LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-518611

QMDS: A File System Metadata Management Service Supporting a Graph Data Model-based Query Language

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

December 6, 2011

The International Journal of Parallel, Emergent and Distributed Systems

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–25

RESEARCH ARTICLE

QMDS: A File System Metadata Management Service Supporting a Graph Data Model-based Query Language

Sasha Ames[a,b*], Maya Gokhale[a] and Carlos Maltzahn[b]

[a]*Computer Science Dept., University of California, Santa Cruz, USA*
[b]*Lawrence Livermore National Laboratory, CA, USA*

()

File system metadata management has become a bottleneck for many data-intensive applications that rely on high-performance file systems. Part of the bottleneck is due to the limitations of an almost 50 year old interface standard with metadata abstractions that were designed at a time when high-end file systems managed less than 100MB. Today's high-performance file systems store 7 to 9 orders of magnitude more data, resulting in numbers of data items for which these metadata abstractions are inadequate, such as directory hierarchies unable to handle complex relationships among data. Users of file systems have attempted to work around these inadequacies by moving application-specific metadata management to relational databases to make metadata searchable. Splitting file system metadata management into two separate systems introduces inefficiencies and systems management problems.

To address this problem, we propose QMDS: a file system metadata management service that integrates all file system metadata and uses a graph data model with attributes on nodes and edges. Our service uses a query language interface for file identification and attribute retrieval. We present our metadata management service design and architecture and study its performance using a text analysis benchmark application. Results from our QMDS prototype show the effectiveness of this approach. Compared to the use of a file system and relational database, the QMDS prototype shows superior performance for both ingest and query workloads.

Keywords: graph data model; query language; metadata management

1. Introduction

While storage systems continue to increase in volume and to improve in throughput, the organization of user-defined metadata has lagged behind and has become a bottleneck for data-intensive applications. The growth in storage accompanies a surge in data-intensive scientific computing, in which we have witnessed thousand-fold increases in data volume over the past decade [1]. Cheaper costs for storage and higher bandwidth per system has enabled data capture at the required volumes. Much of the data is stored in file systems using the POSIX interface, a standard based on the systems first designed fifty years ago, based on a hierarchal data model for file organization. This arrangement originated for systems with 10,000s of files, yet now we store 7 to 9 orders of magnitude greater numbers of files within a single system.

Raw data growth has been accompanied by an increase in the capabilities of data analysis, which has resulted in the growth of additional related metadata. For instance, the Sloan Digital Sky Survey [2, 3] (SDSS) is comprised of a collection of

*Corresponding author. Email: sasha@llnl.gov photographic image and spectra files, and a catalog containing image and instrument metadata, photometric and spectroscopic object data. While the data size of the raw photographic imagery has grown from 2.3 TB to 15 TB over the span seven releases, the catalog has grown from being an order of magnitude smaller than the raw data size to become fourteen percent larger.

The needs of today's applications force us to address the limitations of the POSIX interface, in which the only user-defined metadata constructs available are the file name and its location within a hierarchical name space. While hierarchies have served as a useful organizational tool and still have benefits, they are deficient [4], in part due to the inability to express more complex relationships that occur among data. In contrast, a data model that conveys relationships and extends hierarchies would be one employing a graph structure. The graph data model is suitable to many arrangements of data found in a variety of applications [5]. Graphs fit existing file systems through subsuming hierarchical tree structures and have proven useful in experience with ranking algorithms [6] for search results.

We identify three application domains with examples of graph metadata:

- Mentioned above, the SDSS catalog metadata contains some hierarchical organization with additional relationships throughout. Notably, the catalog contains a network of neighbor relationships between pairs of photometric objects. All these relationships are conventionally represented using relational tables.
- The Livermore Entity Extractor [7] (Lextrac) was designed as a benchmark for text analysis. The application uses text document corpuses, which are processed within the extractor application to find significant entities and entity co-occurrences. These are significant pairs of entities within a particular document with a distance metric. Their discovery results in a factor of ten greater volume of metadata stored over the original text data size. Relationships among the documents, entities, and co-occurrences form a graph that must be managed and queried. The relationship between documents and entities is bipartite, and this metadata appears difficult to partition.
- Metadata from HPC performance tools, such as OpenSpeedShop tool [8], related provenance information from the HPC application build environment, and source file version control are presently managed in separate schemes for each. The graph data model fits the interrelationship of code objects, data collected by the tool, and the provenance of the build process from executable, object binary files and source code. Moreover, there is no established methodology for developers to manage these types of metadata, so each developer may choose an ad hoc scheme of his own [9].

Applications also need to locate files, and POSIX paths are limited in that capability. Paths require exact knowledge of location, namely the directory and file name components, in contrast to a query interface that allows many combinations of terms or expressions that could result in desired data. It is important for such an interface to fit the data model, thus a query language for searching a graph data model would be appropriate.

Applications instead have employed their own solutions for file metadata management separate from file systems, often using relational databases given the strengths of the relational model, SQL language, and mature technology. The use of relational databases requires a schema and index configuration specifically for every application. When the application's data structures change, the database design must be modified. On the other hand, file systems are generally configured and tuned for the shared use of a variety of applications. Moreover, when applications must use two systems, the file system and the database, their dual use creates management problems. For instance, references to files within the databases must be updated to reflect changes in file systems.

In this paper, we discuss an exploration of our approach to the problem: the use of a graph data model for representing file system user-defined metadata and a query language for retrieval. The purpose of this approach is to provide management of user-defined file metadata along with data under a single file system interface, delivering a common service across applications. Applications would be able to offload their metadata management needs to the service, alleviating the need for their own solution. This arrangement would benefit applications by reducing their code complexity, by virtue of not having their own custom metadata management components. A second benefit is improved opportunities for interoperability among separate applications. For instance in HPC code development, developers would have the opportunity to consider the metadata from several performance tools, such as OpenSpeedShop, provenance from their build process and version information.

This approach we have named QMDS, which we envision as extending existing metadata services (MDS), adding **Q**ueriability. However, for the purpose of exploring the graph data model and query language for this paper, we have implemented a single-host FUSE file system service. We have limited the scope of our research to retrieving files and attributes of files according to user-defined metadata and relationships among the files. We consider changes to POSIX systems calls as a consequence of our approach of extending POSIX with a query interface, but file data I/O is not part of the work. Also, we do not try to solve general graph problems, such as a shortest path algorithm, nor are standard file system operations a focus of this work.

Prior experience in attempting to use a relational database for file metadata with a graph structure [10] has led us to our approach of implementing our own store for such metadata [11], and we have continued to pursue that approach in this work. Our performance results presented in section 6 show a validation of this approach, exhibiting superior performance of our system against that of a relational database.

The contributions of this paper are: (1) the design and prototype implementation of QMDS based on a *graph* data model; (2) the design and prototype implementation of the Quasar query language specifically designed for the graph data model; (3) findings from the static and dynamic analysis of three workloads using QMDS; (4) quantitative evaluation of a QMDS prototype compared to a hierarchical file system plus relational database. Our evaluation uses a workload from the Lextrac case study described above, and we use its metadata for examples of the data model and queries from our language throughout the paper. We have chosen it because of its relatively large metadata to data size and complex graph structure. We additionally contrast our data model with the use of a extended attribute based approach and the use of RDF triples. We evaluate those approaches in part of the evalutation.

2. Related Work

Vast growth in data has been accompanied by the need to grow individual file system namespaces, resulting in distributed file systems. Some only distribute the data, while the metadata management of files and the hierarchical namespace can be handled by a single host. However, advances in distributed metadata management have shown that a directory hierarchy can span multiple nodes efficiently [12]. This approach assumes a hierarchical name space with few non-hierarchical links.

The extended attribute (x-attr) API gives applications some ability to store their own metadata for files, but the API is not widely adopted. The API works on a per file basis, *i.e.,* attributes are retrieved given a path to a file, but lacks query functionality for files based on attribute values. Semantic File Systems [13] provided that functionality by placing attribute-based naming into path expressions. The Logic File System [14] provided and interface that use boolean algebra expressions for defining multiple views of files. However, these concepts have not been adopted into mainstream file systems. Other approaches have a separate systems interface to handle searching and views of files, namely the Property List DIRectory system [15], Nebula [16], and attrFS [17]. Some systems combine POSIX paths with attributes [18, 19] and directories with content [20]. Most recently, Prospective provided a decentralized home-network system that uses semantic attribute-based naming for both data access and management of files [21]. While many of these system maintain the equivalent of extended attributes on files before these became part of POSIX, none provide edges to denote relationships among files.

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. Search within file systems has, in practice, often relied on command-line utilities such as ls, find and grep.

Examples of searchable file systems using relational databases and keyword search engines include Apple's Spotlight[22], Beagle for Linux [23], and Windows FS Indexing [24], where Spotlight also includes application-defined attribute-based searches for files. These systems provide full-text search, have indexing subsystems that are separate from the file systems, as opposed to index management within the same module or process as file system metadata management. A recent experimental file search system, Spyglass [25], 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, but the research focused on traditional file attributes. None of the above systems allow for search over relationships.

PASS [26] proposed how provenance data could be managed behind a kernel-based interface and tied into the file system. Their model includes relationships among files, but they do not keep name-value pair attributes on these relationships. They restrict the scope of the relationships to the provenance domains. We propose a metadata store that manages *any* conceivable relationship between pairs of files.

Holland *et al,* propose a query language for Provenance [27]. They choose the Lorel language [28] from the Lore system as a basis for provenance. The Lorel data model (OEM) differs from ours, as it requires class definitions and treats attributes and data nodes as the same.

Many researchers have proposed graph query languages [29]. Of those that have been accompanied by backing implementations, relational databases were used to implement the languages. Experience with a relational database and graph language have not yielded suitable performance [30]. Moreover, there has been a trend towards the use of graph-based data model over relational systems due to inability of RDBMS to manage graph-structured data [31] and the growing interconnectedness of real-world data [32]. In the introduction to this paper, we identify several applications which use relational databases to represent what is essentially graph structured metadata, however, these applications are configured with schemas specific to their needs rather than a general graph schema.

There are several RDF triple stores, for instance, 4store [33] and RDF3X [34], which support the SPARQL language [35]. However, RDF is a different data model than what we are proposing in that RDF has only nodes and edges; there is no concept of an attribute. Moreover, these engines do not support range queries,

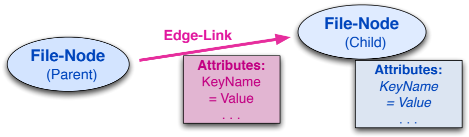

Figure 1. The QMDS graph data model contains files, links and attributes.

which makes their use very limited for our workloads. Neo4j [36] is an example graph database system that features a simple Java API. This approach requires that application developers program graph queries and handle complex operations, rather than use a robust query language interface that removes the complexity and handles optimizations for the application. The study performed in [37] describes some strengths, but points out some critical shortcomings of Neo4j.

3. Logical Design

Our goal in exploring QMDS is to examine its potential for the analysis and management of text, scientific and provenance metadata. In this section, we describe our logical data model and query language for QMDS.

3.1 Data Model

Our data model for file system metadata is a directed graph with attributes on nodes and edges, shown in figure 1. Nodes in the graph can represent files, and this allows the system to manage relationships among files. We call our directed edges *links*, connecting parent and child nodes. Attributes are name-value pairs, like POSIX extended attributes. These may be placed on both nodes and links. Moreover, multiple edges are permitted between any pair of nodes. Figure 2 shows example file metadata structured using these constructs from the Lextrac example domain. 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.

No application-specific "schemas" need to be explicitly defined for nodes, edges and attributes, and no classes must be defined for node objects, as one would need in most object-oriented systems. This gives a degree of flexibility for applications to change their metadata requirements and allows for heterogeneity within a single system. A heterogeneous approach to managing metadata gives all applications the same tools to manage relationships.

File attributes include the name of the file and are not necessarily unique. Moreover, none of the attribute values must be unique for a particular attribute name, except for system-assigned IDs on files and links. This provision allows for different files using the same name in multiple views, such as for different applications, users or versions. File nodes act as directories using links to represent directory membership.

3.2 Other Data Model Approaches

To contrast our graph-based data model, we consider two other data models which could be used for QMDS. We look at, first, extended attributes and second, the RDF "triple" data model.

3.2.1 Extended Attributes

Extended Attributes allow for the use of arbitrary name-value pairs attached to files. In contrast to a graph data model, there is no defined construct for relationships. Extended attributes could be used to store the relationships, but that sort of metadata requires some logical overhead for management on the part of application.

Meta-attributes are required to store sets of relationships. In the example below for Lextrac we have found that we need two layers of set identification for particular attributes. Then, because we have attributes on the links, those attributes need to either (1) encoded in a single BLOB attribute-value on the file attribute, or (2) each link attribute represented as a separate file attribute with a specific attribute name-based encoding scheme.

For the BLOB encoding approach to link representation, the file attribute name might be specific to the relationship, but for a general solution for relationships, we might use two relationship-specific meta-attributes per file: one for parent link count, another for child link count. The meta-attributes will be overwritten frequently during application ingest of metadata if the application adds many links to particular files. Moreover, in order to have the link attached to both the parent and child file, such link metadata as extended attributes must be attached to both.

Another alternative for using attributes is to encode in multiple attributes any information attached to QMDS data model nodes to be added and attached to data file nodes of a particular application. This approach would not have the overhead of representing the links and creating additional files, but instead, stressing the extended attribute API. Additionally, there are no duplicated attributes for representing links on both parent and child files.

As an alternative to representing Lextrac using the QMDS graph data model, we have attempted to represent the metadata using extended attributes on files. The files are the traditional document files. For Lextrac, we do not need to create relational links between pairs of files. However, we do need to manage entities and co-occurrences. We opt to store these items within extended attributes as sets. Managing the entities requires a meta-attributes to enumerate the sets of entities, where each set is generated by a particular extractor module within the Lextrac application pipeline. Then, we have a meta-attribute for each set, providing the count (hence, the two-layer approach mentioned above). We also have a meta-attribute for the co-occurrences to store the count. In section 6.1 we present the results of ingest of this arrangement in comparison to the use of QMDS data model for Lextrac ingest.

3.2.2 RDF

The key distinction between the RDF data model and the graph-based data model that we have chosen for QMDS is the use of higher-level constructs. RDF uses triples to represent a graph of all concepts. In contrast, the QMDS data model makes a distinction among the following three concepts: nodes, edges and attributes. Nodes and edges, the higher-level constructs each may have attached attributes. The components of triples are the basic constructs. Verbs (RDF graph edges) in conjunction with objects (the RDF graph nodes) are used to describe the subjects (also RDF graph nodes).

To map the QMDS data model, one must define RDF nodes for every node and edge. Thus a system (application or service) must define the ids. Then triples are used for the following:

(1) Connect RDF nodes representing QMDS graph nodes (parents) to RDF nodes representing QMDS graph edges.

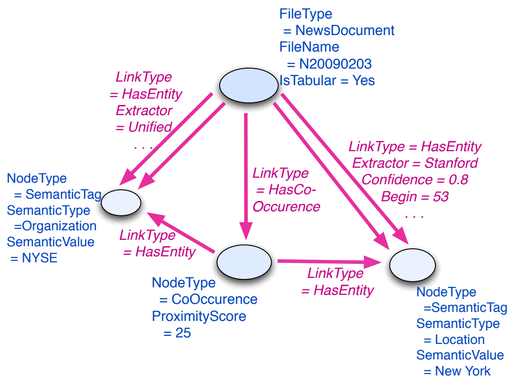
FileType = NewsDocument
FileName = N20090203
IsTabular = Yes
LinkType = HasEntity Extractor = Unified . . .
LinkType = HasCo-Occurence
LinkType = HasEntity Extractor = Stanford Confidence = 0.8 Begin = 53 . . .
NodeType = SemanticTag SemanticType =Organization SemanticValue = NYSE
LinkType = HasEntity
NodeType = CoOccurence ProximityScore = 25
LinkType = HasEntity
NodeType =SemanticTag SemanticType = Location SemanticValue = New York

Figure 2. An example of files (nodes), links (edges) and attributes from the Lextrac case study application. Circles represent files, arrows represent links.

(2) Connect RDF nodes representing QMDS graph edges to RDF nodes representing QMDS nodes (children).
(3) Connect RDF nodes representing both QMDS nodes and edges to RDF nodes representing QMDS attribute values. In this circumstance the triple "verb" or RDF edge holds the QMDS attribute name.

In above cases (1) and (2), the RDF verb describes the generic relationship of either being the parent or child component of the QMDS edge.

The use of RDF triples to represent QMDS graph data is problematic in several ways. A collection of triples must be used to represent concepts which can be represented with other higher-level abstractions. Additionally, having the single concept of RDF nodes have to represent multiple concepts — QMDS graph nodes, edges and attribute values — adds confusion and ambiguity, as one can't automatically know which type of concept is being represented. To further evaluate the use of RDF, we discuss an example of its use in practice in Section 6.3.

3.3 Query Language

Our query language, called Quasar, provides retrieval capabilities within graph-structured file system metadata.The query model centers on processing of sets, in which elements of the sets are files (nodes of the graph). Sets can be identified by particular attributes (*e.g.,* all .pdf files) or the parents or children of a particular file. Each query produces a *result set* of nodes, which in context of file system metadata are virtual directories. Traditional set operations from set algebra can be applied to sets of files, these operators being set *intersection, union* and *set difference.* The other operations, *attribute match, neighbor match*, and *navigation* are specific to the query language and described below.

3.3.1 Operators

The *attribute match* operator, indicated by the MATCH keyword provides for identification and conjunction based on sets of files, each denoted by set elements that contain the attributes specified in the query expression. The match can be a single, a range, or a set of specified attributes. Conjunction of sets through the match operator allows for attribute-based refinement of collections of files. We provide for the conjunction of multiple attributes within a single match operator clause using the semicolon meta-character.

The *neighbor match* operator enables refinement of a set of nodes based on the condition of particular parents or children to the nodes within the set. In a simple case, a neighbor match operator might refine a set based on a particular attribute on the parents or children of the nodes in the initial set. A Quasar expression using a neighbor pattern match looks like:

```
MATCH FileType = NewsDocument CHILD
{ MATCH SemanticType = 'Location' }
```

where an input set containing files with [FileType, NewsDocument] are filtered to only those whose children match [SemanticType, Location]. A pattern match operator may also specify constraints on edges to parents or children based on edge attributes. The keywords PARENT or CHILD indicate neighbor pattern mach expressions. The brace meta-characters hold a sub-query

The *navigation* operator manipulates one or more elements in a set (node of the graph) by the action of "following edges." Navigation can go either in the direction of the edge (from parent to child), or vice-versa. Navigation can be constrained to only follow edges that meet particular criteria, as specified via attributes placed on the particular edges. 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

```
MATCH FileType = 'NewsDocument'
NAVIGATE Extractor =Unified
```

will change the result set from all files with [FileType, NewsDocument] following links with the attribute [Extractor, Unified].

For attribute retrieval, the language features presentation functionality that returns attributes of files or links in a tabular-string format, as is common functionality for various query languages. Attribute names are specified with OUTPUT keyword-based clauses within the queries. Each clause contains the name of one ore more attributes of whose values will be returned in the tabular output. Each row in the output corresponds to an element in the result set and each column contains an attribute value corresponding to the attribute name in the *output clause*. For example, the query expression:

```
MATCH FileType = 'NewsDocument'
OUTPUT FileName
```

lists all the files of [FileType,NewsDocument] by the values corresponding to their FileName attributes.

3.3.2 Examples

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][1] 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

[1]This example comes from our workload evaluation application (see Section **??**), 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.

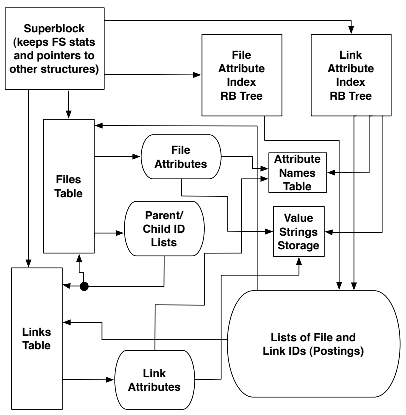
Superblock (keeps FS stats and pointers to other structures)
File Attribute Index RB Tree
Link Attribute Index RB Tree
Files Table
File Attributes
Attribute Names Table
Parent/ Child ID Lists
Value Strings Storage
Links Table
Lists of File and Link IDs (Postings)
Link Attributes

Figure 3. The schema of the QMDS metadata store is optimized for attribute matching, neighbor pattern matching, and navigation.

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:

```
MATCH FileType = 'NewsDocument'   ; IsTabular = 'yes'
CHILD LinkType = 'HasEntity'
{ MATCH NodeType = 'SemanticTag' ;
SemanticType = 'Location' } .
```

Similarly, MATCH FilleType = 'NewsDocument'
CHILD { MATCH SemanticType = Location ;
SemanticValue = 'New York' } OUTPUT 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.

```
MATCH FileName IN 'N20090201' ˜ 'N20090301'
NAVIGATE LinkType = 'HasCoOccurence' OUTPUT ProximityScore
```

The above query, first, matches files in the specified range (in this example files named by a date between February 1st and March 1st, 2009, and the IN keyword paired with the tilde meta-character indicates a range query predicate). Second, it traverses links from the matching source files (NAVIGATE), only following links that match the [LinkType, HasCoOccurence] attribute. Finally, it lists the resulting file set by the ProximityScore attribute.

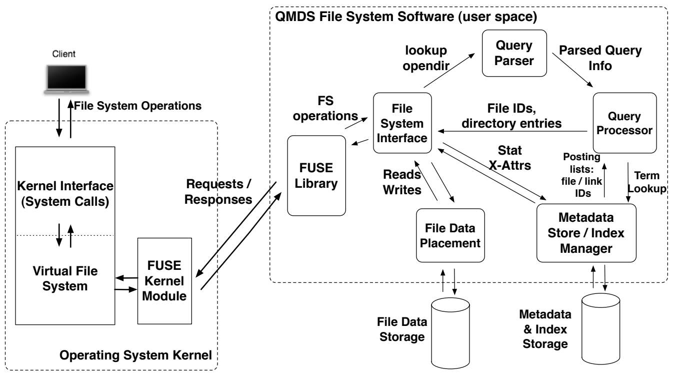

Figure 4. The QMDS prototype software architecture is a single-host file server exporting a FUSE interface that allows clients to the POSIX file system interface to pass Quasar expressions.

4. Service Design and Implementation

To explore the graph data model and query language in practice, we have implemented a prototype of QMDS. The prototype system runs in a single-host FUSE file system. Use of a single host for metadata is comparable to several distributed file systems used in production today that have single-host name nodes: HDFS, PVFS[1] [38], and Lustre [39] (in practice).

As an enhancement to POSIX, the query language interface works with existing file system operations: it provides file and virtual directory handles as responses to queries. Additionally, a "synthetic" file interface (comparable to Linux `/proc`) provides for efficient bulk metadata updates and access to attribute-oriented query results.

4.1 Overview

As shown in Figure 4 QMDS is implemented as a file server running in user space using the FUSE interface [40]. *Clients* pose standard POSIX file system operations to the *Kernel Interface* via systems calls. The *Virtual File System* forwards the requests to the *FUSE Kernel Module*, as is standard for mountable file systems. The FUSE client kernel module serializes the calls and passes the messages to the *QMDS Software* running in user space.

The *FUSE Library* implements the listening part of the service which receives the messages from the kernel and decodes the specific file system operations. The QMDS *File System Interface* implements handler routines for the various file system 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

[1]in PVFS, file metadata can be distributed but directories and file names are managed on a single host

been looked up. File data operations (read/write) go to a *File Data Placement* manager. In our QMDS prototype, this module maps file data to files stored within an underlying local (ext2) file system. Only non-zero length files[1] are represented in the ext2 file system. Zero-byte 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.

The software design strives to decouple its various software components to the greatest reasonable extent. The query processing module and parser must have access to the same structures in order to be interoperable. Parsed *query objects*, returned by the parser, consist of linked lists of parsed query operators. Each operator's linked list node points to the list of attributes. Neighbor match and set operators also point to a list of operators for the sub-query.

Result set objects are used internally by the query processor and also returned to the interface module. This object contains the result count, a union of several array datatypes for results, and an integer indicating the particular array datatype: (0) file/node id only, (1) file/node ID, link/edge ID pair (2) extended tuple of IDs (see end of Section **??**, (3) a list of pointers to lists of IDs (direct to metadata store address space). Type number *3* has a limited implementation within the query processor, *i.e.,* it is not handled by the routines to translate result sets to strings. These result set objects generally correspond to the logical Quasar result sets of each logical query operation.

4.2 QMDS semantics for directory/file operations

QMDS follows POSIX semantics as closely as possible, and extend the semantics as needed for operations that involve metadata and links (excluding read, write, fsync, etc.) In particular, as many file system operations require a pathname to a particular file, operations posed to QMDS may specify a "pathname query", which accepts any valid Quasar expression, including POSIX paths. A consequence of this change is that the semantics of some of the POSIX file system calls that concern metadata must change as well.

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

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

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

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.

[1]or zero-byte files without links

opendir Looks up the pathname query. For each query result, opendir looks up particular attributes to return for the result based on a "ListBy" operator in the query. Opendir 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 the next directory entry (or query result) in the result cache.

close(dir) Passes file handles to the 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, or (3) update the affected link(s) and their associated attributes. The pathname must resolve to a single source file.

unlink Looks up the 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, including all of its link attributes.

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 QMDS assigned immutable file or link IDs to address files (equivalent to inode numbers), as both are searchable attributes in QMDS, 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. Our Lextrac example uses this approach for the document files through its document file name attribute, which it controls. For an application to use the first approach, it would obtain each ID upon file creation, then store the ID in its own temporary data structures for use, in case the application were to modify the attributes of that file. We have not explored an application using this procedure.

4.3 Metadata Storage

Our QMDS prototype features a metadata store and index manager using structures tailored to our graph data model. The data structures of the metadata store are a collection of arrays, sorted lists, and red-black trees. We have chosen these structures based on experience with previous in-memory graph file system metadata management from the LiFS prototype [11] and search engine design for indexing [41, 42]. These data structures are backed by memory-mapped files in an underlying file system. Each type of data structure is assigned to a separate memory-mapped file, each with its own allocator. We have used configurations with five or thirteen separate files. The sizes of each file must be set prior to loading the the QMDS software module. This design is suited for storage class memories, as their use has been suggested for metadata storage [43], based on their lower latencies for random access, as opposed to conventional disk-based storage. 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 3).

The metadata store has a *Superblock*, which contains references to the other structures within the store and some global statistics used for query optimization, such as the total numbers of files or links. We place the superblock at block zero of the memory-mapped file containing the *File Table*. The File Table is a single-indirect array of pointers which point to arrays of pointers. The table size is based on a preallocated value, given the preset file size. Such is akin to setting the number of inodes available for a file system, based on available space on a partition. There are 512 arrays (4K / 8 byte pointers). When each block fills completely with pointers, an additional 4K-block array is allocated from the store. The second-level arrays contains the pointers to the file objects. For example for 100 million nodes, we require $\sim$ 800 MB of pointers and a 4K page for the top level. Instead of allocating the 800MB array in one go, we incrementally allocate $\sim$ 1.5 MB arrays, to which the entries in the top-level 4K array point.

Likewise, the *Link Table* maps link IDs to each *Link Attribute* list using the same singe-indirect structure. Unlike the File metadata, the entries in the link table contain a count field and a pointer to an array of the list attributes. The attribute lists can increase in length beyond the initial ingest, given that there is a reserved pointer at the beginning of the array-list block. In the event that attributes are added to existing files or links, the system can add additional blocks for new attributes.

The file objects, similar to *inodes*, each include lists of *File Attributes*, pointers to 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.

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)*. We integrate the libavl rb-tree library [44] to handle tree inserts and lookup operations. We have implemented our own range query predicate index scan procedure that uses a queue to temporarily store subsequent nodes to process. The procedure, first, finds the leftmost tree node that fits the range, and then, traverses nodes post-order [45] until the rightmost node is found. Values for all matching nodes are returned in an array of pointers to posting lists of file/link IDs.

File/link attribute names within the file/link tables and indices contain string references to entries in the *Attribute Name Table*, a hash table using a commonplace hash function. String attribute values refer to a common *Value Strings Storage* shared with the indices. When any file or link attribute is added, the software determines if the attribute name exists in the attribute name table via a standard hash function, and determines if an attribute string value is present via an index lookup. Numeric values (integer and floating point) are stored directly in the file attributes.

The FS interface module does not make any direct calls to functions within the MS/IM module. The MS/IM data structures have a simple interface to facilitate modification to the underlying storage scheme. For instance, trees should support inserts, delete, and point/range lookups. Lists of file/link IDs are iterated through as their means of access, as within QMDS query processing, there is no need to search for one particular item within a list of IDs. The attribute lists are accessed through a linear search interface; we have determined that even up to 452 attributes, linear search is more efficient than `glibc` binary search.

The list structures use the most complex memory allocation scheme. A chunk allocator maintains a pool of available chunks of some number of sizes. The tail node of the list grows by swapping for the next sized chunk when the current one has filled up, until it reaches the maximum size. Then, it points to a new node starting with the smallest size, and the process repeats for subsequent additions. Thus, each list has a linked list of maximum sized chunks up to the tail node, which may or may not be a smaller sized chunk. Attribute index lists with four or fewer elements and parent/child lists with two or fewer are stored in the referring node structure within the index RB tree or attribute node storage respectively.

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

4.4 Query Planing and Processing

In this section, we discuss several scenarios by which we optimize the processing of Quasar queries. For a first optimization scenario, consider the match operator: single Quasar match operators find the search attribute name and value in the file attribute index tree. Once the attribute structure is located, the list of matching file IDs is returned. In the case of match operators with multiple attributes, the query processor determines the best of the following two strategies:

(1) multiple lists should be intersected (computation time $O(n_1 + n_2)$, where n_1 and n_2 are the lengths of lists)
(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 $O(K \times n_1)$).

In order to optimize for the two strategies shown above, in cases where there are multiple attribute query terms for an individual query operation, the query planner orders the terms from smallest to largest based on the numbers of nodes or edges that match the particular attributes. This process enables the query processor to select the more efficient of the two strategies when multiple terms are encountered in a query match operation. In our current prototype implementation, both strategies and the query processor's ability to select which to perform are implemented only for attribute match query operations.

Filtering by attribute terms on navigation and neighbor match operators only use strategy (2). However, the query planner can estimate the potential cost for those operators based on information from the index reflecting node or edge counts of attribute query terms, including range query predicates. Each operator is assigned a score. Then, the operators might be reordered, based on a heuristic comparison function that takes the scores as input. We use a simple product of one input with a configurable constant value and evaluate the performance of various constants in Section **??**. This behavior is analogous to query plan selection that occurs with the query optimization of relational databases. In addition, the query processor may merge ordered lists with either linear processing of the lists as described in strategy (1) above, or binary search of the longer list by items in the shorter list [46]. Statistics of the index maintained after every metadata ingest operation determine which strategy to use.

5. Common Workload Patterns

In this section we briefly describe results from several workload analyses. Our primary goal is to evaluate the effectiveness of the data structures chosen to organize and index QMDS metadata. In summary, our findings do not contradict the design choices made for QMDS. Our approach to these analyses is to consider both static and dynamic analysis of QMDS using the three example domains mentioned in section 1, namely the text analysis application (Lextrac), SDSS and OpenSpeedShop metadata. We give a summary of the approach to our analyses, our observations, and the implications of our findings.

5.1 Approach

We employ several static techniques to examine the properties of metadata stored in QMDS. To determine the storage requirements per component data structure, we retrieve the amount of storage used by each. The file system that stores files for each data structure component (one file for each) can report the number of blocks stored per file. This information is simply retrievable using the commonplace UNIX **du** utility.

Particular metadata item counts that we measure have been directly extracted from QMDS Metadata Store files through a utility that memory-maps the files and crawls the data structures. The QMDS software maintains these counts, as they are used in query processing. Thus, they are easily reportable. The specific counts that we report are: nodes; edges; total attributes (name-value pairs); unique attributes; the numbers of nodes and edges per unique attribute; the in and out-degrees for the nodes within the graph structure; the numbers of attributes on nodes or edges. We present the latter three categories of counts as distributions. Because each unique attribute is found on multiple edges, the total attribute counts are considerably larger.

Our approach to the dynamic analysis of the QMDS metadata store and query processing uses traces of queries from the three workloads. The traces that we have collected are comprised of *load* memory operations specifically targeted to the address ranges for the persistent data structures. Other load operations that target temporary structures used in query processing are ignored. Each access in the trace has a *sequence number* and the accessed memory address (in the QMDS process's 64-bit virtual address space). Additionally, the accesses are grouped by distinct queries in the workload and by their target data structure component.

We use the Valgrind [47] utility to handle the execution and to output every load instruction to a temporary file. Next, the temporary file is processed: each load instruction is kept if the address falls into one of the thirteen ranges. Then, we write out the ID of the component and the relative address for that particular access. We use an instrumented version of QMDS, which outputs the base addresses for each of our memory-mapped files and the boundaries of each individual query execution.

To address the issue of reducing storage usage, we quantify how the various data structures that we have chosen might benefit from common lossless compression techniques. Use of compression is well-established for building full-text indices, as such indices can grow to become larger than the original data. We show how one might mitigate such an issue through the use of two common lossless compression techniques: *gzip* and *bzip2*.

5.2 Summary of Measurements and Observations

Our main findings are as follows (particular findings are designated in *italics*). Unless specified, these are common, observed patterns across all the workloads.

Majority attributes - Of all the metadata stored, a majority of storage is dedicated to the node/edge attribute list structures.

strings smallest - Strings take up the least storage compared to other data structures.

Index and *graph-degree distribution* - The distributions of both the number of nodes or edges to unique attributes follows a power-law distribution power. The same can be observed for the in and out-degrees of the metadata graph patterns.

Few attributes - pertaining to the storage of attributes placed on nodes or edges

Tree percentage - With a small exception, the fraction of total accesses to tree data structures is under ten percent.

SDSS strings - For the SDSS workload, string accesses are in the fifty to eighty percent range. They remain a significant fraction for the other workloads as well (but not to the same extent).

Super-hot pages - We found that there are on the order of ten pages with access counts several orders of magnitude larger than the next most frequently accessed pages.

Hot page distribution - The remainder of accesses to pages, irrespective of the particular data structure, evenly spans from tens to tens of thousands.

Intra-page locality - The trees and list data structures have a high degree of locality within a single page, *i.e.,* once a page is loaded, there is a good chance it will be accessed very soon again repeatedly.

Random access - Many access patterns are inter-page, and those appear to be random, even within particular data structures.

Compressibility - Gzip compression does not yield impressive compression ratios. However, we observe quite a number of highly compressible data structures using bzip2 compression. Notably, the node and edge attributes compress by factors of at least ten for two of the workloads In addition to the results shown in the table, we have measured that the Lextrac 100K corpus metadata compressed from an original size of 5.4 GB to 750 MB overall: a compression ratio of 7.2.

5.3 Discussion

This discussion of the analyses focuses primarily on the various data structures employed in QMDS. Additionally, we discuss general systems configuration issues. We evaluate each data structure based the observations presented in Section 5.2

Given our *strings smallest* finding, we surmise that a loss of locality in the Strings component due to hashing for string deduplication is not significant. In addition, from our *SDSS strings* finding, we realize the importance of string access in our workloads. Thus, the String data structures are good candidates to be maintained in cache for lower latency of access, as opposed to paging to and from secondary storage.

The File and Link Tables — arrays of pointers to the attributes for the File/Nodes or Link/Edges — benefit from our *intra-page locality* finding. The use of an LRU cache replacement policy should maintain that the single indirect block pointers (found at the front of each of these structures and one of the *super-hot pages*, will remain in cache, given the frequency of access (observed in the *hot page distribution* finding.

The Attributes structures are simple array-based lists of the name-value paired attributes. From the *few attributes* finding, we determine that the use of array-based lists for attributes fit the common case of a small number of attributes on each node or edge. Access to the Attribute component are subject to our *random access* and *majority attributes* findings. Thus, we should least expect to find useful co-located data within larger sized workloads, increasing a need for paging from storage suitable for low-latency random access.

The Index and Adjacency Lists both use the same chunked-list structure. The use of these structures is supported by the *intra-page locality* and *random access* findings, where the random access is at the page level. There is no need to search these lists in the current query processing model. The *index* and *graph-degree distribution* findings suggest several properties for lists. First, because lists of the smallest sizes are stored directly in the node metadata, no additional paging or pointer dereferencing is needed to access such lists. These are the most frequently accessed and befittingly provide the highest efficiency. Second, maintaining a pool of smaller chunks satisfies the mid-range sizes with moderate frequency of access. Third, longer lists of larger chunks are accessed with the least frequency, but should not require the greatest efficiency.

Finally, the RB trees exhibit *intra-page locality* of access in all traces studied except the query trace from the Lextrac workload . We attribute the lack of overlapping data for this workload to the fact that pages in the trees do not co-locate the necessary data. Other tree data structures could be selected, but for this workload, given the favorable performance we measure, a balanced binary tree appears to be reasonable. Our *tree percentage* finding suggests that efficiency of a specific tree data structure will not dominate overall query performance. There is no harm in the use of a particular tree data structure that performs more efficiently than the other structures, but gains in efficiency for the Index Tree component will not have a significant overall impact. Moreover, from the *hot page distribution finding*, given that tree sizes of the Lextrac 800,000 workload are larger than those of 100,000 document corpus workload, the upper levels of the trees have node structures (closer to the root of the tree) on particular pages that are accessed more than the pages that contain the structures found on the lower rungs. In contrast, for the smaller workload, it is more common to find the upper and lower rungs within the span of the same page. The LRU cache policy should, in theory, preserve these hotter pages in memory, if the lower rungs of the tree must be swapped in and out of storage.

One key finding that might influence system and hardware configuration for QMDS is our *random access* finding. Therefore, the common practice of prefetching data from storage should not be beneficial to QMDS metadata when the memory available for buffer cache is limited. Additionally, the *small working-set* finding suggests that prefetching would not be beneficial because, in all likelihood, whatever additional data is prefetched, based on the accesses in the initial query within a given workload, will not be accessed by a subsequent query.

6. Evaluation

In this section, we report on results of ingest and query experiments using the Lextrac application. In its original configuration, Lextrac writes document metadata to conventional files in initial analysis phases, followed by a phase that writes the data to a searchable SQL database. We use PostgreSQL 4.3 in our experiments, and refer to that configuration as FS+DB. To produce a scalable ingest, we have configured PostgreSQL with a schema specific to the Lextrac metadata structure and create indices on several of the columns. To properly generate the graph structure, the ingest application must leverage the index and query functionality of the metadata storage system (the SQL database), so incremental indexing is important.

We extended Lextrac so it supports the QMDS storage interface in addition to the POSIX I/O interface and can take full advantage of the QMDS data model.

System	QMFS	FS+DB
Ingest Time (Hours)	23.78	58.95
Metadata Size (GB)	43	30
Temporary Size (GB)	0	14

Table 1. Properties of Lextrac metadata storage (QMDS and FS+DB), 800000 Reuters documents. The graph size is approximately 100 million nodes and 350 million edges, while the original text data size is 3.8 GB. Thus, the metadata size is roughly an order of magnitude larger.

This is an example where an application (*i.e.,* Lextrac) can offload its metadara management needs to QMDS. In the second configuration (labeled "QMDS"), we have replaced the file system and SQL database with our QMDS prototype (with its backing store also acting as a store for file data). In this configuration, the final SQL database building phase is unnecessary.

Our evaluation was conducted on a Dual-Core AMD Opteron 2.8 GHz, 4 socket server with 32GB main memory running Linux kernel version 2.6.18 with a 250GB SATA drive configured with an ext3 file system, shared by all configurations for file and metadata storage. 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 provide suitable SQL query performance. (Experiments with PostgreSQL without indices have resulted in performance so uncompetitive, that a comparison would best be characterized as a "straw-man".) In addition, we run the database without transactions or isolation.

6.1 Ingest

Table 1 shows the results of our document ingest experiments in which the Lextrac application processes a Reuters News corpus containing *800,000* documents (the full corpus available contains very slightly more). The application configured to write to QMDS completes in close to 2.4 times faster than the FS+DB configuration.

While the storage space required for QMDS metadata is somewhat larger, it does not need the additional space overhead for temporary files to write metadata prior to writing that metadata into PostgreSQL. Moreover, we are aware of instances in which space might be better used within QMDS metadata storage. As we run on 64-bit architectures, we store 64-bit pointers within our data structures. Much space savings should be possible though storage of smaller, relative addresses. We consider this an implementation issue that can be resolved as future work. In addition, we have determined that many of our data structures are compressible using standard techniques.

We have unsuccessfully attempted to implement the graph data model using a SQL database. We configured PostgreSQL with a general schema for the graph data model consisting of four tables: files, links, file attributes, link attributes, in contrast to the specific schema that we created for Lextrac. Standard indices were configured for all columns within the tables. However, the ingest times were not able to scale. The largest size attempted with this configuration was *20,000* documents in the Lextrac reuters corpus, and its processing takes over 24 hours. In contrast, the PostgreSQL schema configured specifically for Lextrac takes about a half-hour for processing with that sized workload.

In another attempt to use common database software to represent and store our graph data model, we configured our prototype to use BerkeleyDB as a storage layer for metadata and indices. The purpose of constructing a BerkeleyDB implementation was to evaluate the benefit of using a highly optimized library with support for out-of-core data structures as the underlying QMDS access mechanism. While the structures appeared to exhibit scalability for small numbers of documents, our data ingest experiment did not complete after 36 hours of processing with a *100,000* document workload. These results reinforce our position to develop a custom metadata store and index manager for a graph data model.

We briefly consider the implementation of QMDS using an extended attribute-only data model (no links), as discussed in Section 3.2.1. We evaluate this alternative using a modified Lextrac ingest application workload. This modified application version encodes all metadata pertaining to entities and co-occurrences, including the nodes, edges and attributes used in QMDS, as attributes attached to the document files. We use the Quasar interface for attribute access, but we only use a subset of the language features that pertain to attributes attached to files. Overall, we observe a speedup factor of close to 2 for the use of links and attributes for Lextrac ingest over the use of attributes only.

6.2 Query Experiments

The query study uses query templates $Q0 - Q4$ representative of queries that would be applied to the document set. Below we describe these query templates with examples that follow Figure 1. Our goal in selecting these query scenarios is to stress both searches for files and semantic querying capabilities of the metadata managed by QMDS: of the query templates presented here, $Q0 - Q1$ return files and $Q2 - Q4$ return metadata items.

$Q0$ Find all documents that are linked to a particular entity. Example: Find all documents linked to a place “New York.”

$Q1$ Find all documents that link to both entities X and Y that have a particular proximity score between them. Example: Find all documents that link to a place “New York” and organization “NYSE” with co-occurrence of proximity score “25”.

$Q2$ Find all entities related to entity X in documents with names in a particular range and whose proximity to entity X has a score of Y. Example: find entities co-occurring with “New York” in documents with names in the range “N20090101” – “N20090331” whose proximity score with “New York” is ”25”.

$Q3$ Find all proximity scores within a particular range relating two particular entities in documents with names in a particular range. Example: find the proximity scores in the range of ”20” – ”30” relating “New York” and “NYSE” in documents with names in the range of “N20090101” – “N20090331.”

$Q4$ Find ALL proximity scores (no range constraint unlike $Q3$) 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.”

An example of a $Q0$ query in Quasar:

```
MATCH SemanticType = 'Location' ;
SemanticValue = 'New York'
BACKNAV MATCH FileType = 'NewsDocument'
OUTPUT FileName
```

A more complex $Q1$ query example:

```
MATCH SemanticType = 'Location' ;
SemanticValue = 'New York'
BACKNAV MATCH Proximity = 25
CHILD { MATCH SemanticType = 'Organization' ;
SemanticValue = 'NYSE' }
BACKNAV MATCH FileType = NewsDocument
OUTPUT FileName
```

All query classes contain SQL queries and corresponding Quasar queries with the same general language-specific clauses. For each SQL query, we select literal values for the WHERE clauses and then use the same values for the corresponding Quasar query.

For the query workload experiment $Q0$, literal query terms (the entity values) were selected 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 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 should favor the majority of entity-value terms, which correspond to relatively few documents. Specifically, our query test suite for $Q0$ selects entity values based on combining the arithmetic and geometric series, split evenly between each.

For $Q1-Q4$, We follow a different procedure to generate their query terms, which might be one or two entity values, a proximity score, range of scores or range of documents. For each query, we randomly choose a document co-occurrence, that is a 4-tuple consisting of a document, a pair of entities, and the proximity score for the pair. This procedure guarantees us at least one valid result.

These experiments were run with the *800,000* document corpus used in the ingest experiments. We only used queries that have at least one result returned in the measurements discussed below. QMDS answers empty-result queries 10 to 40 times faster than PostgreSQL, and this we attribute to an optimization we made in the QMDS query planner to verify all attribute search terms in the index prior to processing the remainder of the query.

Figure 5 shows the mean query response times for each class[1]. Each distinct query was executed 10 times, and we measured 3% relative standard deviation across all classes of query. However, the variance for each class is due to different processing times from the use of a variety of literal values within each class. We have written in the very large values that would not otherwise fit into the chart with a reasonable y-axis scale to fit most of the values.

For our simple class of query, $Q0$, QMDS answers the queries 4 times faster on average than PostgreSQL. We attribute the QMDS speedup for $Q0$ to the use of the navigation operation over the relational join operations needed to combine tables and arrive at the relevant results.

[1]These measurements were made on "warm" systems, both with QMDS and PostgreSQL.

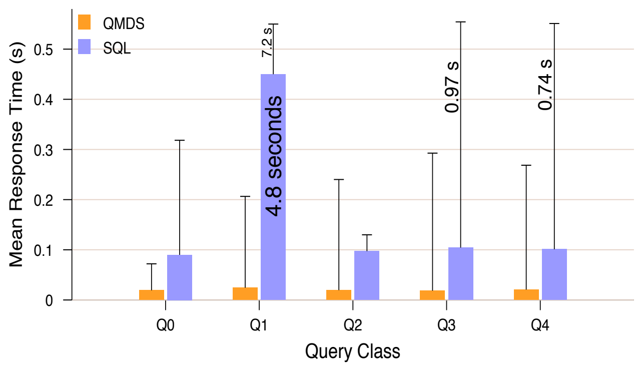

Figure 5. Mean query times for our five classes of queries comparing QMDS (left bars) with PostgreSQL (right bars).

For $Q1$, more complex than $Q0$ in terms of query operations (joins in SQL), also locates document files. QMDS answers the query on average 200 times faster than PostgreSQL. We suspect that this class of query is particularly tough for the SQL query optimizer due to a relatively very large table of proximity scores, where there are tens of millions of rows. Indexing this table can only help a little because there are fairly few (about 50) individual score values in the index. The database, then must scan all the matching values and join those with the other tables. In contrast, the QMDS query processor simply navigates to the entries corresponding to nodes in the graph with the matching values, producing the observed speedup.

In the other three cases, $Q2 - Q4$, the queries run on average about 5 times faster using QMDS than using PostgreSQL. These are semantic queries that return particular document metadata: entity values or entity proximity scores. For these cases, PostgreSQL leverages the document ranges to produce better query plans unlike $Q1$ with no doc range, but as with $Q0$, the SQL joins do not perform as well as QMDS navigation. In summary, as query complexity increases in terms of the number of query operations, QMDS query performance remains roughly constant, while PostgreSQL must spend additional time processing the queries.

The error bars in the figures show that both systems have some degree of variance to its response time. Results for $Q0$ in both systems can be directly attributed to the number of documents that match the requested entity in each query, subsequently corresponding to the number of results returned. For $Q1$, there are two entities involved in the query, so the sum of the number of documents corresponding to each entity is correlated with an upper bound to the response time, while there is a constant lower bound. The error bar $Q2$ for PostgreSQL is shorter than all the others, and we attribute this to nature of the SQL query and the data. The query planner leverages a small number of documents in the specified range, and unlike the other complex queries, there is only one Entity term to match, yielding a less expensive join in most cases.

6.3 RDF-3X Queries

To consider the performance of an RDF SPARQL engine we have selected RDF3X and converted the Lextrac relational tables of the *40,000* document corpus to RDF triples. RDF3X can index the 22 million triples contained that corpus in about 10 minutes. By extrapolation, we would expect to find approximately 440 million triple to represent the metadata for a 800,000 document workload. Note that this value is significantly smaller than what we would expect for converting the metadata constructs from the QMDS graph data model to triples, as described in Section 3.2.2. For the largest Lextrac workload we consider, the total number is over a billion triples.

Several factors account for the discrepancy noted above. First, using a Lextrac-specific ontology saves spaces over a general QMDS graph data model ontology. The general ontology must convert some application "type" identification attributes to triples, which can be eliminated when using an application-specific ontology. Also, the Lextrac-application specific schema does not maintain a sequence number attribute, which is used to maintain the provenance of which entity is selected first within a co-occurrence. This information in $FS + DB$ is present in the flat files written by the application for temporary metadata storage, but not needed for queries (thus omitted from the searchable relational and RDF databases). We expect to find a couple hundred million of these attributes within the largest Lextrac workload.

RDF3X does not support range queries; thus, we could only faithfully use the $Q0$ and $Q1$ query classes. In attempt to support $Q2$ and $Q3$ for a simple comparison we use sets of documents that would otherwise appear in the range predicates. We observe the following times with some sample queries: $Q0$ - 7 ms, $Q1$ - 22-24 seconds, $Q2$ - 310-340 ms, $Q3$ - 42-44 ms. These values are all significantly slower than PostgreSQL running the same workload, which, in turn, exhibits slower on-average performance than QMDS. Note that these response times are measured using a 20 times smaller Lextrac workload than that used in the comparison between QMDS and PostgreSQL. Similar to what we observed in the QMDS comparison with PostgreSQL, we attribute the $Q1$ response time to our understanding that matching the proximity score is the most time consuming aspect of this particular query class. There are a large number of rows/triples for each co-occurrence given a relatively small range of score values (2-50).

7. Conclusion

In this paper, we present our rationale for managing user-defined metadata in a file system metadata service using a graph data model. We describe the data model, the query language and the metadata storage design of our prototype system, QMDS. Using a text document data mining application, we evaluate our prototype system's performance on ingest and query workloads.

Acknowledgements

This work is supported in part by the Department of Energy under Contract DE-AC52-07NA27344, the Institute for Scalable Scientific Data Management (ISSDM), and the sponsors of the UCSC Systems Research Lab (SRL). We thank Ethan L. Miller for his advice during the early stages of the work, John Compton of LLNL for his guidance in the design of the query templates, and our colleagues at LLNL and UCSC for their feedback.

References

[1] G. Bell, T. Hey, and A. Szalay, "Beyond the data deluge," *Science*, vol. 323, no. 5919, pp. 1297–1298, March 2009.

[2] M. J. Raddick and A. S. Szalay, "The universe online," *Science*, vol. 329, no. 5995, pp. 1028–1029, August 2010.

[3] A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg, "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: ACM, 2002, pp. 570–581.

[4] M. Seltzer and N. Murphy, "Hierarchical file systems are dead," in *HotOS XII*, 2009.

[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data.* New York, NY, USA: ACM, 2010, pp. 135–146.

[6] C. A. N. Soules and G. R. Ganger, "Connections: using context to enhance file search," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05).* New York, NY, USA: ACM Press, 2005, pp. 119–132.

[7] J. Cohen, D. Dossa, M. Gokhale, D. Hysom, J. May, R. Pearce, and A. Yoo, "Storage-intensive supercomputing benchmark study," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TR-236179, Nov. 2007.

[8] T. K. Institute, "Open—speedshop - overview," http://www.openspeedshop.org/wp/, 2011.

[9] M. Schulz, "Personal communication," November 2010.

[10] A. Ames, N. Bobb, S. A. Brandt, A. Hiatt, C. Maltzahn, E. L. Miller, A. Neeman, and D. Tuteja, "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. [Online]. Available: http://www.ssrc.ucsc.edu/Papers/ames-mss05.pdf

[11] S. Ames, N. Bobb, K. M. Greenan, O. S. Hofmann, M. W. Storer, C. Maltzahn, E. L. Miller, and S. A. Brandt, "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: IEEE, May 2006. [Online]. Available: http://www.ssrc.ucsc.edu/Papers/ames-mss06.pdf

[12] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04).* Pittsburgh, PA: ACM, Nov. 2004. [Online]. Available: http://www.ssrc.ucsc.edu/Papers/weil-sc04.pdf

[13] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr., "Semantic file systems," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91).* ACM, Oct. 1991, pp. 16–25. [Online]. Available: http://www.ssrc.ucsc.edu/PaperArchive/gifford-sosp91.pdf

[14] Y. Padioleau and O. Ridoux, "A logic file system," in *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, Jun. 2003, pp. 99–112. [Online]. Available: http://www.ssrc.ucsc.edu/PaperArchive/padioleau-usenix03.pdf

[15] J. C. Mogul, "Representing information about files," Stamford Univ. Deptartment of CS, Tech. Rep. 86-1103, Mar 1986, ph.D. Thesis.

[16] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti, "A File System for Information Management," in *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, March 1994, nebula FS.

[17] C. E. Wills, D. Giampaolo, and M. Mackovitch, "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.

[18] S. Sechrest and M. McClennen, "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.

[19] B. C. Neuman, "The prospero file system: A global file system based on the virtual system model," *Computing Systems*, vol. 5, no. 4, pp. 407–432, 1992. [Online]. Available: citeseer.ist.psu.edu/neuman92prospero.html

[20] B. Gopal and U. Manber, "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. [Online]. Available: http://www.ssrc.ucsc.edu/PaperArchive/gopal-osdi99.pdf

[21] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger, "Perspective: Semantic data management for the home." in *fast09*, M. I. Seltzer and R. Wheeler, Eds. USENIX, 2009, pp. 167–182. [Online]. Available: http://dblp.uni-trier.de/db/conf/fast/fast2009.html#SalmonSCG09

[22] Apple Developer Connection, "Working with Spotlight," http://developer.apple.com/macosx/tiger/spotlight.html, 2004.

[23] Beagle Project, "About beagle," http://beagle-project.org/About, 2007. [Online]. Available: http://beagle-project.org/About

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

[25] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "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. [Online]. Available: http://www.ssrc.ucsc.edu/Papers/leung-fast09.pdf

[26] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006, pp. 43–56.

[27] D. A. Holland, U. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. Seltzer, "Choosing a data model and query language for provenance," in *2nd International Provenance and Annotation Workshop (IPAW'08)*, June 2008.

[28] S. Abiteboul, D. Quass, J. Mchugh, J. Widom, and J. Wiener, "The lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, pp. 68–88, 1997.

[29] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1–39, 2008.

[30] I. L. Kaplan, G. M. Abdulla, S. T. Brugger, and S. R. Kohn, "Implementing graph pattern queries on a relational database," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-400310, January 2009.

[31] M. A. Rodriguez, http://www.slideshare.net/slidarko/graph-databases-and-the-future-of-largescale-knowledge-management, 2009.

[32] E. Eifrem, "A nosql overview and the benefits of graph databases," http://www.slideshare.net/emileifrem/nosql-east-a-nosql-overview-and-the-benefits-of-graph-databases, 2009.

[33] Garlik, "4store - scalable rdf storage," http://4store.org/, 2009.

[34] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[35] E. Prud'hommeaux and A. Seaborne, "Sparql query language for rdf," http://www.w3.org/TR/rdf-sparql-query/, 2007.

[36] N. Technology, "The neo database," http://dist.neo4j.org/neo-technology-introduction.pdf, 2006.

[37] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th Annual Southeast Regional Conference*, ser. ACM SE '10. New York, NY, USA: ACM, 2010, pp. 42:1–42:6. [Online]. Available: http://doi.acm.org/10.1145/1900008.1900067

[38] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for Linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, Oct. 2000, pp. 317–327. [Online]. Available: http://www.ssrc.ucsc.edu/PaperArchive/carns-linux00.pdf

[39] R. Hedges, B. Loewe, T. McLarty, and C. Morrone, "Parallel file system testing for the lunatic fringe: The care and feeding of restless i/o power users," in *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 3–17.

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

[41] E. A. Brewer, *Readings in Database Systems*, 4th ed. MIT Press, 2004, ch. Combining Systems and Databases: A Search Engine Retrospective.

[42] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes.* Morgan Kaufmann Publishers, 1999.

[43] A.-I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The Conquest file system: Better performance through a disk/persistent-RAM hybrid design," *ACM Transactions on Storage*, vol. 2, no. 3, pp. 309–348, 2006. [Online]. Available: http://www.ssrc.ucsc.edu/PaperArchive/wang-tos06.pdf

[44] B. Pfaff, *An Introduction to Binary Search Trees and Balanced Trees.*

[45] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* New York: The MIT Press, 2001.

[46] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *Annual Symposium on Combinatorial Pattern Matching (CPM)*, S. C. S. et al., Ed., vol. 3109. Springer, 2004, pp. 400–408.

[47] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746