ROARS: A Scalable Repository for Data Intensive Scientific Computing

Hoang Bui, Peter Bui, Patrick Flynn, and Douglas Thain
University of Notre Dame

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 demonstrate that ROARS is capable of importing and exporting large quantities of data, migrating data to new storage nodes, providing robust fault tolerance, and generating materialized views based on metadata queries. Our experimental results demonstrate that ROARS' aggregate throughput scales with the number of concurrent clients while providing fault-tolerant data access. ROARS is currently being used to store 5.1TB of data in our local biometrics repository.

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 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 digital libraries, to 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DIDC 2010, Chicago, IL
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

The two canonical models for data storage – the filesystem and the database – are not well suited for supporting these kinds of applications. 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 advance schema design and system administration. A database can store large binary objects, but it is not highly optimized for this task [14]. 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 hierarchical 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), an online 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 [9, 7, 8, 25, 2, 3, 19] 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 [4], 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 paper is organized as follows. In section 2, we present the abstract data model and user interface to ROARS. In section 3, we describe our implementation of ROARS using a relational database and storage cluster. In section 4, an operational and performance evaluation of ROARS is presented. In section 5, 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 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 fileystem-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
subjectid   string  nd1S04388   pflynn 1257373461
state       string  problem     dthain 1254049876
problemtype integer 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. Because objects may have varying number and types of attributes and the user never specifies an exact specification of what objects should contain, this data model is schema-free. However, since scientific data tends to be semi-structured, the data model allows for the storage system to transparently group similar items into collections for efficiency and organizational purposes. Due to this regularity in the name and types of fields, and the ability to automatically group objects into collections, we consider the data model to be schema-implicit. The user never formally expresses the schema of the data objects, but an implicit one can be generated from the metadata records due to the semi-structured nature of the scientific data.

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 transactional 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>
```

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` removes data objects and the related metadata from the repository, and is usually invoked only after a failed `IMPORT`. Given constraints by users, ROARS finds all associated data objects, deletes them and removes the metadata entries.

Applications may also view ROARS as a read-only filesystem. Individual objects and their corresponding metadata

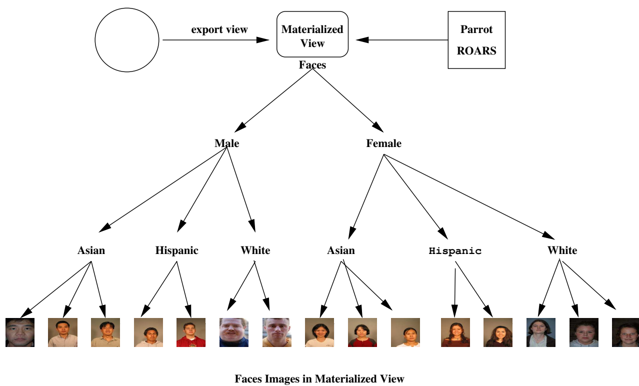


Figure 1: Example Materialized View

can be accessed via their unique file identifiers using absolute paths:

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

However, most users find it effective to access files using materialized views. Using the VIEW command above, a subset of the data repository can be queried, depositing a tree of named links onto the local file system. Each link is named according to the metadata of the containing object, and points to the absolute path of an item in the repository. For example, Figure 1 shows a view generated by the following command:

```
VIEW faces WHERE true AS "gender/race/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 this 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 work flow and applications.

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 Servers 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

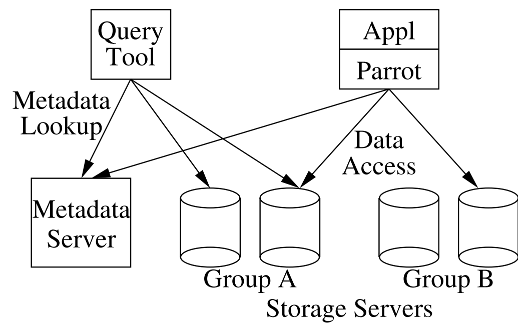


Figure 2: ROARS Architecture

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 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, 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.

Overall, this storage organization is similar to the one used in the Google Fileystem [7], and Hadoop [8], where simple

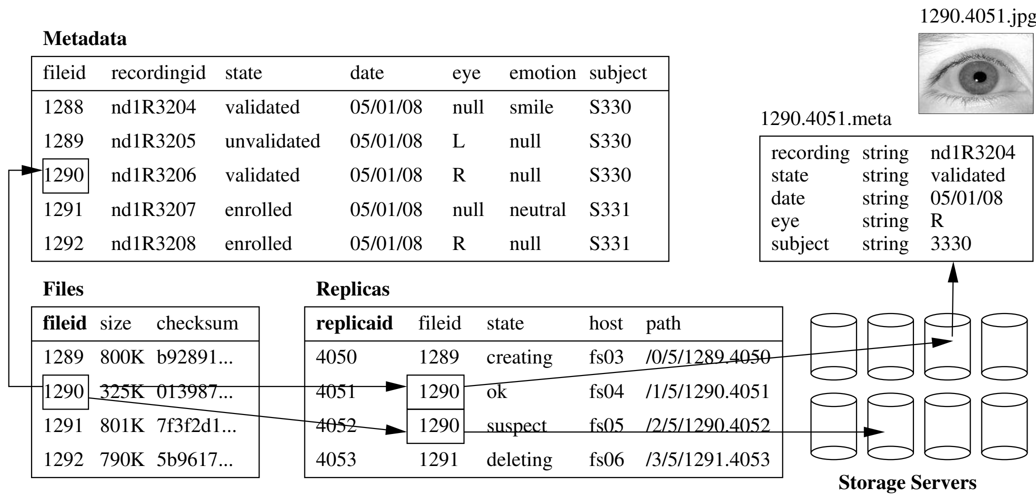


Figure 3: ROARS Metadata Structure

Data Nodes store raw data and a single Name Node maintains the metadata. Our 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.

3.1 Database Structure

In ROARS, the metadata is stored in three main database tables: a metadata table, a file table and a replica table. The metadata table obviously stores the domain-specific scientific metadata. Each entry in this table can have one or more pointers to a `fileid` in the file table. In the case where there is only metadata and no corresponding raw data, the entry would have a NULL `fileid`. 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. Entries in the this table represent ROARS inodes, and therefore have the following important file information: `fileid`, `size`, `checksum`, and `create time`. ROARS utilizes this information to not only keep track of files but also to emulate traditional UNIX system calls such as `stat`. For any given `fileid`, there can be multiple replica entries in the replica table. This third table keeps track of where the actual raw data files are stored. The structure of the replica table is very straightforward and includes the following fields: `fileid`, `host`, `path`, `state`, and `lastcheck`.

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.

3.2 Storage Nodes

ROARS utilizes an array of Storage Nodes running Chirp [21] for replicating data. These Storage Nodes are usually conventional machines with large local single disks organized in a compute cluster. These Nodes are grouped together based on locality into different Storage Groups, and given a `groupid`. During an `IMPORT`, ROARS makes a conscious decision to spread out replicas so that each Storage Group has a least one replica, thus providing a static form of load balancing. 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 distributed applications scalable, high throughput data access. 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. To facilitate determining where a certain data object resides, ROARS includes a `LOCATE` command that will find the closest copy of the data for a requesting application. If the application is running on the same Storage Node, then the data is already on the local node, and so no data transfer is needed.

3.3 Robustness

Due to the use of data object replication, ROARS is stable and has support for recovery mechanisms and fail-over.

By default, data is replicated across the Storage Nodes at least three times. During a read operation, if a replica is not reachable due to server outage or hardware failure, ROARS will randomly try another available replica after a user specified timeout (normally 30 seconds). As mentioned earlier, Storage Nodes are organized into groups based on their locality. When data is populated into the data repository, ROARS intelligently places the data to ensure that there is a replica in each server group. By spreading replicas to multiple groups, a systematic failure of a Storage group only has a minimal effect on ROARS operation and performance. Similar to read operations, write operations performed during `IMPORT` will randomly choose a server within a server group to write data. If the server is not responsive, another is chosen until the write is successful.

ROARS also ensures integrity of the data repository by tracking and comparing checksums of replicas. As a data file is ingested into ROARS, its checksum is calculated and recorded as a part of the data object's metadata. Read/write requests can internally check to make sure the replica's checksum matches the original data file. However, frequent checksum calls can reduce system performance, and so this integrity check is only performed during special data integrity management operations such as `AUDIT`. This command will scan the metadata database and perform checksums on the data objects and ensure the current checksums match the one recorded in the database. In the same process, the `AUDIT` command will also check the status of the Storage Nodes and perform any maintenance as necessary. Because of this ROARS gives integrity checking a broader meaning since it maintains integrity of the system as a whole, not simply single replicas.

ROARS' robust design also enables transparent and incremental management. Whenever a Storage Node needs to be taken offline or decommissioned, invoking `REMOVE` will delete all entries associated with that node from the replica table and update the metadata database in a transactional manner. To add new Storage Nodes, an `ADD_NODE` followed by `MIGRATE` will spawn the creation of replicas on the new Storage Nodes. ROARS takes advantage of the atomic transactions of the database server to manage these operations. Because of this use of the database as a transaction manager, these operations can be performed transparently and incrementally. For instance, an `AUDIT` can be schedule to run for only a few minutes at a time or during off peak hours. Even if it does not complete, due to the use of the database as a transactional log, it can continue where it left off the next time it is run. The same goes for operations such as `MIGRATE` or `REMOVE`. These commands can be paused, rescheduled, and restarted with out affecting the integrity of the system and without having to shutdown the whole repository.

This robustness further extends to the ability to provide failure independence. Since the data is stored as complete data files rather than striped across multiple Storage Nodes, the integrity of the data files is never effected by a sudden lost of data servers. Additionally, failure of one server does not affect the performance or behavior of the other Storage Nodes. Because the metadata is stored along side the data file replicas, ROARS is also capable of recovering from the lost of the Database Node which only serves as a persistent metadata cache. To perform this recovery, a parallel scan can be performed on the Storage Nodes to reconstruct the metadata cache. This is in contrast to systems such as Hadoop and Google Filesystem which take special care to replicate all of the state of the Name Node. In the case of a lost of the Name Node in this system, the layout and organization of the data can be completely lost since the data is striped across multiple servers. ROARS avoids this problem by storing complete discrete data files, and maintaining the metadata log next to these replicas on the Storage Nodes. This enables ROARS to robustly provide failure independence and a simple means of recovery.

4. EVALUATION

To evaluate the performance and operational characteristics of ROARS, we deployed a traditional network filesystem, Hadoop, and ROARS on a testbed cluster consisting of 32 data nodes and 1 separate head node. Each of these storage nodes is a commodity dual-core Intel 2.4 GHz machine, with 4GB of RAM and 750GB of disk, all connected via a single Gigabit Ethernet switch.

The traditional network filesystem was a single Chirp file server on one of the data nodes. For Hadoop, we configured the Hadoop Distributed Filesystem (HDFS) to use the 32 storage nodes as the HDFS Datanodes and the separate head node as the HDFS Namenode. We kept the usual Hadoop defaults such as employing a 64 MB block size for HDFS. Our ROARS configuration consisted of a dedicated metadata server running MySQL on the head server and 32 Chirp servers on the same data nodes as the Hadoop cluster. To provide our test software access to these storage systems, we utilized Parrot as a filesystem adaptor.

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 data management capabilities of ROARS.

4.1 Data Import

Before performing any data access experiments, we first tested the performance of importing large datasets into both Hadoop and ROARS. For this data import experiment, we divided our test into several sets of files. 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 copy the data from the local machine to HDFS. For ROARS, we used the `IMPORT` operation.

Figure 4 shows the data import performance for Hadoop and ROARS for several sets of data. The graph shows the throughput as the file sizes increase. For the small file dataset, ROARS data mirroring outperforms HDFS striping, while for the larger file dataset, Hadoop is faster than ROARS. In either case, both ROARS and Hadoop import larger files faster than they do smaller files.

The differences in performance between ROARS and Hadoop are due to the way importing and storing replicas works in both systems. In the case of Hadoop, a replica creation in-

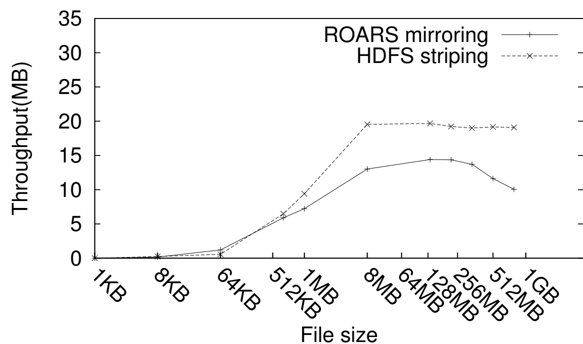


Figure 4: Import Performance.

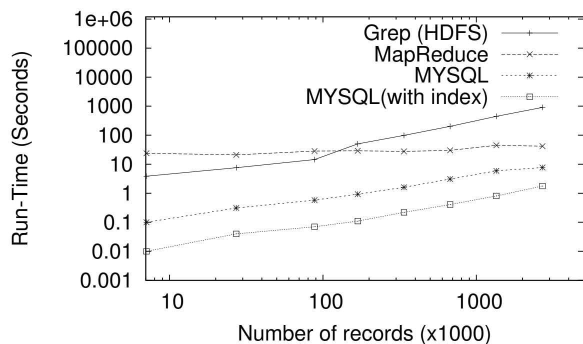


Figure 5: Query Performance.

volves a complex set of transactions that setup a dataflow pipeline between the replication nodes of a single block. This overhead is probably the reason why Hadoop is slightly slower for smaller files. For larger files, however, this data pipeline enables higher overall system bandwidth usage and thus leads to better performance than ROARS which does not perform any data pipelining. Rather, it merely copies the data file to each of the replicas in sequential order. This import and replica creation overhead also explains why large file import is faster than small file importation. In ROARS, each imported file needs 9 database transactions which can be costly when importing small files, where the time spent transferring data is overwhelmed by database transaction execution time. With the larger files, there is less time lost to setting up the connections and transactions, and more time spent on transferring the data to the Storage Nodes.

4.2 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 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, and queried the metadata by executing the custom script using MapReduce [5]. 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.

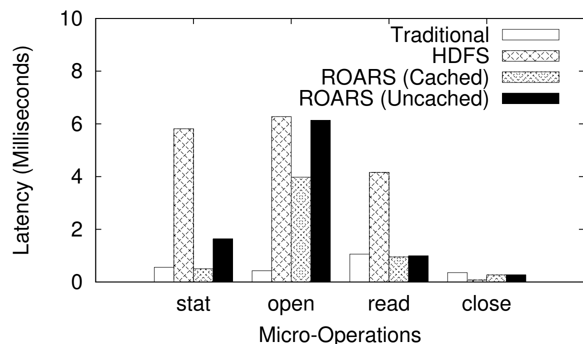


Figure 6: Microbenchmark.

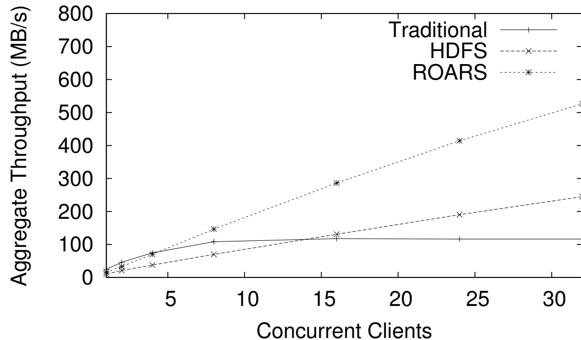
Figure 5 clearly shows that ROARS takes full advantage of the database query capabilities properly and is much faster than either MapReduce or standard *grepping*. Evidently, as the metadata database increases in size, the *grep* performance degrades quickly. The same is true for the `QUERY` operation. Hadoop, however, mostly retains a steady running time, regardless of the size of the database. 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 MySQL. Futhermore, the standard *grep* and MySQL 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 2,699,488 data objects.

4.3 Microbenchmark

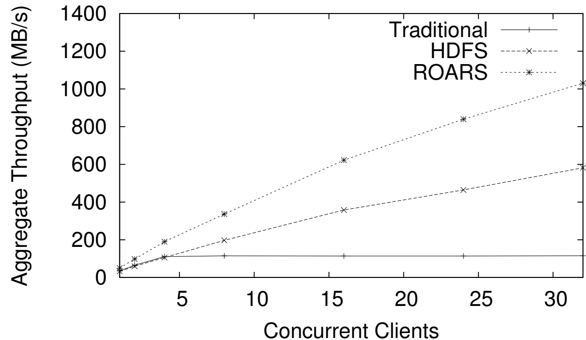
As mentioned earlier, ROARS does not directly support traditional system calls such as `stat`, `open`, `read`, and `close`. Rather, ROARS provides these operations to external applications through a Parrot ROARS service which emulates these system calls. Since Hadoop also does not provide a native filesystem interface, we also implemented a Parrot HDFS service. To test the latency of these common filesystem functions, we performed a series of `stat`s, `open`s, `read`s, and `close`s on a single file on the traditional file server, HDFS, and ROARS. For ROARS we provide the results for a version with SQL query caching and one without this small optimization.

Figure 6 shows the latency of the micro-operations on a traditional network file server, HDFS, and ROARS. As can be seen, ROARS provides comparable latency to the traditional network server, and in the case of `stat`, `open`, and `read`, lower latency than HDFS. Since all file access went through the Parrot storage adapter, there was some 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 while using a SQL database as a persistent metadata cache does incur an overhead cost that increases the latency of these filesystem micro-operations,



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



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

Figure 7: Concurrent Access.

the latencies provided by the ROARS system remain comparable to HDFS. Moreover, this additional overhead can be slightly mitigated by caching the SQL queries on the client side as shown in the graph. With this small optimization, operations such as `stat` and `open` are significantly faster with ROARS than with HDFS. Even without this caching, though, ROARS still provides lower latency than HDFS.

4.4 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, while for the traditional filesystem and HDFS, we exported the data directory to each of those systems. We ran our test program using Condor [22] with 1 - 32 concurrent readers.

Figure 7 shows the performance results of all three systems for both datasets. In Figure 7(a), the clients read 10,000 320KB files, while in Figure 7(b) 1,000 5MB 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 single file server is limited to a maximum upload rate of about 120MB/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 hit 1030.94 MBS/s and HDFS managed 581.88 MB/s. There are a couple possible 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.

Method	Iris Still (300KB)	Face Still (1MB)	Iris Video (5MB)	Face Video (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

Figure 8: Transcoding in Active Storage

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.

4.5 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* [12]. 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* [20] 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.

Figure 8 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

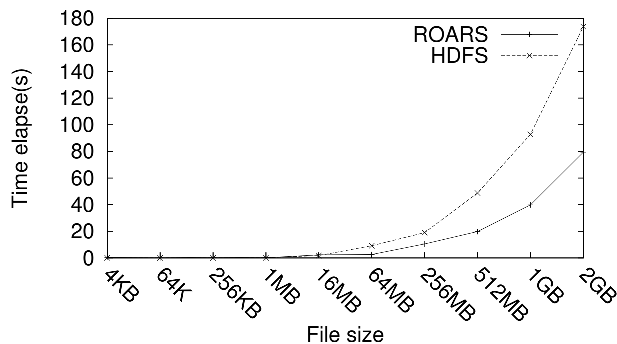


Figure 9: Cost of Calculating Checksum.

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.

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 9 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 (64MB), 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.

When a replica is deemed to be suspect, ROARS will spawn a new replica and delete the suspect copy. ROARS does this by making a copy of a good replica. There are two ways to do this. The first way is to read the good relica to a local machine and then copy it to a Storage Node (first party put). Another way is to tell the Storage Node where the good copy is located and then perform the transfer on the user's behalf (third party put). The latter would requires 2 extra file server operations on the target Storage Nodes.

4.7 Dynamic Data Migration

ROARS is highly flexible data management system where users can transparently and incrementally add and remove Storages Nodes without shutting down a running system. The system provides operations to add new Storage Nodes (`ADD_NODE`), migrate data to new Storage Nodes (`MIGRATE`), and remove data from old unreliable Nodes (`REMOVE_NODE`). To demonstrate the fault tolerance and fail over 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 ensure that ROARS still could be functional even when hardware failure occurs. Figure 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. As Nodes remain inactive, the ROARS continues to endure more and more timeouts. That leads to the decrease of 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. 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 robust 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.

5.1 Filesystems

In order to facilitate sharing of the scientific data, scientific researchers usually employ various network filesystems such as NFS [13] or AFS [9] 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 [25] or Hadoop [8]. 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.

Where filesystems 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

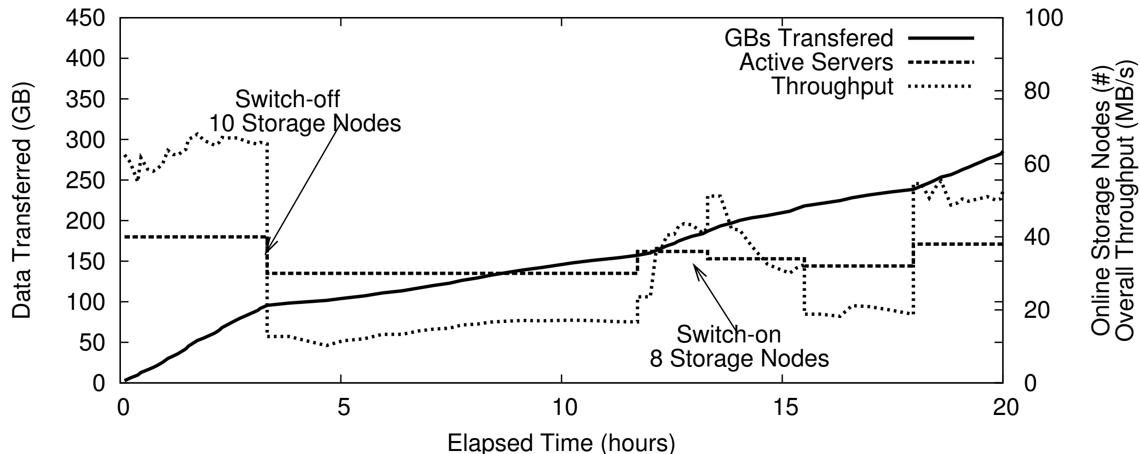


Figure 10: Dynamic Data Migration

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 [5] 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.

5.2 Databases

The other common approach to managing scientific data is to go the route of projects such as the Sloan Digital Sky Survey [18]. 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 [10] or Vertica [23]. 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 work flow commonly used by scientific researchers. In projects such as the Sloan Digital Sky Survey and Sequoia 2000 [17], 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 [16] 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 [1] on distributed systems such as Condor [22]. 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 [11] 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 [6], while ROARS uses a distributed active storage cluster. Another example of a filesystem-database combination is HEDC [15]. HEDC is implemented on a single large enterprise-class machine

rather than an array of Storage Nodes. iRODS [24] and its predecessor the Storage Resource Broker [3] 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.

6. 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.

Currently ROARS is used as the backend storage of BXGrid [4], a biometrics data repository. At the time of writing, BXGrid has 265,927 recordings for a total of 5.1TB of data spread across 40 Storage Nodes and has been used in production for 16 months.

7. 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.

8. REFERENCES

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

[2] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/, 2009.

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

[4] H. Bui, M. Kelly, C. Lyon, M. Pasquier, D. Thomas, P. Flynn, and D. Thain. Experience with BXGrid: A Data Repository and Computing Grid for Biometrics Research. *Journal of Cluster Computing*, 12(4):373, 2009.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.

[6] J. J. Dongarra and D. W. Walker. MPI: A standard message passing interface. *Supercomputer*, pages 56–68, January 1996.

[7] S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.

[8] Hadoop. http://hadoop.apache.org/, 2007.

[9] 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.

[10] 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.

[11] J. No, R. Thakur, and A. Choudhary:. Integrating parallel file i/o and database support for high-performance scientific data management. In *IEEE High Performance Networking and Computing*, 2000.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] Vertica. http://www.vertica.com/, 2009.

[24] 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.

[25] 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.