Quasar: A Scalable Naming Language for Very Large File Collections

Technical Report UCSC-SSRC-08-04
October 2008

Sasha Ames
sasha@cs.ucsc.edu

Carlos Maltzahn
carlosm@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

Quasar: A Scalable Naming Language for Very Large File Collections

Sasha Ames
sasha@cs.ucsc.edu

Carlos Maltzahn
carlosm@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
University of California Santa Cruz
Santa Cruz, CA 95064, USA

ABSTRACT

As storage capacities increase, managing petabytes of data becomes increasingly challenging. One reason is the POSIX file system interface, originally designed in the 1970s in the context of file collections many orders of magnitude smaller than those found in today's petabyte-scale storage systems. We show the scalability problems of the *naming language* imposed by POSIX, i.e. the language to identify an individual file or a group of files. We identify common features of popular applications that manage large file collections as *search*, *attributes*, and *relationships*. The increasing size of file collections has already motivated file system designers to include support for these features, so highly optimized implementations can be shared across all applications. Existing approaches treat these features as add-ons to the POSIX naming language. One consequence of this lack of integration is that searches cannot be scoped to a fragment of a file system name space, which makes search hard to scale to very large file collections. We present a naming language (Quasar) that offers operators for search and view specification within file systems. Quasar supports scope limiting by subtrees and by link distance. A Quasar name expands into a collection of Quasar names that represent a connected graph. We evaluate Quasar by contrasting its use with SQL and XPath in scenarios that are typical for very large file collections.

1. INTRODUCTION

The size of today's largest file systems have grown to many petabytes containing billions of files. The management of such collections forces us to re-examine the effectiveness of naming individual and groups of files. Current file systems are using hierarchical name spaces, which have proven effective for small-scale file collections: the hierarchical structure offers simple establishment of name subspaces and its static nature allows for a number of important performance optimizations [20, 2]. As the number of files increase, the need for a more flexible mechanism to identify individual and groups of files becomes more crucial [3, 22].

Since file systems have essentially not changed their name space structure for the last 30 years, users have adopted user-space applications to manage large collections of files of a particular type, such as separate applications dedicated to image, movie, or audio file collections. In addition to hierarchical namespace structures, these applications take advantage of the semi-structured nature of file collections, in that they allow users to identify files by terms that are occurring in unstructured content. They also allow for attributes in structured content, such as header information from various document types. Each of these applications implements its own search engine and manages its own indices over files and relationships between files.

The separate management of type-specific file collections by user-space applications has a number of drawbacks: each application has to re-implement search engines and metadata management with perhaps less optimized performance, and the metadata cannot be easily shared among multiple applications or be used for more general system services that span file types, such as backup, provenance tracking, or workflow. More recently, systems designers have started to develop applications that are more tightly integrated with file systems and are aiming to include all file types and work across multiple file systems [4, 6].

It is difficult to scale search engines to petabyte-sized file collections if the scope of searches cannot be effectively focused on subsets of these file collections. Existing file system search engines support keyword-based queries that define a flat namespace with a global scope over all files, or attribute-based queries that reduce the scope to the names of the terms in a query, but require the maintenance of attribute name-specific indices over all files. We are not aware of file namespaces that integrate scope as offered by hierarchical namespaces with keyword- and attribute-based search.

We propose Quasar, a path-based *naming language* that is designed for very large file collections and integrates hierarchical and attribute-based naming. We call Quasar a naming language instead of a query language because it defines the namespace as opposed to being a separate language for searching. By combining scope operators with operators for attribute-based search, we define a namespace that allows searching within a specified scope as well as attribute-based naming. We evaluate our design by contrasting Quasar expressions of important retrieval scenarios with expressions in common query languages: SQL, the standard query language for relational databases, and XPath [38], the query language preferred for hierarchically organized XML docu-

ments.

We make the following contributions:

- Proposal of a naming language (Quasar) that integrates scope and searching
- Evaluation of this naming language against XPath and SQL, two representative query languages for hierarchical and relational data, respectively.

The remainder of the paper is organized as follows. Section 2 extends the motivation for this work. Section 3 presents semantics and operators for our language (Quasar), section 4 evaluates the language by contrasting it to SQL and XPath in terms of scenarios important in very large file collections, and section 5 discusses the ramifications of our evaluation. Section 6 presents related work, and in Section 7, we conclude.

2. MOTIVATION

The design of Quasar is motivated by the current limitation of the POSIX file system interface, the emergent common features of applications that manage large file collections, the need for being able to limit the scope of searching, and the emergence of the successful path-based language XPath.

2.1 POSIX Namespace

Almost all common file systems in use today adhere at least to some significant extent to POSIX or "Portable Operating System Interface", a set of standards specified by IEEE in 1988 for any operating system [16]. In 2001 the standard was revised to not only include the system calls but also shell commands and utility interfaces [17]. For file systems the standard defines a hierarchical namespace, in which all non-terminal nodes are directories and all terminal nodes are files. This namespace is queried and manipulated with a set of system calls, shell commands, and a simple naming language. This naming language specifies a file or a directory by a path, either starting from the root or relative to the "current working directory" of the namespace. The naming language contains operators to refer to the current working directory and to the parent directory, which allows the specification of paths from any directory in the namespace to any other directory or file.

The POSIX naming language provides no support for querying. On the file system API level, the only query function is the listing of directory entries: the command `opendir` opens the directory for reading, and `readdir` returns one entry per call. All pattern matching functionality provided by shell commands require the listing of the entire contents of a directory. In other words, the naming language only allows the naming of files or directories where the location in the hierarchical namespace is already known. To find files or directories, users have to rely on shell commands or utilities like `find`, which recursively retrieve directory contents and try to match each file or directory to specified search criteria. These methods do not rely on any kind of indexing and are only viable when the search can be limited to a small subtree of the overall hierarchy.

Due to this limited querying functionality, the POSIX file system interface does not scale to very large file systems. Its design emerged from work in the 1970s [36] in the context of file collections with sizes multiple orders of magnitude smaller than today's collections. Now that file systems scale up to billions of files, the process of searching for a file under POSIX involves the listing of very large directories and the traversal of very deep hierarchies.

2.2 Large File Collections

Organizing large amounts of information is an open challenge and an active area of research in the context of personal information management, as well as information management on the institutional or enterprise level. At this point, there is wide agreement that the various requirements of storing and retrieving information does not map sufficiently well onto hierarchical namespaces. Lansdale identified two important reasons for this [18]. First, both deciding which categorization to use, and recalling or guessing the label of that categorization is hard due to likely ambiguities in label meaning and information generally falling into several categories [11]. Second, users remember far more about documents than can be used in strictly hierarchical retrieval procedures.

As file collections grow larger, users are forced to rely on increasingly sophisticated applications, which convert a file system namespace to a namespace tailored to a particular domain. For example, the number of files per directory can reach millions to billions [15]. Some applications therefore reduce the number of names in directories by compressing subhierarchies into single files [13]. Applications also enhance manipulation and retrieval by their own metadata management, including associations of domain-specific attributes and relationships to files, and implementation of search engines that search and index this metadata. Searching frequently extends over domain-specific file header information and unstructured file content.

Based on an informal survey of a number of applications managing large file collections, we identified three common features: *search*, *attributes*, and *relationships* [3]. Combined with a hierarchical namespace these features imply the following fundamental retrieval scenarios: matching on file attributes and relations between files, traversing file paths in both directions, setting the scope for further retrieval operations, and defining views on retrieved data. In section 4 we will evaluate our proposed naming language using these scenarios.

Perhaps the most important drawback of these applications is that their metadata and search engines are not available to other applications or the file system itself for general tasks such as backup, archiving, or workflow among multiple applications. Another drawback is that these applications duplicate implementation and maintenance efforts of common metadata management and search functionality instead of relying on shared file system service that perhaps would focus efforts more on optimizations than on re-invention. A third drawback is that applications cannot always tell file system changes that would require updating their indices. Popular file systems support notification mechanisms that either allows applications to watch a particular directories [23, 21], or the entire file system but limited to directory granularity [33]. It is not clear how scalable these notification mechanisms are.

Many applications use relational databases with application-specific schemas to manage their metadata because of the convenient fact that a well-established query language (SQL) already exists, and the assumption that a relational database

management system is a "one size fits all" database management system. As shown in [8, 35, 34] specialized DBMS can outperform RDBMS by at least one order of magnitude in text processing, scientific intelligence applications, and other cases that are relevant for searching semi-structured data. Also, the relational namespace usually serves as an alternative to the hierarchical namespace of the file system, as opposed to being integrated into one namespace: even though a search into the relational database might only be targeted to a small fragment of the file system namespace, the relational query processor has to always start with all database entries.

2.3 Integrating Scope & Search

We are aiming for a naming language that integrates attribute-based search with scope as provided by a hierarchical name space in form of directories. Such a naming language can identify files by their attributes and relationships to other files and directories. This includes matching directories based on attributes of files and directories they contain. The use of attributes on related files allows us to name virtual directories such as: "web pages that link to bar-chart images" or "directories owned by sasha that contain pdf files". Names that include attributes or relationships to other files always refer to a virtual directory because attributes or relations are not guaranteed to be unique.

The naming language can limit the scope of a name expression in three ways: (1) a single directory without its subdirectories, (2) a directory with its transitive closure of subdirectories (a subtree), and (3) a subtree up to a specified depth. Limiting scope to a subtree matches file system access and query locality: recent investigation in file system indexing [19] shows that attribute-based searching within subtrees can be implemented efficiently by constructing indices that exploit the locality inherent to hierarchical structures. Hence, limiting scope to subtrees is not only useful but can also be implemented efficiently. Limiting scope to a subtree up to a specified depth is useful when considering arbitrary relationships between files, where the namespace structure is properly characterized as a random graph and where the notion of a subtree is not well-defined. In this case, the notion of neighborhood within a specified link distance is more meaningful.

2.4 XPath

XPath is a widely adopted path based language for use with the hierarchical namespaces of XML documents [38]. Unlike other query languages, it functions more as a naming language for nodes. It does not have other query language features, such as data manipulations or relational-style joins. XPath results are abstract lists of nodes that match the naming string *i.e.,*the XPath query. Unlike POSIX, these names may contain attributes to match or filter, and results may be from disparate parts of the hierarchy. XPath features paths for descending through XML node hierarchies, which are similar to file system hierarchies. Additionally, XPath can treat all nodes as a flat namespace through the "//" operator.

XPath lacks a number of important features for managing large file collections. It does not have fine-grained scope control: only entire subtrees under a particular node. It does not have syntax to handle linkages between nodes, such as XLink [37], which utilizes attributes to describe the linkages that overlay on top of the XML hierarchy. Moreover, the XML data model only places attributes on the nodes themselves and assumes a single hierarchy. Finally, XPath cannot determine how the abstract results, *i.e.,*any nodes that match an expression, appear in listings to the user.

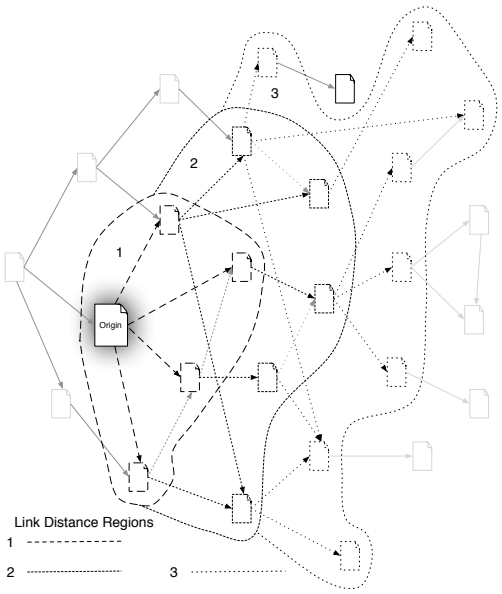

Figure 1: A graph of files, showing link distances from an origin.

3. Quasar

We have developed the naming language Quasar for identification of files and groups of files in very large file collections. Quasar expressions are designed to replace POSIX paths in file system calls and used as names to manipulate the metadata for single files (inodes) or collections of files (directories).

We base the language syntax around XPath. XPath syntax resembles file system paths and already integrates expressions for attribute-based search. The syntax provides a convenient backward compatibility between Quasar names and POSIX paths. POSIX paths are valid within Quasar, provided that the language is configured to do so through a small number of simple systems call-based settings. Quasar, like XPath, supports direct search and navigation of hierarchical structures.

A major feature of Quasar is that it names files by attributes or by relationships between files, e.g. expressions that specify files based on attributes of parent or child files. A second key feature of Quasar is the ability to express scope by directory, subtree, and subtree up to a specified depth. Finally, Quasar provides operators which help structure very large virtual directories (see 4.6).

3.1 Data Model

Since we want to include arbitrary relationship between files ("links") we must characterize our namespace as random directed graph, where the graph's nodes represent files and directories. Links fit emerging metadata structures such

as provenance, temporal locality, and established inter-file references, such as hyperlinks, include directives, and bibliographic citations. We do not assume any structure within the content of files. Data models that combine richly structured metadata and unstructured data are referred to as semi-structured.

We attach attributes, consisting of key-value pairs, to each node of the graph. We assume no limits on the number of attributes per node, including multiple values for any single key, as some categories for document tagging (as represented by keys) may deserve multiple labeling. The only required attributes are the fields found within standard file system metadata (similar to the inversion file system [26]) and a unique identifier which is assigned by the system and immutable.

As already mentioned, links are designed for arbitrary relationships between files. There has been compelling work regarding the value of maintaining provenance relationships among files [24] within storage systems. Links are intended for storing such relationships. However, links also represent the containment relationship between file and directory nodes. Containment is therefore just a special application of links. The data model allows for unlimited attributes on links, and the only "required" attribute is a system-generated unique link identifier akin to a file's inode number.

3.2 Semantics

Our language maintains the use of the traditional "/" as the "namespace" separator for its pathnames. Each unit of the namespace, the characters between each "/", consists of a series of query operations or presented terms. We add units onto pathnames to change the current result set (one or more files), through refinement, traversal operations, or both. The "/" may represent a default language operator if none is present. For virtual hierarchies (see section 4.6) we may navigate using values as our pathname unit. Likewise, "Going up a level" or ".." means removing the rightmost unit from the query context. This feature might be described as a lexicographic approach to the ".." directory entry, as many shell environments in modern OS employ this approach today. However past experience in UNIX with hard and symbolic links did not always "get dot-dot right"; experience with Plan 9 recognized and employed a lexicographic solution [28].

Under our system model, the only attribute key that the system guarantees to be unique for each file is the file's inode number. Applications may recall the inode number as part of custom directory listings or stat calls. Custom metadata transducers may provide other "unique" keys, but those modules must provide the mechanism to enforce uniqueness, as a file system utilizing Quasar makes no other enforcement of unique values, with the exception of configurable POSIX name enforcement. When such enforcement is not activated, a directory may contain (link to) two or more files that have the same value for the POSIX name field (the conventional "name" found in a POSIX directory entry), but the pair of files sharing the common value for "name" may differ by other attributes (modification time, size, version number, etc.) Of course, this means that a Quasar pathname can result in zero or more files, where zero files results in "file not found", one file is a single file, and multiple files is a virtual directory.

Hence, the reality of pathnames mapping to multiple results implies that conflicts may arise when a user presents a name for a file and retrieves a number of results. There are a number of ways to resolve such conflicts. First, items within virtual directories can be addressed by an entry number, as is possible when using XPath. When each item is opened, applications may use `fstat()` or `fgetxattr()` using the open file's handle to look up properties or add a new attribute to the item. These operations work within virtual directories, as well. Additionally for many pathnames, it is possible to select the most recent matching file as an alternate way to resolve conflicts.

A concern that may arise from a naming scheme that utilizes attributes is that the names lack the same persistence for sets of results as is the case with regular directory entries. Such "results" in traditional file systems are only persistent when the files are not being relocated or renamed. Attribute-based searches could be more dynamic because new files may consistently match existing names, while it is much less likely to find files in directories changing so frequently. How dynamic results appear depends on the Quasar name, because some terms are more likely to match a wide variety of results, especially if metadata on some particular files changes frequently or from a constant stream of new files. If a user would like to make their dynamic results appear persistent, the results can be copied to a static directory or tagged with a particular identifier.

In the future, when we adequately address typing options for extended attributes, Quasar will fully support range queries over extended attributes. Regular attributes are numeric data-typed already. However, dates must be converted to UNIX time in order to be compared against the standard POSIX time stamp attributes.

3.3 Language Configuration

In order to allow the Quasar language interpreter to accept conventional POSIX path strings, we must be able to configure the interpreter to do so. An added benefit of such configuration is that we may present more brief language syntax in pathnames. For a solution, we have devised some configurable default settings:

- Global search term(s) to be applied to all traversals
- Default search operator, when none is specified following a `/`. The operation could be a term-based refinement or single traversal, matching all following terms
- A search field (attribute key) to apply to both traversal or refinement operators.
- Constraint of returned items to single files, file not found otherwise (see section 4.7).

The use of defaults allows Quasar to support POSIX compatible paths over multiple namespaces. Such configuration is possible at a per-process granularity, as is shown by the Plan 9 system [29, 30]. Of course, having multiple configurations means that paths in one namespace will not produce the same results in another. The same would happen if users or administrators set up local environments with separate defaults. To deal with changing defaults, it should be trivial to provide a utility that can translate shortened paths, which provide results under a certain set of defaults, to longer paths that utilize more Quasar complex syntax and operators hidden by the default configurations. Then,

the fully expanded paths could be utilized within any environment. Defaults are to be set and retrieved by a series of systems calls, much like one may presently set or get the current working directory for a given process. Hence, the utility to expand a path need only to parse the shortened path and expand based on retrieving the current default settings.

4. EXAMPLES

We present and evaluate the Quasar language in fundamental retrieval scenarios by by comparing its expressions to SQL and XPath. The queries are examples of scenarios that are typical for very large file collections We also include an overview of miscellaneous Quasar operators, but do not present examples or evaluation of those.

4.1 Compared Languages

We present the schemata we use for each of the compared languages. In our comparison with XPath, we must ensure that XML properly represents our file system metadata, in particular non-hierarchical links. Non-hierarchical links between nodes within XML documents can be represented by attributes using the XLink standard but XPath does not support traversing such links. Therefore we represent both files and links as XML elements. Files that contain other files must have link elements that each contain a single file node. We will use the element name to signify each and use the element attributes to store the attributes for both files and links (using F and L characters respectively in our examples). Hence, the XML DTD would need to have the following (recursive) definitions:

- The root node must be an "F" element.
- "F" element may contain any number of "L" element (including none).
- "L" element must have a single "F" child.
- "F" elements require traditional file system attributes as XML attributes. They may have unlimited additional attributes as extended attributes. "L" elements have no attribute requirements, but may also have any number of XML attributes to represent attributes on the links.

In contrasting Quasar with SQL, we need to represent the file system metadata model as a set of tables:

```
files table (inodes):
inode # | mode | size | owner | group | ctime |
  mtime | atime

links table (links):
link id | source | destination

link attributes table (link_attr):
link id | key | value

file x-attributes table (file_attr):
inode # | key | value
```

Note that there are a variety of possible schemas for file systems metadata representation. A more simple version does not contain links and stores full paths in a single table for all inodes. This enables more efficient lookups for individual paths, provided that the database indexes that particular column. Some file system indexing tools, such as Spotlight [4] and Beagle [6] follow this approach in their use of sqlite as a backing DB store. However, we must maintain a separate table for links between files in order to attach multiple attributes to links (see end of section 3.1. Another problem with storing full paths for files is that it complicates moving entire directory trees, since all full-path values for files within the tree need to be renamed, in contrast to simply changing a single row for the appropriate link.

4.2 Attribute-based Matching

By default and unless directed otherwise by a leading operator a name always starts at the current working directory. The `@` character operator prefixes an attribute key/value pair.

Here, we present the Quasar syntax for a simple example query: "`@Author=Ames`". The "`@`" opens the query space to all files. The "`=`" separates the key-value pair components. To refine the query, we add a second term to make it appear as "`@Author=Ames;Year=2008`" The `;` separates the two terms. An alternate way to write the query would be "`/Author=Ames/Year=2008`" if we set "matching" as our default operator. The approach taken here follows the original semantic file system approach to paths [12].

The translation of this query into XPath is trivial: `//F[@Author="Ames"]`. Note that we use "F" to denote that we wish to match an attribute attached to the file as opposed to the link.

SQL queries can match basic file metadata through simple "AND xxx=123" predicates added to the WHERE clause. For matching extended attributes on files, we require an inner join with the file_attr table.

```
SELECT ino FROM inodes WHERE inodes.ino =
file_attr.ino AND file_attr.key = 'Author' AND
file_attr = 'Ames'
```

The `!filter:` operator may be used for refinement and also to begin naming strings. Specified terms carry over for each subsequent traversal. For instance, `!filter:username=sasha` will constrain all to only files pertaining to that user ID, regardless of which directory might be specified following the expression.

4.3 Link Traversal

Our link traversal operator resembles the operation for path resolution in today's file systems, which traverses hard links from directory entries. We traverse links from actual directory entries, links conveying inter-file relationships, or virtual links. We expect that many files will still be placed in directory hierarchies and may not have searchable extended attributes. The traversal operation is necessary for browsing of directory contents. Given the locality exhibited in hierarchies, it should be more convenient to browse for nearby relevant files, rather than rely on an attribute-based search [5]. Virtual links are the entries within virtual directories (collections of query results) and following those either arrive at a file, or another virtual subdirectory containing subsequent virtual links.

In the following examples: we denote"linktype" of "containment" for the links that place files within directories and "linkname" is the traditional POSIX directory entry.

XPath:

```
/F/L[@linktype='containment' AND @linkname='usr']
/F/L[@linktype='containment' AND @linkname='bin']
/F/L[@^linktype='containment']/F[NOT @filetype=executable]
```

SQL:

```
SELECT i.ino FROM inodes i,  links l1, links l2, links l3,
link_attr la1_1, link_attr la1_2, link_attr la2_1,
link_attr la2_2, link_attr la3_1, file_attr fa3_1
WHERE
l1.source = 0 AND l1.target = l2.source AND
l2.target = l3.source AND l3.target =  i.ino AND
l1.id = la1_1.id AND la1_1.key = 'linktype' AND
la1_1.value = 'containment' AND l1.id = la1_2.id AND
la1_2.key = 'linkname' AND la1_2.value = 'usr' AND
l2.id = la2_1.id AND la2_1.key = 'linktype' AND
la1_1.value = 'containment' AND l2.id = la2_2.id AND
la2_2.key = 'linkname' AND la1_2.value = 'bin' AND
l3.id = la3_1.id AND la3_1.key = 'linktype' AND
la3_1.value = 'containment' AND (NOT (fa3_1.ino = l3.target
AND fa3_1.key = 'filetype' AND fa3_1.value = 'executable'))
```

Example 1: Queries to traverse multiple links

```
$^linktype=containment;^linkname=usr
$^linktype=containment;^linkname=bin
$^linktype=containment;-filetype=executable
```

Such a query appears way too verbose. Fortunately, using the language configuration features of Quasar, we can set: `^linktype=containment` for a global match on all links; a default field to match of `^linkname`; and default operation of traversal. (^ indicates attribute on a link) With these settings, we may instead present `/usr/bin/-filetype=executable` as our query string. While the string resembles a POSIX path, it still uses the minus and equal signs to provide enhanced search functionality (as found in some search engine syntax, `-` means exclude from results).

As shown in Example 1, for XPath, we initially match the root inode without any qualification. Subsequently, we match each L and file node with the pertinent attributes to match for each step along the way. In SQL, because we traverse three links, we require two self-joins over the *links* relation. Additionally, we join the *file_attr* and *link_attr* relations to account for the attributes in the queries. To compose queries programmatically, we label the reference variables respectively for each traversal and attribute requested in the query.

4.4 Matching Related Files

These operators are related to traversal, but function differently than "walking a path". The goal of their use is to narrow down specific results based on attributes on files explicitly linked to those in the current result set. It is important to retain these operators, as otherwise, we would only have intrinsic attributes and content available as the basis for searches. Under our data model, we have two categories: parent/ancestor matching or child/descendent matching. Consider the following request: find me a PowerPoint file that includes a .png bar graph and a jpg photo authored by "Ames". Such a query requires we locate the children matching the criteria for both those types of images. Figure 4.4 shows appropriate syntax in each language.

Similar queries, in which we want to match attributes on **two parents**, cannot be performed in XPath within a single query (provided we are using some construct within the scope of XML that can provide additional linkages between nodes). Under **Quasar,** matching the first parent is accomplished through a traversal from that parent. For the second, we use the `!matchparent:` operator, followed by the terms we wish to match. SQL queries of this class take a very similar form as they do for matching children, except that the column references to join the *links* table with the *inodes* table are switched from source to target, and the joins of*links* with *file_attr* are switched from target to source. Hence:

Quasar:

```
@filetype=PowerPoint!matchchild:^linkttype=include;+filetype=png;
+imagetype=bar_graph!matchchild:^linktype=include;+filetype=jpg;
+Author=Ames
```

XPath:

```
//F[@filetype="PowerPoint"][L[@linktype="include"]
/F[@imagetype="bar_graph" AND @filetype="png"]]
[L[@linktype="include"]/F[@filetype="jpg" AND @Author="Ames"]]
```

SQL:

```
SELECT i.ino FROM inodes i, links l1, links l2, file_attr fa0_1,
file_attr fa1_1, file_attr fa1_2, file_attr fa2_1,
file_attr fa2_2, link_attr la1, link_attr la2
WHERE
la1.id = l1.id AND la2.id = l2.id AND
la1.key = 'linktype' AND la2.key = 'linktype' AND
la1.value = 'include' AND la2.value = 'include' AND
l1.source = i.ino AND l2.source = i.ino AND
l1.target = fa1_1.id AND l1.target = fa1_2.id AND
fa0_1.id = i.ino AND fa0_1.key = 'filetype' AND
fa0_1.value = 'PowerPoint' AND
fa1_1.key = 'filetype' AND fa1_1.value = 'png' AND
fa1_2.key = 'imagetype' AND fa1_2.value = 'bar_graph' AND
l2.target = fa2_1.id AND l2.target = fa2_2.id AND
fa2_1.key = 'filetype' AND fa2_1.value = 'jpg' AND
fa2_2.key = 'Author' AND fa2_2.value = 'Ames'
```

Example 2: Syntax for finding files using related files. With SQL we join two child to the primary inode table.

```
l1.source = i.ino AND l2.source = i.ino AND
l1.target = fa1_1 AND l1.target = fa1_2 AND
```

becomes

```
l1.target = i.ino AND l2.target = i.ino AND
l1.source = fa1_1 AND l1.source = fa1_2 AND
```

4.5 Scope

The first type of scope controlled naming that we present is based on use of a link distance parameter. Through this operator we hope to down large search spaces where attribute-based search alone may not be so reliable. In Quasar, a link distance-based search uses the following form (N is a positive integer specifying the link distance parameter and xx-criteria are key=value pairs of attributes to match):

```
@<origin-criteria>$[N]<link-criteria>;
<target-criteria>
```

The Quasar form must be translated into other query languages, as we show in example X, employing UNION operators to combine results present at various link distances within the scope of the query. The following shows the

XPath:

```
//F[origin-criteria]/L[link-criteria]/F[target-criteria]
| //F[origin-criteria]/L[link-criteria]/F/L[link-criteria]
/F[target-criteria] | ...
```

SQL:

```
SELECT i1.ino FROM inodes i1, file_attr origin1_[1..n],
links l1_1, links_attr la1_[1..n], file_attr target1_[1..n]
WHERE
links1_1.target = i1.ino AND
links1_1.id = links_attr1_[1..n].id AND
links_attr1_[1..n].[key|value] = <link-criteria> AND
origin1_[1..n].ino = links1_1.source AND
origin1_[1..n].[key|value] = <origin-criteria> AND
target1_[1..n].id = links1_1.target AND
target1_[1..n].[key|value] = <target-criteria>
UNION
SELECT i2.ino from inodes i2, file_attr origin2_[1..n],
links l2_1, links l2_2, links_attr la2_[1..n],
file_attr target2_[1..n]
WHERE
links2_2.target = i1.ino AND
links2_2.source = links2_1.target
links2_1.id = links_attr1_[1..n].id AND
links2_2.id = links_attr1_[1..n].id AND
links_attr2_[1..n].[key|value] = <link-criteria> AND
origin2_[1..n].ino = links2_1.source AND
origin2_[1..n].[key|value] = <origin-criteria> AND
target2_[1..n].id = links2_2.target AND
target2_[1..n].[key|value] = <target-criteria>
UNION
...
```

Example 3: Syntax for queries with link distance-based scoping

equivalent in XPath, where no N-levels-deep operator exists. It is possible to traverse N levels deep to nodes using multiple "/*" calls, and combine each with the | operator (union). For SQL, we abbreviate some of the expressions where multiple criteria may exist. Moreover, the origin may be a known file, where an inode number could be set in the query. SQL has the similar problem as XPath, requiring UNION operators to combine multiple distances.

The second type of scope considered, subtrees, can utilize any type of link within the file collection in addition to directories, such as a "provenance" subtree, in which we may consider all files that have a traceable origin. Nonetheless, it is important to note that there are limitations to solely relying on finding files within directory subtrees, as what were once manageable subtrees eventually contain too many files to be useful. As we utilize storage, files are continuously added to the hierarchy over time, and the subtrees also grow when new subdirectories are added, as well.

To handle naming within entire subtrees within a namespace, Quasar provides a single operator to select the current subtree and filter based on a global *!filter:* or on any subsequent attributes or keywords to match. The syntax style is almost the same as for link distance scoping queries. The same operator as used for traversals takes "-1" as a parameter to specify an entire subtree, *i.e.,*`$[-1]`, or we may use the `!subtree:` operator in its place, which should appear more clearly.

For subtrees, XPath queries that utilize the "descendant::" axis or "//" in XPath 2.0 can successfully search indefinitely through a subtree. However, this operation is not parameterized as is with Quasar, so it is not possible to limit depth. We should assume under XML documents that such searches are safe under XPath.

On the other hand, SQL cannot handle this form of search in a single query. In order to perform such a search, a procedure is required to issue SQL queries, which must repeatedly check the stop condition if it has reached the bottom of the tree, retrieve any matching results for the current level, and merge results from all levels.

4.6 Results Presentation

The presentation operators determine whether results should be listed as tuples within directory entries, in virtual hierarchical views, or a combination of both. For virtual views, our functionality performs *group by* or *roll up* on requested fields (each a dimension), and makes the space browseable using navigation, as opposed to issuing repeated declarative queries.

Directory listings under Quasar may also present attributes as tuples: simple lists containing multiple values. For each file, this functionality presents more meaningful naming information than a single file name. For instance, consider the images cataloged with the cryptic file names generated by the camera. When images are amassed from cameras that use the same naming scheme, there are bound to be collisions. Using other metadata items should be sufficient to differentiate between the different files. Consider listing by "Author" (or photographer), location (tagged manually or by spacial coordinate match, day/time, and sequence number (either in the query or for the attributes). Of course, using Quasar we can roll up these fields into virtual directories for browsing. iPhoto has some of this capability already, but it presents a fixed interface and the images it maintains are limited in their portability.

We revisit the example above for both presentation modes. If we want to roll up each value as a virtual hierarchy, we use the syntax: `&Author&Location&Date&Sequence`
Alternatively, if we wish to show all values within each tuple for the listing name, we use the syntax:
`&Author;Location;Date;Sequence`
The `&` character represents the "ListBy" operator. In conjunction, `;` denotes that additional fields should be used for each tuple. Additionally, we could also have shorter tuples with two levels of roll up, instead of four, using:
`&Author;Location&Date;Sequence`

As XPath is a selection language, it does not handle presentation of results, *i.e.,*. values from which fields to return. While XQuery does, its syntax is procedural, which is too heavy-handed to be adopted in a naming language. Instead, we prefer to directly specify only the field names to minimize required syntax.

SQL queries that select from only single table make the specification of values presentation for results simple, in that we expect each column to reflect the requested values. However, queries that join multiple tables, as are the ones we would require for file system metadata management are likely to cross other tables. In our case, the table name and column names are generic. The key name field specifies which property we would like to see.

One issue that we must handle is missing attribute values on files. Tagging is likely to be inconsistent, especially when end-users are responsible, but also system agents that tag files with attribute metadata might be missing their context field due to extenuating circumstances. When encoding .mp3 files for example, if an external database for tagging

cannot be contacted, the .mp3 files may be likely to be missing many of the fields that the software may read from that service. Users may enter the artist and album name, as that information pertains to several tracks, but do not wish to take the time to enter additional information.

Our experience with SQL showed that we cannot use our general file system metadata schema to properly outer join to account for missing attributes. Thus, we need to create a SQL `VIEW` on demand for new attributes when requested. A query writer interface would need to perform that function. First, it must check if the view exists. If not, create it. Then, issue a query utilizing the view(s). The query must use a number of outer joins (one less than the number of requested fields) so null values may be represented in the result set. To compose the query, the query contains the SQL `UNION` operation of N subqueries. Each subquery has $N-1$ `LEFT OUTER JOIN` operations to attach the subsequent field. The query writer must keep the columns consistent between the sub-queries, but rotate the views from each sub-query as to ensure complete coverage of all possible missing values. We learned through mistakes in code to generate the proper SQL statements programmatically that this procedure must be followed or some missing rows may not appear or we may end up with duplicates.

Presenting attributes on links adds some additional complexity to names presentation. The rule is that any final links traversed (at the end of a series of traversals) to target files that match the criteria may present their attribute values within the returned directory listings.

Under SQL, as with the file attributes, views must be created for each requested field that resides on links. These views must join the links table with the links attributes table, such that a user may query by inode number and be able to retrieve the right value. So SQL syntax would look like:

```
CREATE VIEW <name> AS
SELECT link_attr.value, links.target
WHERE link_attr.key = <field> AND
link_attr.link_id = links.id
```

The problem with these views is that the joins will be performed constantly if there is a frequently used field for viewing. A key example is in POSIX emulation, where the "name" field resides on the link to each file from the enclosing directory, as opposed to residing on the file itself. Thus, we should expect a view for that field to be referred to frequently.

4.7 Miscellaneous Operators

To add to the operators, we adopt a number of existing database operators, including "TOP N", "ORDER BY", and introduce a constraint within expressions to the count of the items returned from the current search (default of 1). More complex queries add flexibility to finding stuff. It is meaningless to perform a TOP N unless the results are properly ordered (by an attribute).

While it may be argued that sorting and retrieving the "TOP N" are better left to the front end, we choose to place within the query processing, because the results of these operations may factor into additional query operations, (specifically traversal, but others may be possible) For example, we might want to find images included by ten most recent presentations. The equivalent in SQL would be the so-called "nested" query, where the results become a set for the next queries. Secondly, we expect that ordering of results will be a commonly used feature and so, it should consistently be part of the language.

Finally, (count = N) constraints allows the system to mimic POSIX compliance for directories, since we require only a single match to traverse. As the language parser processes a name string from left to right, if the parser encounters the count constraint operator, the system must check the current count of file results based on the name string processed up to the point of the operator. If the result count is greater than the constraint, the system should return an error. The constraint default value is a single result, as utilized for the name attribute within directory entries in order to comply with POSIX.

5. DISCUSSION

The examples above show that SQL requires rather extensive syntax for query concepts that are preferable to keep simple. XPath is more comparable to our needs, yet it lacks some of the needed features and those it makes up cost it in some additional verbosity, *i.e.,*. handling attributes on links vs files.

As queries increase in their complexity, the number of joins will increase as well. The impact that the increase has on query performance depends much on the back-end implementation. Search engines and relational databases take different approaches to optimization for joins. Because search engines are optimized around document retrieval and build their indices accordingly, they may better optimize joins (intersections) of multiple postings lists. Relational databases—given a more general approach to treating structured data—do not have the same luxury, thus all the joins we present are computed at the same rate as any other, based on general optimizing principles. For instance, a query for all files with a particular owner and within 3 links of the "Presentations" directory is likely to be a large intersect, and proves to be inefficient for the relational database to compute. The database, in this case, will need to go through the processing of computing the set of all rows that match the "within 3 links" part of the query, through the long recursive query we present in section 4.5.

6. RELATED WORK

There has been a good amount of research focused on enhancing file systems through attribute-based paths or view definitions. The Semantic File System [12] and the Logic File System [27] utilized path-based interfaces for views of files, the latter incorporating a series of logical operators into path expressions. Other solutions had a separate systems interface to handle searching and views of files, namely the Property List DIRectory system [22], Nebula [7], and attrFS [39]. Some additional systems have attempted to retain hierarchies, either based on attributes attributes [32, 25] or mixing directories with content [14]. While some of these systems try to unite attribute-based naming with hierarchies, we contend that they were not successful because the interface that they chose was not sufficient in combining the use of the two approaches. Also, these approaches had not considered the use of different classes of scope.

Additional approaches focus on interfaces outside of the file system. Presto [10] allowed for the interaction with doc-

uments as sets based in attributes in a fluid, GUI environment. Spotlight [4] provides a separate interface for file system search that includes keyword search and smart folder (virtual directory) creation that involves multiple orthogonal search categories. Spotlight's search indices are not part of the file system, but are integrated through the operating system. Phlat is an experimental interface for personal information management that attempted to provide a smooth continuum between search and browsing [9]. These systems, in addition to the file systems above, utilize search to present conventional file names. Instead, Quasar shows files in lists based on choosing appropriate attributes. Furthermore, the above-mentioned search tools and file systems only consider based on attributes placed on files directly, not utilizing relationships.

The Linking File System [3] showed how relational links between files might be stored and maintained, given storage class memories for which to store metadata. For it, we proposed a simple search interface to search for files based on attributes on links, but in conjunction with traditional paths. This work is a precursor to our present language work with Quasar.

The OLAP-Object Query Language [31] combines paths with attribute based filtering or search terms. Unlike file system paths, these object-oriented paths follow relationships of classes of objects that refer to other classes.

Additionally, we consider how the semantic web standard might also be applied to describe file system metadata via RDF, ontology definition languages, and use of SPARQL to query. Such an approach has disadvantages as well. While it may tightly wrap up data into classes like object-oriented data models, and handle relationships via tuples that are akin to links, it appears to be generally isomorphic to the relational model [1]. This means traversal-based searches using SPARQL succumb to similar problems that we have encountered in observing SQL queries.

7. CONCLUSION

For very large file collections, the use of hierarchical or attribute-based namespaces alone are not adequate. We have discussed the use of a language that combines the advantages of both approaches. We introduce the use of scope and view definition as part of the naming language. Our contrast with other languages shows that use of specialized language for file collections is promising.

8. ACKNOWLEDGEMENTS

This work was supported in part by the Department of Energy under awards DE-FC02-06ER25768, DE-AC52-07NA27344 (through Lawrence Livermore National Laboratory), and by the industrial sponsors of the Storage Systems Research Center at the University of California, Santa Cruz. We thank the members of the SSRC for their feedback.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, Feb. 2007.

[3] 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, May 2006. IEEE.

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

[5] D. Barreau and B. A. Nardi. Finding and reminding: file organization from the desktop. *SIGCHI Bull.*, 27(3):39–43, 1995.

[6] Beagle Project. About beagle. http://beagle-project.org/About.

[7] 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.

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

[9] E. Cutrell, D. Robbins, S. Dumais, and R. Sarin. Fast, flexible filtering with phlat. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 261–270, New York, NY, USA, 2006. ACM.

[10] P. Dourish, W. K. Edwards, A. LaMarca, and M. Salisbury. Presto: An experimental architecture for fluid interactive document spaces. *ACM Transactions on Computer-Human Interaction*, 6(2), 1999.

[11] S. T. Dumais and T. K. Landauer. Using examples to describe categories. In *Proceedings of the 1983 Conference on Human Factors in Computing Systems (CHI '83)*, Boston, MA, 1983.

[12] 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)*, pages 16–25. ACM, Oct. 1991.

[13] M. Gokhale. Personal communication, September 2007.

[14] 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)*, pages 265–278, Feb. 1999.

[15] G. Grider. Personal communication, May 2008.

[16] IEEE. *1003.1-1988 INT/1992 Edition, IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988)*. IEEE, New York, NY, USA, 1988.

[17] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, New York, NY, USA, 2001.

[18] M. W. Lansdale. The psychology of personal information management. *Applied Ergonomics*, 19(1):55–66, 1988.

[19] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L.

Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. Technical Report UCSC-SSRC-08-01, University of California, Santa Cruz, May 2008.

[20] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[21] R. Love. Kernel korner - intro to inotify. *Liunx Journal*, September 2005.

[22] J. C. Mogul. Representing information about files. Technical Report 86-1103, Stamford Univ. Deptartment of CS, Mar 1986. Ph.D. Thesis.

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

[24] 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*, pages 43–56, 2006.

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

[26] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, California, USA, Jan. 1993.

[27] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[28] R. Pike. Lexical file names in plan 9 or getting dot-dot right. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[29] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. *Operating Systems Review*, 27(2):72–76, Apr. 1993.

[30] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, and H. Trickey. Phil winterbottom: Plan 9 from bell labs. *Computing Systems*, 8(2):221–254, 1995.

[31] E. Pourabbas and A. Shoshani. The composite olap-object data model: Removing an unnecessary barrier. In *SSDBM06: 18th International Conference on Scientific and Statistical Database Management*, pages 291–300, 2006.

[32] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *12th International Conference on Distributed Computing Systems*, pages 572–580, June 1992.

[33] J. Siracusa. Mac os x 10.5 leopard: the ars technica review. http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/7, October 2007.

[34] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all?—part 2: Benchmarking results. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2007.

[35] M. Stonebraker and U. Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[36] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[37] W3C. Xml linking language (xlink) version 1.0. http://www.w3.org/TR/xlink/, June 2001.

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

[39] 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*, pages 359–365, March 1995.