LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-409717

A Metadata-Rich File System

S. Ames, M. B. Gokhale, C. Maltzahn

January 8, 2009

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Metadata-Rich File System

Sasha Ames
University of California, Santa Cruz

Maya B. Gokhale
Lawrence Livermore National Laboratory

Carlos Maltzahn
University of California, Santa Cruz

Abstract

Despite continual improvements in the performance and reliability of large scale file systems, the management of file system metadata has changed little in the past decade. The mismatch between the size and complexity of large scale data stores and their ability to organize and query their metadata has led to a de facto standard in which raw data is stored in traditional file systems, while related, application-specific metadata is stored in relational databases. This separation of data and metadata requires considerable effort to maintain consistency and can result in complex, slow, and inflexible system operation. To address these problems, we have developed the Quasar File System (QFS), a metadata-rich file system in which files, metadata, and file relationships are all first class objects. In contrast to hierarchical file systems and relational databases, QFS defines a graph data model composed of files and their relationships. QFS includes Quasar, an XPATH-extended query language for searching the file system. Results from our QFS prototype show the effectiveness of this approach. Compared to the defacto standard, the QFS prototype shows superior ingest performance and comparable query performance on user metadata-intensive operations and superior performance on normal file metadata operations.

1 Introduction

The annual creation rate of digital data, already 281 exabytes in 2007, is growing at a compound annual growth rate of 60%, with a projected 10-fold increase over the next five years [15, 14]. Sensor networks of growing size and resolution continue to produce ever larger data streams that form the basis for weather forecasting, climate change analysis and modeling, and homeland security. New digital content, such as video, music, and documents, also add to the world's digital repositories. These data streams must be analyzed, annotated, and searched to be useful; however, currently used file system architectures do not meet these data management challenges.

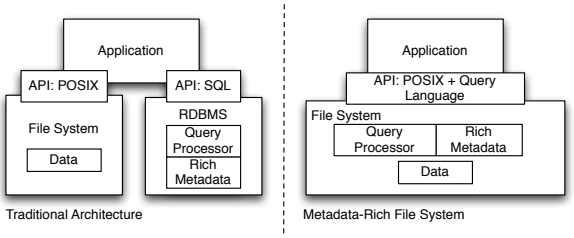


Figure 1: The Traditional Architecture (*left*), to manage file data and user-defined metadata, places file data in conventional file systems and and user-defined metadata in databases. In contrast, a metadata-rich file system (*right*) integrates storage, access, and search of structured metadata with unstructured file data.

There are a variety of ad hoc schemes in existence today to attach user-defined metadata with files, such as a distinguished suffix, encoding metadata in the filename, putting metadata as comments in the file, or maintaining adjunct files related to primary data files. Application developers needing to store more complex inter-related metadata typically resort to the *Traditional Architecture* approach shown on the left in Figure 1, storing data in file systems as a series of files and managing annotations and other metadata in relational databases. An example of this approach is the Sloan Digital Sky Survey [33, 34], in which sky objects and related metadata are stored in a Microsoft SQL Server database and refer to the raw data stored in regular file systems by absolute pathname.

This approach likely emerged because of file systems' ability to store very large amounts of data, combined with databases' superiority to traditional file systems in their ability to query data. Each complemented the other's weakness: file systems do not support flexi-

ble queries to identify files according to their metadata properties, and few databases can efficiently support the huge volume of data that must be stored. Unfortunately, this separation increases complexity and reduces performance and consistency in several ways. First, the metadata must be cast into a relational database form, even though metadata and data conform more closely to a graph model. Then, application developer must design and build a relational database tailored to the application. As the application changes, the database schema might require modification, and all the metadata migrated to the new schema. Using the database to retrieve metadata involves a two-step process of evaluating a query and resolving a potentially large number of file names. Furthermore, the association between metadata and files via POSIX file names is brittle and can become inconsistent when files are moved. Finally, queries cannot easily be restricted to portions of the namespace.

The access profile of data stream ingest, annotation, and analysis-oriented querying does not require the stringent semantics and overhead of database transactions [13], making it feasible to integrate a lighter-weight index into the file system to facilitate the update and query needs of many applications.

To address these needs, we propose, implement and evaluate a *metadata-rich*, queryable file system architecture that maintains user-defined metadata as an intrinsic part of the file data, and simultaneously provides a sophisticated metadata query interface. *Rich metadata* extends POSIX file system metadata, such as standard names, access rights, file types, and timestamps, to include arbitrary user-defined data associated with a file, as well as linking relationships between files [1]. Although many existing file systems support storage of rich metadata in extended attributes, none efficiently support a graph data model with with attributed relationship links or integrate queries against all of the extended attributes into file system naming.

The contributions of this paper are: (1) the design and prototype implementation of the QFS metadata-rich file system based on a *graph* data model (2) the design and prototype implementation of the Quasar path-based file system query language specifically designed for the data model of files, links, and attributes. (3) quantitative evaluation of QFS compared to the Traditional Architecture of hierarchical file system plus relational database.

2 A Metadata-Rich File System

We define a *metadata-rich file system* as one that augments conventional file system I/O services (such as the ones defined by POSIX) with an infrastructure to store and query user-defined file metadata and attributed links between files. Our goal in exploring metadata-rich file systems is to examine their potential for the analysis and management of scientific, sensor, and text data.

Under the Traditional Architecture metadata and data are kept in different systems (see Figure 1, left). The separation has disadvantages in terms of complexity, consistency and performance:

Brittle Schema—The application developer must design a schema specialized for the application. When new attribute or link types must be inserted, the schema must be re-defined, and the database must be migrated to the new schema, a prohibitively expensive operation.

Brittle metadata/data association—The association of metadata to files via POSIX file names is brittle. Large data streams require continual ingest of new data and de-staging of older data into archives. When files get de-staged, their filesystem-specific POSIX path names change. Updating the database requires extra maintenance of indices with considerable update and querying overhead.

Expensive path name evaluation—A query in the Traditional Architecture returns a list of file names that need to be retrieved from the file system. Thus retrieving data involves a two-step process of evaluating a query and resolving a potentially large number of file names.

Global scope—Files are stored hierarchically. Filesystem directories align to semantic meaning and access locality [22]. Yet, the Traditional Architecture does not allow restricting the scope of queries to a directory without extra indexing overhead that is aggravated by the continual stream of new data entering and older data leaving the filesystem.

In contrast (Figure 1, right), the metadata-rich file system integrates the management of and provides a single interface for metadata and data with a general and flexible graph-based schema. Association between data and metadata automatically remains consistent regardless of path name changes. For improved performance, such an integrated system can support combined data and metadata writes. It becomes possible to append additional metadata items to existing files identified by resolved file IDs. Queries presented to such systems resolve directly to files, obviating the need to resolve file names. The query interface, based on the XPATH standard, extends the POSIX file system interface with syntax to select files matching arbitrary metadata characteristics while allowing the query to limit the scope of such selections using path names.

2.1 Data Model

We represent rich metadata using file attributes (similar to extended attributes as defined in POSIX), directional

links between files,[1] and attributes attached to links [2]. File attributes include traditional file system metadata (similar to the Inversion File System [27]). A link is a first-class file system object representing a directional edge from a *parent* file to a *child* file, as shown in Figure 2.

In Figure 2, each file (circle) has a set of attributes in the form of attribute name/value pairs. Files are connected by links, which can also have attributes attached to them. The example shows attribute/value pairs such as [filetype, NewsStory], [IsTabular, yes], [NodeType, SemanticTag], etc. Links can also have attribute/value pairs, such as [LinkType, HasEntity] or [Extractor, Stanford]. In the example, the attributes placed on links contain the provenance of the relationship. For instance, the depicted rightmost link was created by the Stanford Extractor, while the leftmost link was from the Unified Extractor.

Links are attached to files by object ID so that changing file path names will not break links as long as the file remains within the same object ID name space. A file cannot be deleted until all links from and to that file are deleted. Note that more than one link can connect a pair of files. There can be multiple links between two files as long as the links can be distinguished by at least one link attribute or by their direction.

In practice link attributes often include a name and a type attribute. For example, a file directory can be represented by a file pointing to other files with links of the type "child" and with "name" attributes identifying the relative path name of a file. Thus, our data model for metadata-rich file systems does not require extra directory objects. Whenever convenient we will refer to a file with children as "directory". Links can also respresent any kind of relationship that is not necessarily hierarchical, such as provenance, temporal locality, hyperlinks, and bibliographic citations. Files may or may not contain any content other than attributes and links.

This data model addresses brittle metadata/data associations in Traditional Architectures by storing metadata in the corresponding file system objects such that changes to file path names does not break metadata/data associations. It also provides a general schema for storing metadata—attributes and links—rather than requiring application developers to design customized relational schemas. The query language used to search within this graph data model addresses the other weaknesses of the Traditional Architecture, *i.e.,* expensive path name evaluation and the global scope problem.

[1] QFS directional links are not to be confused with hard or symbolic links of POSIX file systems.

2.2 Query Language

The Quasar query language is an integral part of the metadata-rich file system that use the graph data model. Quasar expressions are designed to replace POSIX paths in file system calls and are used as names to query and manipulate the metadata for files. By integrating querying with naming, Quasar avoids full path name evaluation required by the Traditional Architecture.

We base the language syntax around XPath [36], the W3C standard language for XML node selection queries. XPath syntax resembles file system paths and integrates expressions for attribute-based search and node selection based on matching children. Quasar integrates querying into the file system name space by equating queries and path names. Thus, Quasar (as does XPath) subsumes POSIX paths by adding declarative operations to navigational ones: matching operations require a query plan while navigation does not. However, unlike XPath, Quasar has syntax to differentiate between attributes on links or on the files themselves. Additionally, as the rich-metadata file system data model (unlike XPath's data model) is a graph and not a strict hierarchy, Quasar has a search operator to match based on attributes on more than one parent to any given node.

A Quasar query expression is a list of one or more operations. Each operation specifies an operator and its parameters. An operation is evaluated in the context of a current set of file IDs (file set context) generated by the previous operation. The final file set is the result set of the query. The feature of each operation processing the previous file set context allows the language to combine *search* and *navigation* operations within individual queries which solves the global scope problem of the Traditional Architecture. A third type of operation is *presentation* which translates query results into strings. A Quasar language implementation returns the result as a directory whose name is the query expression and whose contents is a list of names of the final file set.

There are two search operations, *attribute matching* and *neighbor pattern matching*. Attribute matching is applied to files and their attributes, while neighbor pattern matching involves parents and/or children of files in the file set context. If a Quasar query begins with a search operation, the initial file-set context is the entire file system.

2.2.1 Search: Attribute Matching

The first search operation, attribute matching, takes one or more attributes as parameters. Attribute search returns a new file-set context containing those files whose attributes match *all* attributes specified in the operation (*i.e.,* the parameter list of attributes is interpreted as conjunction). An attribute may be flagged as "prohib-

ited", in which case, matching files are omitted from the result set. Attribute match operations effectively functions as filters over file sets. For example, suppose we wish to find files with attribute/value pairs [FileType, NewsDocument] and [IsTabular, Yes]. Our output file set is the intersection of all files having [FileType, NewsDocument] and all files with [IsTabular, Yes]. The Quasar query expression is
@IsTabular=Yes;FileType=NewsDocument

2.2.2 Search: Neighbor Pattern Matching

The second search operation, neighbor pattern matching, refines an input file set based on neighbor patterns of each file in that set, *i.e.,* based on attributes of parents or children of that file set. A pattern match operator may also specify constraints on links to parents or children based on link attributes . A Quasar expression using a neighbor pattern match looks like
@FileType=NewsDocument@child:SemanticType
=Location
where an input set containing files with [FileType, NewsDocument] are filtered to only those whose children match [SemanticType, Location].

2.2.3 Navigation

In contrast to search operations which filter file sets, navigation changes the file set through the action of following links from the file set context. The navigation operator accepts link attributes to constrain the links to be followed and file attributes to constrain the result file set. The navigation operation (@navigate) follows links in their "forward" direction, from parent to child. There is also a corresponding operation (@backnav) to traverse from child to parent. For example, the query expression
@FileType=NewsDocument@navigate:^Extractor
=Unified
will change the result set from all files with [FileType, NewsDocument] following links with the attribute [Extractor, Unified]. The ^ character indicates that attribute to match should be found on the links to be followed.

2.2.4 Presentation

The presentation operation translate each query result in the result set into a string. These strings can be attribute values attached to files in the results, including the files' names. For example, the query expression
@FileType=NewsDocument&listby:FileName
lists all the files of [FileType,NewsDocument] by the values corresponding to their FileName attributes.

2.2.5 Examples

The following examples reference the example file system metadata graph shown in Figure 2. A simple query in Quasar only matches file attributes. For example,
@IsTabular=Yes; FileType=NewsDocument
&listby:FileName
will return all files that match (@) the listed key=value pairs, and each file is listed by the FileName attribute value.

To illustrate neighbor pattern matching, suppose we have a file system containing some files with attribute/value pair [FileType, NewsDocument] and other files with attribute/value pairs [NodeType, SemanticTag][2] Each "NewsDocument" links to the "SemanticTag" files that it contains. Each link is annotated with a "LinkType" attribute with value "HasEntity" ([LinkType, HasEntity]). Our input file set consists of NewsDocument files that are tabular (files with [FileType, NewsDocument], [IsTabular, yes] attribute/value pairs). We refine the file set context by a neighbor pattern match that matches links of type "HasEntity" ([LinkType, HasEntity]) and child files that have [NodeType, SemanticTag] and [SemanticType, Location]. The output file-set context will contain only those NewsDocuments that link to SemanticTags matching the above criteria. In Quasar, the query expression is:

@FileType=NewsDocument/@child:^LinkType
=HasEntity;NodeType=SemanticTag;
SemanticType=Location.

Similarly,
@FilleType=NewsDocument@child:
SemanticType=Location;SemanticValue=New York
&listby:FileName
specifies properties that child nodes must match. First, files of the specified FileType are matched. Second, we narrow down the set of files by matching child nodes with the specified SemanticType and SemanticValue file attributes. Finally, using the presentation operator, we return the set according the document FileName attribute value.
The query,
@FileName=N20090201~N20090301@navigate
^LinkType=HasCoOccurence&listby:ProximityScore,

first, matches files in the specified range (in this example files named by a date between February 1st and March 1st, 2009). Second, it traverses links from the

[2]This example comes from our workload evaluation application (see Section 4), in which text documents are annotated with semantic entities found within. We represent the semantic entities as directories linked from the file containing the text, and label such directories as "nodes", hence the use of the "NodeType" attribute.

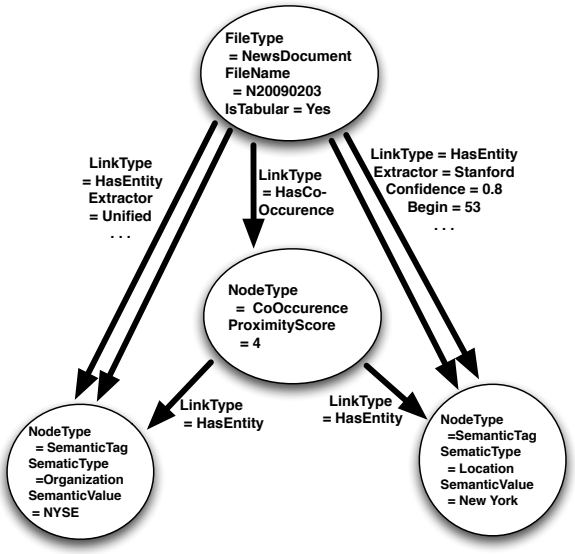


Figure 2: An example of files links and attributes. Circles represent files, arrows represent links.

matching source files (@navigate), only following links that match the [LinkType, HasCoOccurence] attribute,, The ^ character indicates that the link attribute should be matched. Finally, it lists the resulting file set by the ProximityScore attribute.

3 Implementation

We have implemented a prototype metadata-rich, querable file system called the Quasar File System (QFS). This prototype allows us to explore the fundamental costs, benefits, and challenges that are incurred by the graph data model approach and by searchable metadata within the file system.

3.1 Overview

As shown in Figure 3 QFS is implemented as a file server running in user space and exporting a 9P interface [20]. *Clients* pose standard POSIX file system operations to the *Kernel Interface* via systems calls. The *Virtual File System* forwards the requests to the *9P File System Client* kernel module, as is standard for mountable file systems. The 9P client kernel module serializes the calls and passes the messages to the *QFS Software* service running the file server code in user space.

The *9P Service Library* implements the listening part of the service which receives the messages from the kernel and decodes the specific 9P operations, which resemble POSIX file systems operations. The QFS *File System*

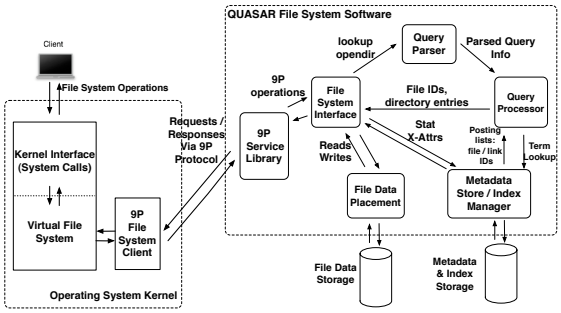


Figure 3: The QFS prototype software architecture is a single-host file server exporting a 9P interface that allows clients to the POSIX file system interface to pass Quasar expressions.

Interface implements handler routines for the various 9P operations and interacts with the other components of the system.

To obtain a file id, the client submits a Quasar expression, which is parsed by the *Query Parser* and then passed to the *Query Processor*. The processor generates a query plan and then looks up query terms in the *Metadata Store / Index Manager*. The MS/IM returns posting lists of relevant files or link ids, or may filter attributes for a particular file. The query processor uses standard query planning strategies using statistics on the stored metadata. The store manager uses the underlying file system to store metadata structures. Once the query processor has computed an answer to the query, it returns the list of ids to the file system interface.

Other file system operations may go directly from the interface operation handler to the data or metadata management components. Stat and attribute update/retrieval calls go directly to the store manager, once the specified file has been looked up. File data operations (read/write) go to a *File Data Placement* manager. In our QFS prototype, this module maps file data to files stored within an underlying local (ext2) file system. Only non-zero length files[3] are represented in the ext2 file system. Zero-length files that contain only attributes and links are managed solely by the metadata store and are therefore significantly cheaper to manage than regular files. For POSIX compliance, a zero-byte file with links is equivalent to a directory.

3.2 QFS semantics for directory/file operations

QFS follows POSIX semantics as closely as possible, and extend the semantics as needed for operations that in-

[3] or zero-byte files without links

volve metadata and links. In particular, as many file system operations require a pathname to a particular file, operations posed to QFS may specify a "pathname query", which accepts any valid Quasar expression, including POSIX paths.

A high level description of QFS behavior for common file system calls is as follows:

stat Looks up the pathname query. If it matches a single file, it returns the POSIX attributes for that file from the metadata store. If more than one file match, it returns attributes for a virtual directory.

open (create, write) Looks up the pathname query. If no match, it creates a new file object in the metadata store, stores the name or attributes given in the query expression, and looks up a parent file. If a parent is found, it creates a link with parent as source and new file as link target. It creates a file in the underlying file system for data storage and opens the file. If initial query matches a file, it opens the corresponding underlying file. Finally, it returns the handle to client.

open (read) Looks up the pathname query. If exactly one result is found and it is not flagged as a directory, it opens the corresponding data file in the underlying file system. Otherwise, it follows the opendir semantics.

mkdir Same as "open create", but sets the "DIR" flag in the file object, but does not create or open an underlying file as no data storage is necessary.

opendir Looks up the pathname query. For each query result, it looks up particular attributes to return for the result based on a "ListBy" operator in the query. It returns a directory handle to the client. It stores the attribute value strings in a cache for successive readdir operations until the directory handle is closed.

readdir Retrieves next directory entry (query result) in the result cache.

read/write For a given open file handle, performs a read/write operation on the corresponding file in the underlying file system for data storage.

close(dir) Passes file handles to underlying file system to close the file. Frees temporary structures used to represent query results for directory listings.

chmod/chown,time Looks up the pathname query. Then, modifies the permissions, owner, or time attribute for the result file's object structure.

rename Depending on the result of the pathname query, will do one of the following: (1) Change the name (or other) attribute for a file, without affecting its parents/children (2) Change the parent of a file (3) Update the affected link(s) and their associated attributes. The pathname must resolve to a single source file.

unlink Looks up a pathname query. If the query matches a single file, it also looks up the parent to the file within the query, determines the link between parent and child, and removes that link from the metadata store.

A consequence of changing attributes of a file is that it might invalidate the path name that an application uses to refer to that file. For example, if an application names a file by the attribute $k = v$ and then subsequently changes its attribute to $k = v'$, the original name does not resolve to that file anymore. One way to provide greater name space stability is to (1) use QFS assigned immutable file or link IDs to address files (equivalent to inode numbers), as both are searchable attributes in QFS, or (2) make a unique, immutable object ID for each file and link available as attributes and include object IDs into the Quasar name space (if applications need the convenience of their own ID schemes). Either scheme provides applications with names that are immune to any metadata changes. The second approach is already used in existing systems, for instance, document databases use DOI.

3.3 Standard Optimizations

Similarly to regular file systems, the Query Processor maintains a name cache which maps Quasar path expressions to a already computed query plan and result set of file and link IDs. We have found that even a single element name cache has significant performance benefit since it can handle consecutive query operations with the same pathname. This is a frequently recurring pattern as applications often stat a pathname query prior to issuing an open or opendir.

Another commonly used optimization, batching commands from client to server, has also been implemented using the 9P protocol. The client can open a distinguished "synthetic" file and batch multiple newline-delimited Quasar metadata update expressions into a single write operation. Additionally, reading a group of directory entries can be accomplished by a single client command. Instead of issuing a single readdir call for every single directory entry, the client can issue a *batch-readdir* and receive a handle to a synthetic file that includes all directory entries.

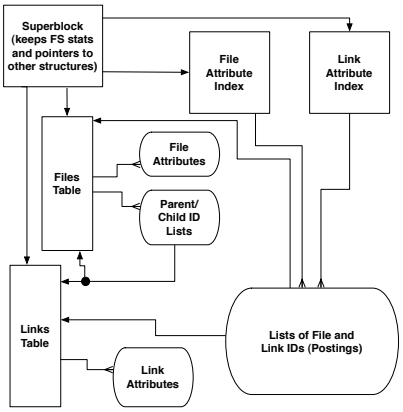


Figure 4: The schema of the QFS metadata store is optimized for attribute matching, neighbor pattern matching, and navigation.

3.4 Metadata Store / Index Manager (MS/IM)

The MS/IM assumes a graph data model for the metadata-rich file system and the basic Quasar operations as introduced in Section 2. The data structures of the metadata store are a collection of arrays, sorted lists, and red-black trees. These data structures are backed by a memory-mapped file in the underlying file system.

The data structures are optimized for query operations expressible in Quasar, namely attribute matching for a given set of files, neighbor pattern matching, and navigation (see Figure 4). The metadata store has a *Superblock*, which contains references to the other structures within the store and some global statistics, such as file and link counts. The *File Table* is an array and maps file IDs to file objects (similar to inodes), each of which includes pointers to a tree *File Attributes*, a list of parents and a list of children (recall that "parents" are files with links pointing to the current file and "children" are files to which the current file's links point). Within the list (*Parent/Child ID Lists*) entries, each parent and each child is represented as a tuple containing a file ID and a link ID. The link source and target need not be stored explicitly as they can be accessed through the File Table. For instance, Neighbor pattern matching and link traversal query operations start with a given set of files and never with a set of links, so the links' sources and targets are already known. The *Link Table* is an array that maps link IDs to each *Link Attribute* list. The *File and Link Attribute Indices* are red-black trees, and they map attributes (name-value pairs as keys) to the *Lists of File and Link IDs (Postings)*.

To illustrate how these structures are used, consider the match operator. Single Quasar match operators find the search attribute name and value in the file attribute index tree. Once the attribute node is located, the list of matching file ids is returned. In the case of match operators with multiple attributes, the query planner determines if (1) multiple lists should be intersected (computation time $O(n_1 + n_2)$, where n_1 and n_2 are the lengths of lists), or (2) the initial list of file ids should be filtered by looking up attributes via the file table (constant time lookup for each attribute, thus $O(n_1) * C$)

The design and careful implementation of metadata management is key to the QFS prototype. Unlike schemata for relational databases, which are tailored to each application, QFS maintains a single metadata store schema for all applications.

4 Evaluation

In this section we present a quantitative evaluation of QFS. We evaluate QFS using metadata-intensive ingest and query workloads. The workloads are generated by a data management application that fits the Traditional Architecture, similar to those used for survey astronomy (eg. SDSS, PAN-STARRS, LSST [5, 7, 32, 34], high-energy physics [6] and other data-intensive application domains. Second, we explore the costs and potential benefits of the use of metadata rich file systems in comparison with a standard POSIX file system for normal file system operations.

4.1 User-defined metadata: ingest and query

To evaluate the performance of QFS on ingest and query workloads we have extended the entity extraction benchmark Lextrac [9]. The benchmark stresses the novel aspects of QFS by the extensive use of links to express relationships among objects, and by the storage and retrieval of searchable metadata attached to links and files.

In its original form, Lextrac processes files in a multi-stage analysis pipeline. Each stage appends newly derived metadata to the analyzed file, such as entities and proximity scores between entities, so it is available to the next stage along with the original content. At the end the metadata part of each file is stored in a relational database along with references to the corresponding files. Thus, the result of the original Lextrac is a system of Traditional Architecture. When the Reuters News corpus sample data set is fully ingested, Lextrac has created roughly 540,000 entities for 4000 news document files and 59.5 million entities for 450,000 news document files. In comparison, the most recent SDSS data release [33] contains 59 million objects.

We extended Lextrac so it supports the QFS storage interface in addition to the POSIX I/O interface and can take full advantage of the QFS data model. We also added to Lextrac's *ingest* phase a *query* phase which consists of a set of queries that are representative for applications that make use of entity extraction. These queries contain selection and projection critera for documents, entities, and proximity scores, and can be posed either to a relational database as part of a Traditional Architecture or to QFS.

The evaluation was conducted on several configurations. The "8GB HD" configuration is an Intel Xeon quad core, dual socket server with 8GB main memory and a 250GB SATA drive running Fedora Core 9-64 bit (Linux kernel version 2.6.27), the ext2 file system and PostgresSQL 8.3. For the ingest study, we additionally used "16GB HD" and "16GB SSD" configurations consisting of a Dual-Core AMD Opteron 2.8 GHz, 4 socket server with 16GB main memory running Linux kernel version 2.6.18, either with a 250GB SATA drive, or two Pliant SSDs with software RAID-1 striping.

For the FS+DB/SQL configurations discussed in this section, we have configured PostgreSQL with a schema specific to the Lextrac application. We create indexes on all columns within this schema to speed up SQL query performance. In addition, we run the database without transactions or isolation.

4.1.1 Ingest

We compare ingest performance of several workload sizes of the Traditional Architecture vs. QFS, with the sizes ranging from 4,000 to 450,000 documents from the Reuters News Corpus. Exactly the same entity extraction computation is performed in either case, the difference being how the entities, proximity scores, and other metadata are stored. As shown through the first pair of bars based on configuration "8GB HD" for each workload size in Figure 5, QFS completes ingest in less than half the time of the traditional File System plus Database (FS+DB) approach for the smaller workload sizes (4,000-100,000 documents), and in roughly two-thirds the time for the largest workload (450,000 documents). The second pair of bars based on configuration "16GB SSD" show that QFS computes ingest in about 1.85 times faster than FS+DB, consistently for all measured document counts. Figure 6 presents the scalability of QFS ingest for four different document collection sizes, with three different hardware configurations. As our ingest contains random reads in addition to writes, we notice that scalability degrades at a slower rate when 16GB are available. Even with 16GB, there is degradation beginning at the 100,000 document mark, when we start to see increasing numbers of cache misses forcing random reads to the storage device. The SSD configuration appears to respond to the random reads with lower latency, suggesting that future storage systems incorporating SSD will better support scaling to large ingest workloads than HD.

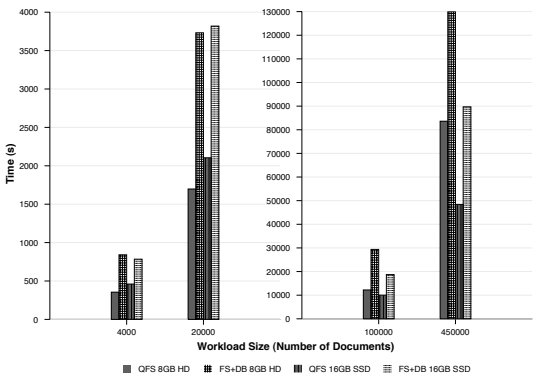


Figure 5: Comparison of Ingest Performance: QFS vs. File System + Relational DB, measured on 8GB HD and 16GB SSD hardware configurations.

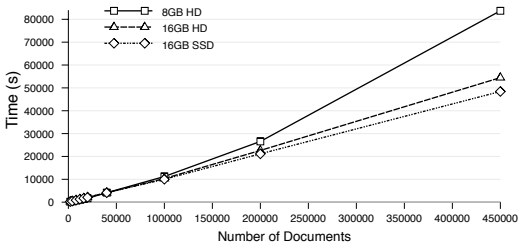


Figure 6: Scalability of QFS ingest, measured over three hardware configurations. The use of 16GB and SDD has closest to linear scaling with the size of the workload.

4.1.2 Querying

The query study uses query templates $Q0 - Q4$ representative of queries that would be applied to the document set. [4]

$Q0$ Find all documents that contain reference to a particular entity. Example: Find all documents containing the place "New York."

$Q1$ Find all documents that contain reference to both entities X and Y that have a particular proximity score between them. Example: Find all documents that

[4]We thank John Compton of LLNL for his guidance in the design of the query templates.

contain "New York" and "NYSE" with proximity score of "25".

$Q2$ Find all proximity scores relating two particular entities in documents with names in a particular range. Example: find the proximity scores relating "New York" and "NYSE" in documents with names in the range of "N20090101" – "N20090331."

$Q3$ Find all entities related to entity X in documents with names in a particular range and whose proximity score with X is in a particular range. Example: find entities co-occurring with "New York" in documents with names in the range "N20090101" – "N20090331" whose proximity score with "New York" is between "20" and "30".

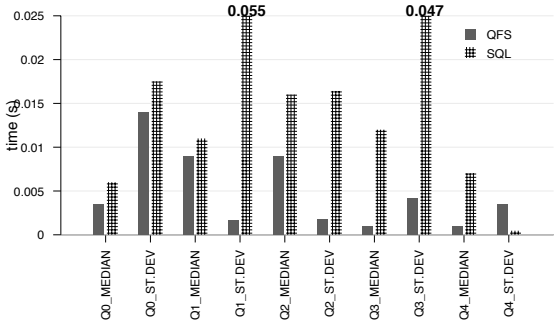
$Q4$ Find all entity pairs with a particular proximity score. Example: find all entity pairs with proximity score "50".

For the query workload experiment, query terms were selected randomly from subsets of the terms appearing in the data. The entire collection of terms was sorted by frequency of occurrence in the document set, and then the subset was created by selecting terms from the sorted list according to either an arithmetic or geometric series. The arithmetic series favors terms with low document frequencies (as are a majority of entities), while the geometric series samples from most frequent to least. Our preliminary work with queries indicated that the use more frequent terms resulted in longer query times than the infrequent terms, due to processing of long lists of results. Thus, we developed this process to provide a meaningful variety of terms to use in the queries, as simply selecting the query term at random might not give good variability. Proximity score ranges were chosen randomly from subranges of $10 \ldots 30$ and document ranges were chosen to select between 1%–10% of the total number of documents. Q0 selects entity values based on combining the arithmetic and geometric series. 34 queries are run for Q0 over the 20,000 document collection and 43 over the 450,000 documents. Q1-Q4 select terms randomly from the geometric series only. We chose this approach because the geometric series contains more of the common terms, which should increase the probability of matching co-occurrences. We run 5,000 queries each in Q1-Q4 for 20,000, 200 queries for 450,000 documents.

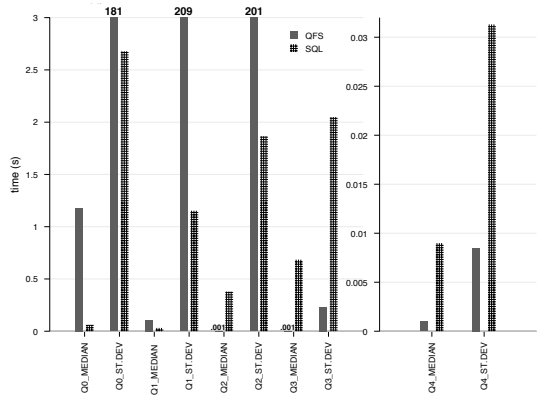
As shown in Figure 7, we find that for the smaller document collection, QFS is consistently faster than FS+DB for all query types. For the larger document collection, FS+DB is faster on query type $Q0$ and $Q1$, while QFS is significantly faster on queries of type $Q2$, $Q3$ and $Q4$. The small document collection FS+DB shows a high standard deviation (wide variation in response time) on $Q1$, and a lesser amount on $Q2$ and $Q3$, with QFS showing greater variability on $Q0$ and $Q4$. In contrast, on the large document collection, FS+DB shows higher variability on $Q3$ and $Q4$, while QFS is worse on $Q0$, $Q1$, and $Q2$. We attribute the high variability to sensitivity to terms with high term frequencies which occur in the correlation stream more often due to the geometric progression. We think that this sensitivity is especially strong in QFS since high term frequencies might result in main-memory misses and QFS' lack of file I/O optimizations.

Original Documents	2.0 GB
Intermediary Files	7.6 GB
Database Storage	18 GB

Table 2: Storage characteristics for FS+DB, 450000 documents



(a) Queries Q0–Q4: 20,000 Documents



(b) Queries Q0–Q4: 450,000 Documents

Figure 7: Comparison of Query Performance: QFS vs. Relational DB

Document Count	4000	20000	100000	450000
Metadata storage size	677 MB	2.83 GB	13.9 GB	63.6 GB
Total Files + Directories	5.41E+05	2.72E+06	1.35E+07	6.01E+07
Links	1.82E+06	9.23E+06	4.62E+07	2.11E+08

Table 1: QFS metadata storage characteristics

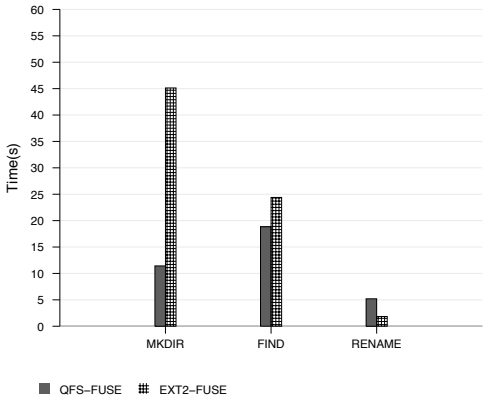


Figure 8: Measurements of QFS vs ext2, benchmarking several file system operations. Both file systems are accessed via a FUSE interface.

4.1.3 Discussion

The performance results show that QFS ingest performance out-performs the Traditional Architecture over a variety of data set sizes. The query performance is excellent on small data sets, but shows variability in performance on the larger data set. QFS is faster than FS+DB by several hundred times on $Q2$, $Q3$, and $Q4$. We attribute this extremely high performance to our ability to use navigation operators to reduce the file set context for subsequent operators. In contrast, the SQL query is purely declarative and relies on the database query optimizer to filter the candidate set.

Table 1 reveals one reason for the lower QFS performance on queries 0, 1, and 2 in the larger data set. The table shows the size of the metadata store used by QFS for different document collection sizes. The metadata store requirement of the Traditional Architecture implementation is much smaller (Table 2). For the largest document set, QFS uses nearly 2.5 times the amount of space for metadata as does the FS+DB approach. We are in the process of reducing QFS' metadata storage size to further increase its performance, e.g. by string de-duplication.

4.2 Standard file operations

We present several measurements to compare the performance of QFS with a conventional Linux file system, ext2, under regular file system workloads and micro-benchmarks that only use POSIX I/O operations. Since the difference between QFS and ext2 is primarily in the management of metadata, we are particularly interested in POSIX I/O metadata-oriented operations. Therefore the several benchmarks we devised exercise mkdir, stat, opendir/readdir, and rename.

For this evaluation, we compare a QFS file system implementation with a File System in User Space (FUSE) interface[35] with ext2 running under FUSE. This was necessary because the base 9P library being used [23] did not implement all the operations being tested. Figure 8 shows three measured categories. For MKDIR (first bars), the system creates a directory 10-ary tree with 111,110 total directories. FIND (second bars) measures running the find command over the directory mentioned above and that measurement exercises stat, opendir and readdir operations. MOVE (third bars) measures 5115 individual directory moves.

In the first (MKDIR) category, we observe that QFS-FUSE completes nearly 4 times faster than EXT2-FUSE. With EXT2-FUSE, the *mkdir* operation is performed by the ext2 file system, whereas in QFS, the operation is performed by the metadata store and index manager. Additionally, our prototype implementation of QFS has been optimized towards writing large numbers of files, as is required to ingest metadata-rich workloads.

The second category (FIND) shows QFS-FUSE completing the procedure in 23% less time than EXT2-FUSE. Due to the nature of the opendir/readdir interface, the FUSE interface likely accounts for much of the measured time in both implementations. The find utility performs a large numbers of lookup operations. Ext2 must search through directory entries linearly in order to resolve pathnames, while QFS uses an index.

We observe a factor of 2.8 slower performance for QFS-FUSE vs EXT2-FUSE for the MOVE benchmark. The QFS prototype has not been optimized for quickly moving files around, and so individual performance of these operations may suffer at the benefit of other operations. However, we consider this an acceptable trade-off as our example domains rarely demand that many files

move intra-device as performed in this example. More often we may see many files move inter-device where both data and metadata must be transferred between file systems.

5 Related Work

Examples of searchable file systems using relational databases and keyword search engines include Apple's Spotlight[3], Beagle for Linux [4], and Windows FS Indexing [25]. These systems provide full-text search and metadata based search and have indexing subsystems that are separate from the file systems and require notification mechanisms to trigger incremental indexing. A recent experimental file search system, Spyglass [22], provides attribute indexing using K-D trees. The authors also compare the performance of Spyglass with a relational database and find that Spyglass has superior query performance when executing joins over multiple attributes.

There has been a good amount of research focused on enhancing file systems through attribute-based paths. The Semantic File System [16] and the Logic File System [28] provided interfaces that use boolean algebra expressions for defining multiple views of files. Other approaches have a separate systems interface to handle searching and views of files, namely the Property List DIRectory system [24], Nebula [8], and attrFS [37]. Some systems combine POSIX paths with attributes [31, 26] and directories with content [17]. Most recently, Prospective provided a decentralized home-network system that uses semantic attribute-based naming for both data access and management of files [30]. While many of these system maintained the equivalent of extended attributes on files before these became part of POSIX, none provide relational linking between files.

Like QFS, the Linking File System [1, 2] included links with attributes between pairs of files. However, LiFS does not implement a query language or any indexing. A key assumption for the design of LiFS' metadata management is the availability of non-volatile, byte-addressable memory with access characteristics similar to DRAM. The design of QFS does not make that assumption.

CouchDB is a distributed, document-centric database system which provides contiguous indexing and supports views [12]. Dataspace systems are an approach to index and manage semi-structured data from heterogenous data sources [19]. It is not clear whether their approach assumes a file system, and if so, how it would interact with their structured data interface.

The transactional record store of [18] attempts to provide a structured storage alternative to databases that addressed the duality between file systems and databases. This approach is a record-based file system, with much of the focus on the transactions used to interact with these records, but the system prototype uses a relational database back-end.

MapReduce [10] is a framework for distributed data processing that shows an example of the use of file system storage for data management problems instead of relational databases. Though Google designed the framework for its need to build and query large distributed indices, their success has pushed the model into other uses in data processing and management. Hadoop has become a popular open-source alternative for applications that wish to employ the MapReduce processing model in Java, and there are a number of technologies in other languages. The Pig and Dryad [21] projects provide alternatives, where they employ specialized high-level languages that have both imperative and declarative features, including syntax for handling joins. In addition, Dryad is a more generalized parallel data processing model.

A common criticism of the approach is that it requires imperative style programming [11], as opposed to posing queries in declarative languages such as SQL. For instance, to join over multiple data sets in Hadoop, the application programmer must implement the join functionality, instead of relying on a query planner. To improve understanding of the trade-offs of these various technologies, Pavlo et al. compare and contrast Hadoop, a traditional RDBMS and Vertica, a column store DB [29].

6 Conclusion

We have presented the design and prototype implementation of a metadata-rich file system and its associated query language. The utility of providing relational links with attributes was demonstrated. We quantitatively analyze the QFS implementation with respect to insertion and query of metadata and links, and compared performance to the defacto standard method of metadata management, the Traditional Architecture using a file system plus relational database. We show that QFS can scale to millions of objects in hundreds of thousands of files. Using a simple, general graph model schema, QFS outperforms the traditional architecture by a factor of 2 in ingest and is comparable to the traditional architecture in query. We identify directions for performance improvement in the QFS implementation to improve scalability and consistency in response time.

Acknowledgements

This work is supported in part by the Department of Energy under Contract DE-AC52-07NA27344, award DE-

FC02-06ER25768, and industry sponsors of the UCSC Systems Research Lab. We thank John May and Scott Brandt for their value feedback on this paper and Ethan L. Miler for his advice during the early stages of the work.

References

[1] AMES, A., BOBB, N., BRANDT, S. A., HIATT, A., MALTZAHN, C., MILLER, E. L., NEEMAN, A., AND TUTEJA, D. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (Monterey, CA, Apr. 2005).

[2] AMES, S., BOBB, N., GREENAN, K. M., HOFMANN, O. S., STORER, M. W., MALTZAHN, C., MILLER, E. L., AND BRANDT, S. A. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, May 2006), IEEE.

[3] APPLE DEVELOPER CONNECTION. Working with Spotlight. http://developer.apple.com/macosx/tiger/spotlight.html, 2004.

[4] BEAGLE PROJECT. About beagle. http://beagle-project.org/About, 2007.

[5] BECLA, J., HANUSHEVSKY, A., NIKOLAEV, S., ABDULLA, G., SZALAY, A., NIETO-SANTISTEBAN, M., THAKAR, A., AND GRAY, J. Designing a multi-petabyte database for lsst, Apr 2006.

[6] BECLA, J., AND WANG, D. L. Lessons learned from managing a petabyte. In *CIDR* (2005), pp. 70–83.

[7] BELL, G., HEY, T., AND SZALAY, A. Beyond the data deluge. *Science 323*, 5919 (March 2009), 1297–1298.

[8] BOWMAN, C. M., DHARAP, C., BARUAH, M., CAMARGO, B., AND POTTI, S. A File System for Information Management. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems* (March 1994). nebula FS.

[9] COHEN, J., DOSSA, D., GOKHALE, M., HYSOM, D., MAY, J., PEARCE, R., AND YOO, A. Storage-intensive supercomputing benchmark study. Tech. Rep. UCRL-TR-236179, Lawrence Livermore National Laboratory, Nov. 2007.

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[11] DEWITT, D., AND STONEBRAKER, M. Mapreduce: A major step backwards. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html, January 2008.

[12] FOUNDATION, T. A. S. Apache couchdb: Technical overview. http://incubator.apache.org/couchdb/docs/overview.html, 2008.

[13] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Oct. 1997), pp. 78–91.

[14] GANTZ, J. F., CHUTE, C., MANFREDIZ, A., MINTON, S., REINSEL, D., SCHLICHTING, W., AND TONCHEVA, A. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC white paper, sponsored by EMC, Mar. 2008.

[15] GANTZ, J. F., REINSEL, D., CHUTE, C., SCHLICHTING, W., MCARTHUR, J., MINTON, S., XHENETI, I., TONCHEVA, A., AND MANFREDIZ, A. The expanding digital universe: A forecast of worldwide information growth through 2010. An idc white paper - sponsored by emc, IDC, March 2007.

[16] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, JR., J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)* (Oct. 1991), ACM, pp. 16–25.

[17] GOPAL, B., AND MANBER, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999), pp. 265–278.

[18] GRIMM, R., SWIFT, M., AND LEVY, H. Revisiting structured storage: A transactional record store. Tech. Rep. UW-CSE-00-04-01, University of Washington, Department of Computer Science and Engineering, Apr. 2000.

[19] HALEVY, A., FRANKLIN, M., AND MAIER, D. Principles of dataspace systems. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2006), ACM, pp. 1–9.

[20] HENSBERGEN, E. V., AND MINNICH, R. Grave robbers from outer space: Using 9p200 under linux. In *Proceedings of the Freenix Track: 2005 USENIX Annual Technical Conference* (2005), pp. 83–94.

[21] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 59–72.

[22] LEUNG, A., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2009), pp. 153–166.

[23] MAGLIONE, K. Libixp. http://www.suckless.org/libs/libixp.html, 2007.

[24] MOGUL, J. C. Representing information about files. Tech. Rep. 86-1103, Stamford Univ. Deptartment of CS, Mar 1986. Ph.D. Thesis.

[25] MSDN. Indexing service. http://msdn.microsoft.com/en-us/library/aa163263.aspx, 2008.

[26] NEUMAN, B. C. The prospero file system: A global file system based on the virtual system model. *Computing Systems 5*, 4 (1992), 407–432.

[27] OLSON, M. A. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference* (San Diego, California, USA, Jan. 1993), pp. 205–217.

[28] PADIOLEAU, Y., AND RIDOUX, O. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 99–112.

[29] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD '09* (2009), ACM.

[30] SALMON, B., SCHLOSSER, S. W., CRANOR, L. F., AND GANGER, G. R. Perspective: Semantic data management for the home. In *fast09* (2009), M. I. Seltzer and R. Wheeler, Eds., USENIX, pp. 167–182.

[31] SECHREST, S., AND MCCLENNEN, M. Blending hierarchical and attribute-based file naming. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS '92)* (Yokohama, Japan, 1992), pp. 572–580.

[32] SIMMHAN, Y., BARGA, R., VAN INGEN, C., NIETO-SANTISTEBAN, M., DOBOS, L., LI, N., SHIPWAY, M., SZALAY, A. S., WERNER, S., AND HEASLEY, J. Graywulf: Scalable software architecture for data intensive computing. *Hawaii International Conference on System Sciences 0* (2009), 1–10.

[33] SURVEY, S. D. S. Web page. www.sdss.org, 2009.

[34] SZALAY, A. S., GRAY, J., THAKAR, A. R., KUNSZT, P. Z., MALIK, T., RADDICK, J., STOUGHTON, C., AND VANDENBERG, J. The sdss skyserver: public access to the sloan digital sky server data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2002), ACM, pp. 570–581.

[35] SZEREDI, M. File System in User Space README. http://www.stillhq.com/extracted/fuse/README, 2003.

[36] W3C. Xml path language (xpath) 2.0. http://www.w3.org/TR/xpath20/, 2007.

[37] WILLS, C. E., GIAMPAOLO, D., AND MACKOVITCH, M. Experience with an Interactive Attribute-based User Information Environment. In *Proceedings of the Fourteenth Annual IEEE International Phoenix Conference on Computers and Communications* (March 1995), pp. 359–365.

Prepared by LLNL under Contract DEAC52-07NA27344.