection session.

The data model fits ROARS perfectly because the raw data never changes 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 (ex. 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 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 32 GB RAM, twelve 2 TB 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 15 shows the entire migration process. It took 40 hours to move approximate 5 TB to the new nodes.

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

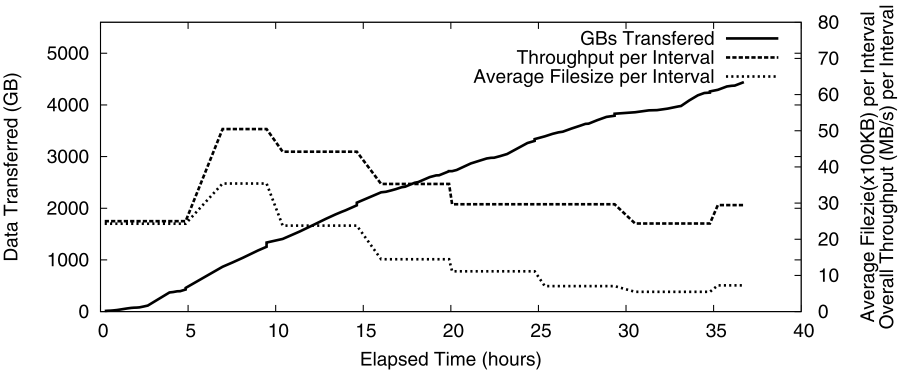


Fig. 15 BXGrid data migration to a new cluster

6 Related work

Our goal was to construct a scientific data repository that required both scalable fault-tolerant data storage, and efficient querying of the rich domain-specific metadata. Unfortunately, traditional filesystems and databases fail to meet both of these requirements. While most distributed filesystems provide scalable 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 work flow of scientific researchers.

6.1 Filesystems

In order to facilitate sharing of the scientific data, scientific researchers usually employ various network filesystems such as NFS [23] or AFS [15] to provide data distribution and concurrent access. To get scalable and fault tolerant data storage, scientists may look into distributed storage systems such as Ceph [36] or Hadoop [13]. Most of the data in these filesystems are organized into sets of directories and files along with associated metadata. Since some of these filesystems such as Ceph and Hadoop 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 a scalable, fault-tolerant data storage, current distributed storage systems adequately meet this requirement.

Google developed GFS [12] 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 64 MB, 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 64 MB 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 [6]. 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 [9] 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.

Although, ROARS storage organization is similar to the one used in the Google Filesystem [12], and Hadoop [13], 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.

6.2 Databases

The other common approach to managing scientific data is to go the route of projects such as the Sloan Digital Sky Survey [29]. 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 [16] or Vertica [34]. 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.

With the explosion of social networks in late 2000's there has been another evolution from traditional DBMS to NoSQL [17]. 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 [11, 14, 21, 24] 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 4 MB. In order to store large data objects, users will need to use GridFS [19] GridFS is system built on top of MongoDB. GridFS breaks up large files into chunks of 4 MB and stitches them back together per users' requests.

Database systems are not ideal for scientific data repositories because they do not fit into the work flow commonly used by scientific researchers. In projects such as the Sloan Digital Sky Survey and Sequoia 2000 [28], the scientific data is directly stored in database tables and the database system is used as an 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. [27] 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 [2] on distributed systems such as Condor [33]. 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 work flow 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 [20] which also utilizes 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 [10], while ROARS uses a distributed active storage cluster. Another example of a filesystem-database combination is HEDC [26]. HEDC is implemented on a single large enterprise-class machine rather than an array of storage nodes. iRODS [35] and its predecessor the Storage Resource Broker [4] 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.

7 Conclusion

We have described the overall design and implementation of ROARS, a archival system for scientific data with support for rich metadata operations. ROARS couples a database server and an array of storage nodes to provide users the ability to search data quickly, and to store large amounts of data while enabling high performance throughput for distributed applications. Through our experiments, ROARS has demonstrated the ability to scale up and perform as well as HDFS in most cases, and provide unique features such as transparent, incremental operation and failure independence.

Acknowledgements This work was supported by National Science Foundation grants CCF-06-21434, CNS-06-43229, and CNS-01-30839. This work is also supported by the Federal Bureau of Investigation, the Central Intelligence Agency, the Intelligence Advanced Research Projects Activity, the Biometrics Task Force, and the Technical Support Working Group through US Army contract W91CRB-08-C-0093.

References

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

2. Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. J. Mol. Biol. **3**(215), 403–410 (1990)
3. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/ (2009)
4. Baru, C., Moore, R., Rajasekar, A., Wan, M.: The SDSC storage resource broker. In: Proceedings of CASCON, Toronto, Canada (1998)
5. Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M.: The zettabyte file system. Technical Report, Sun Microsystems (2003)
6. Borthakur, D.: HDFS architecture guide. In: Hadoop Apache Project. http://hadoop.apache.org/common/docs/current/hdfs_design.pdf
7. Bui, H., Bui, P., Flynn, P., Thain, D.: ROARS: a scalable repository for data intensive scientific computing. In: The Third International Workshop on Data Intensive Distributed Computing at ACM HPDC 2010 (2010)
8. Bui, H., Kelly, M., Lyon, C., Pasquier, M., Thomas, D., Flynn, P., Thain, D.: Experience with BXGrid: a data repository and computing grid for biometrics research. J. Clust. Comput. **12**(4), 373 (2009)
9. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Operating Systems Design and Implementation (2004)
10. Dongarra, J.J., Walker, D.W.: MPI: a standard message passing interface. Supercomputer **12**, 56–68 (1996)
11. Foundation, A.S.: The apache CouchDB project. http://couchdb.apache.org (2012)
12. Ghemawat, S., Gobioff, H., Leung, S.: The Google filesystem. In: ACM Symposium on Operating Systems Principles (2003)
13. Hadoop. http://hadoop.apache.org/ (2007)
14. Holupirek, A., Grün, C., Scholl, M.H.: BaseX & DeepFS joint storage for filesystem and database. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (2009)
15. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M.: Scale and performance in a distributed file system. ACM Trans. Comput. Syst. **6**(1), 51–81 (1988)
16. Ivanova, M., Nes, N., Goncalves, R., Kersten, M.: Monetdb/sql meets skyserver: the challenges of a scientific database. In: International Conference on Scientific and Statistical Database Management, p. 13 (2007)
17. Leavitt, N.: Will nosql databases live up to their promise? Computer **43**(2), 12–14 (2010)
18. Maccormick, J., Murphy, N., Ramasubramanian, V., Weder, U., Yang, J.: Kinesis: a new approach to replica placement in distributed storage systems. ACM Transactions on Storage **4**(1), 1–28 (2009)
19. MongoDB. GridFS Specification. http://www.mongodb.org (2012)
20. No, J., Thakur, R., Choudhary:, A.: Integrating parallel file i/o and database support for high-performance scientific data management. In: IEEE High Performance Networking and Computing (2000)
21. Plugge, E., Hawkins, T., Membrey, P.: The definitive guide to MongoDB: the noSQL database for cloud and desktop computing. In: Apress, Berkley, CA, USA (2010). 1st edn.
22. Rosenthal, D.S.: Lockss: lots of copies keep stuff safe. In: NIST Digital Preservation Interoperability Framework Workshop (2010)
23. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun network filesystem. In: USENIX Summer Technical Conference, pp. 119–130 (1985)
24. Sciore, E.: SimpleDB: a simple java-based multiuser syst for teaching database internals. In: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (2007)
25. Searcs, R., Ingen, C.V., Gray, J.: 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)
26. Stolte, E., von Praun, C., Alonso, G., Gross, T.: Scientific data repositories: designing for a moving target. In: SIGMOD (2003)
27. Stonebraker, M., Becla, J., DeWitt, D.J., Lim, K.-T., Maier, D., Ratzesberger, O., Zdonik, S.B.: Requirements for science data bases and scidb. In: CIDR (2009). www.crdrdb.org
28. Stonebraker, M., Frew, J., Dozier, J.: An overview of the sequoia 2000 project. In: Proceedings of the Third International Symposium on Large Spatial Databases, pp. 397–412 (1992)
29. Szalay, A.S., Kunszt, P.Z., Thakar, A., Gray, J., Slutz, D.R.: Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. In: SIGMOD Conference (2000)
30. Tatebe, O., Soda, N., Morita, Y., Matsuoka, S., Sekiguchi, S.: Gfarm v2: a grid file system that supports high-performance distributed and parallel data computing. In: Computing in High Energy Physics (CHEP), September (2004)

31. Thain, D., Livny, M.: Parrot: an application environment for data-intensive computing. Scalable Comput., Pract. Exp. **6**(3), 9–18 (2005)
32. Thain, D., Moretti, C., Hemmes, J.: Chirp: a practical global filesystem for cluster and grid computing. J. Grid Comput. **7**(1), 51–72 (2009)
33. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In: Berman, F., Fox, G., Hey, T. (eds.) Grid Computing: Making the Global Infrastructure a Reality. Wiley, New York (2003)
34. Vertica. http://www.vertica.com/ (2009)
35. Wan, M., Moore, R., Schroeder, W.: A prototype rule-based distributed data management system rajasekar. In: HPDC Workshop on Next Generation Distributed Data Management, May (2006)
36. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: USENIX Operating Systems Design and Implementation (2006)