Text Search in an NFS-Proxy: A Case Study in Extensible File Systems

Kristen LeFevre Kevin Roundy

CS 736 Course Project
May 10, 2005

Abstract

This paper describes the design of an extensible 3-tiered semantic file system, backed by an existing extensible object-relational database. The system is designed to export the standard NFS interface, while providing indexing and query support for user-defined file types using the virtual directory abstraction.

To illustrate the feasibility of the proposed architecture, we describe its implementation for one important file type, text. Indexing and query support for text are implemented in the database using a plug-in module, and support for full-text queries, including boolean keyword search and information retrieval rank, are exported by the file system interface using virtual directories.

1 Introduction

Every day millions of people use computers at work and at home, creating millions of documents. Unfortunately, as the quantity of stored documents increases, it is often difficult to manage and organize them in an understandable way. Indeed, a directory or file name that once made sense (e.g., "Summer_Pictures") may lose its meaning with time, making it difficult to locate individual files.

One recent user-study found that the majority of individuals surveyed (who were all experienced computer users) were dissatisfied with the organization of their personal documents and e-mail, and many indicated that they did not have time to organize their documents to their satisfaction [5].

This paper describes the design and implementation of a next-generation extensible file system that provides an intuitive interface for associative file access. In designing our prototype system, we had three main goals:

1. **Powerful Query Processing** The system should provide as much functionality as possible for executing queries, using both the meta-data associated with files, as well as the contents of the files themselves. Queryable attributes should be indexable, and query execution should be reasonably efficient.
2. **Ease of Extensibility** Because personal file management is an evolving problem, and the prevalent file types are likely to change over time, it should be relatively simple to extend the system to handle new types of files, as well as indexing and query processing for these new types.
3. **Preserve Existing Client-Server Interface** Existing distributed file system interfaces, particularly NFS, have become ubiquitous, and countless user-level applications have been built to rely on them. These interfaces are also attractive because of their simplicity. For these reasons, it is important not to modify these interfaces more than is absolutely necessary as we seek to achieve the first two goals.

The first main contribution of this paper is a simple file system architecture designed to address each of our stated goals. The proposed architecture has three tiers, and is backed by an object-relational database [3], which provides a simple uniform interface for defining and implementing new file types, as well as associated functions, indices, and access methods. Thus, we are able to incorporate new types of files in a uniform way, rather than writing ad-hoc functionality for each new type.

The proposed architecture uses a proxy layer to export a traditional NFS interface, and queries are issued using the *virtual directory* abstraction [6], which extends the naming semantics of the existing tree-structured file system, while otherwise preserving the existing interface.

In addition to our three primary goals, the proposed architecture has some other desirable implications, including support for atomic transactions and recoverability. One key direction in database research for many years has been in maintaining strong transactional consistency semantics, even in the presence of

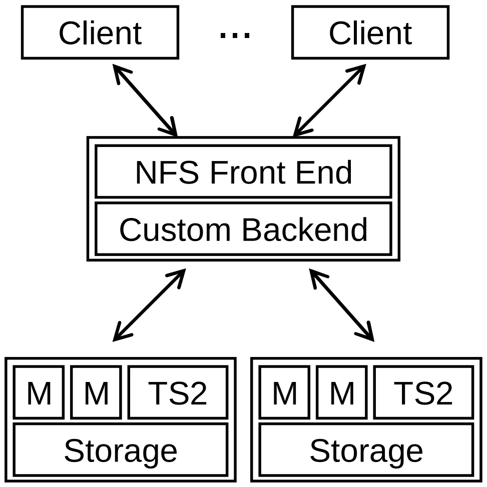

Figure 1: Overview of system architecture

system crashes. Traditional file systems, in contrast, provide only rudimentary guarantees about consistency. Incorporating these techniques into a general-purpose file system has several added benefits, such as the ability to export consistent snapshots of files, and the ability to recover file system data to a consistent state following a crash.

To illustrate the feasibility of our proposed architecture, we focus on one important type of file: text. The second main contribution of this paper is the implementation of full-text query functionality inside our proposed NFS proxy. Specifically, we implement two important types of text queries, *boolean keyword search* and *information retrieval rank*, using the virtual directory abstraction, inside a fully-functional NFS server proxy.

1.1 Paper Overview

In Section 2 we provide an overview of our database-backed NFS-proxy architecture, as well as an overview of the architecture of the underlying object-relational database. Our proposed architecture allows the file system to be extended relatively easily to manage new types of files. As an example of this extensibility, we consider the problem of managing and searching text documents based on content. In Section 3 we detail how text search and rank functionality is exposed through the NFS interface, using virtual directories. The details of our prototype implementation are explained in Section 4.

An overview of related work is given in Section 5. Finally, we describe our conclusions and future work in Sections 6 and 7.

2 System Architecture

The bird's-eye view of our architecture is that we have replaced the back end of an NFS server with an object-relational database, in this case Postgres [3]. As seen in Figure 1, we have a 3-tier architecture with an NFS client layer, a proxy layer, and a database layer. Our implementation allows us to use unmodified NFS clients, and while we add modules to Postgres, we do not make any changes to the database itself.

We are not the first to realize the benefits inherent in using a database as a filesystem component. Olson's Inversion File System [9] creates a file system on top of the Postgres database, using Postgres to allow for the use of advanced functionality through user-defined types. Halverson and Samios also used Postgres in their file system [7], but strived to make its presence completely transparent by utilizing it as the back end of an NFS server and utilizing unmodified NFS clients. Our system architecture is closely tied to theirs, and in fact we used their source code as a starting point, but we utilize the capabilities of the database more fully by exploiting the database's query power in certain ways. The immediate use for the database's query power is for search, and our prototype implements search over text files.

2.1 Why NFS?

The use of standard NFS clients serves several purposes, the most important of which are portability and transparency. Portability has two facets, the first and most obvious being the ability overcome geographical distance between an NFS client and NFS server. The second notion of portability requires the existence of compatible NFS clients from various operating systems platforms. Happily, both of these portability goals are readily met by NFS, which is available on most OS platforms thanks to its many open-source implementations.

Transparency is also a key motivator for our use of NFS, as it provides existing code support and the familiar semantics of Unix. This frees our end users from the frustration of learning a new interface and allows them to focus on using the new functionality, which is also remarkably easy to use, and is explained further in section 3.

In keeping with Halverson's system we use a user-space implementation of NFS [8]. This kept debugging simple and allowed us to leverage the existing proxy code written by Halverson, which is closely intertwined with the User-Space NFS source code. There are of course other file systems that work over a network and that offer similar functionality to NFS (such as AFS) but the NFS protocol is particularly nice to work with because of the statelessness of the NFS server. This keeps the requirements of the server side very simple and made it very easy to replace the existing NFS server with Postgres while staying true to the NFS protocol.

2.2 The Database Back End

There are several reasons why the use of a database is powerful and desirable as a filesystem component. Both [7] and [9] realize the benefits of solid data semantics, transactions, and quick crash recovery. In addition to this, our prototype incorporates search capabilities for text files by using the query power of the underlying database. This is all done without any modifications to the NFS client.

The essential properties of an NFS server are met in our architecture. The NFS protocol demands that writes can only be cached on the client side and that system calls must be idempotent. These two requirements were easy to satisfy in our system thanks to the machinery provided by Postgres. All NFS calls that perform updates wrap their filesystem modifications in database transactions and do not inform the client of success until the transaction-end log entry has reached stable storage. For example, if the server crashes while creating a file it will roll back the partially completed transaction. This allows the server to re-start the transaction when the client re-sends the request without any ill consequences.

The use of an object-relational database is essential to our architecture. Object-relational databases such as Postgres are extensible in that they allow a user to create and plug in new objects (modules) that consist of types and associated functions and indexes. Many such modules are readily available for Postgres and they serve a wide range of purposes, the most popular of which are packaged with the Postgres distribution. Postgres makes it easy to create new modules, and our architecture allows us to leverage them by customizing the NFS proxy. In [7], the use of an object-relational database was not fully justified, as a simpler performance-tuned relational database would have met their needs, but for our implementation the extensible nature of such a database is critical as it is the means through which we provide search functionality and other features. Our prototype uses the tsearch2 module [4] which enables text search through the types, indexes, and functions that we discuss in section 4.1.

2.3 The Proxy Layer

The importance of the proxy layer is that it allows us to selectively expose the database's functionality and grants us significant flexibility. As seen in Figure 1, the proxy translate NFS system calls into SQL statements that query the database and responds with messages formatted according to the NFS protocol. We avoid having to change the NFS client implementation by funneling all requests through this layer. We encode requests for the additional functionality that we provide by using using special syntax in conjunction with the mkdir command and storing the results in the resulting directory. The proxy layer is responsible for identifying such special requests, issuing the appropriate database queries, and creating and storing the results in the new directory. Section 3 details the syntax and semantics of these commands.

This proxy approach allows us to expose database functionality in addition to text search. For example, we could address concerns with NFS' lack of concurrency control by allowing users to designate certain files as "shared". The database would treat these files specially and disallow non-serializable access to them. In such a case the proxy would select an appropriate database schema and utilize Postgres transactions and locking to ensure that safe concurrent access occurred to the requested files.

3 Virtual Directories and Text Search

One of our main goals in designing this prototype system was to expose database functionality to NFS clients, with little or no modification to the existing interface. An interesting abstraction that has been proposed in the literature for this purpose is the *virtual directory*, which was introduced for "semantic file systems" [6].

The main idea is to extend the naming semantics of existing tree-structured file systems in order to support query-based access. The names of virtual directories, specified using a special syntax, are interpreted as queries. Otherwise, the interface remains compatible with existing applications. Virtual directories can be implemented using a number of semantic paradigms, and we discuss the potential tradeoffs in Section 3.1. In Section 3.2, we describe how virtual directories can be used for the specific purpose of querying text.

3.1 Semantics of Virtual Directories

We encountered a number of tradeoffs in choosing the exact semantics of virtual directories in our prototype. In this section, we first outline the semantics we chose, and then we discuss some alternatives.

In our implementation, we took a simple approach, and chose to think of virtual directory creation as a one-time search. In other words, the contents of a virtual directory are determined when the directory is created, and they represent a snapshot of the file system at that point in time. If the user wishes to re-execute a query, he or she mush delete the existing virtual directory, and then recreate it. Additionally, in the prototype, the contents of each virtual directory are *hard links* to the files satisfying the given query. This ensures that the files in the virtual directory will continue to exist in the system until the virtual directory is deleted.

In addition to the semantics implemented in the prototype, there are a number of possible alternatives, and we describe a few.

- **Symbolic Links** A simple alternative to the proposed semantics is to create virtual directories that contain a set of *symbolic links* to files satisfying the associated query. We chose not to use symbolic links because of the potential for dangling or broken links when a file is deleted. Nonetheless, this design is also perfectly valid.

- **Lazy Query-Processing on `readdir`** The virtual directories described in the semantic file system paper [6] were based on lazy query evaluation. The given query was re-processed each time the contents of the virtual directory were read (using `readdir`). The added benefit of this approach is that it guarantees the contents of virtual directories are always up to date when they are read. However, repeatedly re-running these queries will likely put much greater pressure on the NFS proxy, as well as the database back-end, as the contents of virtual directories cannot be cached. This semantics is also perfectly valid, but the performance implications in a distributed environment should be carefully considered based on the anticipated workload.

- **Dynamic Virtual Directories** A third alternative we considered is dynamically updating the contents of virtual directories every time a file is created, updated, or deleted. This semantics can be implemented using database triggers. However, it is likely to suffer the same set of performance drawbacks as the lazy query-processing approach.

3.2 Text Queries and Virtual Directories

Our prototype currently supports two types of queries for text, both of which are implemented in the Tsearch2 Postgres extension [4], and which offer greater functionality than the `grep` utility:

- **Boolean Keyword Queries** These queries are logical expressions, and every document satisfying the logical expression is returned. For example, consider the query ('file' & 'system') | ('disk' & 'drive'). Documents containing either the words 'file' and 'system', or the words 'disk' and 'drive' would satisfy the query.

- **IR Rank Queries** Numerous notions of rank have been proposed in the information retrieval literature. We stress that our architecture supports the use of various rank functions, but the rank utility in the current prototype is based on proximity. Intuitively, the rank function counts how many times each word appears in the document, and also takes into account how close the search terms are to one another. Optionally, these rank scores can be normalized by document length.

As mentioned previously, one of the ideas of virtual directories is to extend the existing directory semantics to encode queries. In our prototype, we encode text queries in the names of virtual directories using a special syntax. Specifically, text queries are enclosed on percent signs (%). For example, consider the following command:

```
> mkdir %(computer&architecture)%
```

This command indicates the creation of a virtual directory containing text files about computer architecture.

We found that the easiest way to indicate rank, without modifying the NFS interface, was to simply append the ordinal rank to the name of each file in each virtual directory. For example,

```
> ls %(computer&architecture)%
[Rank_1]Chip_Design.txt
[Rank_2]Digital_Logic.txt
[Rank_3]Computer_Aided_Home_Design.txt
```

Alternative mechanisms are possible, such as modifying the `ls` function to order files by rank, but would require modification to the client.

4 Prototype Implementation

In this section we discuss the database schema that we implemented, the intricacies of the tsearch2 database module, and the essential features of our NFS proxy. We relied on four sources of open-source code. The largest piece of code is the Postgres database, version 7.4.7, which was the latest version at the time of this implementation [3]. Packaged with the Postgres distribution is the tsearch2 module that we discuss in section 4.1 [4]. We utilize a user-space implementation of the NFS server version 2 [8]. A kernel implementation of the NFS server could have worked for us, but as we were leveraging the code written by Halverson and Samios, we were more or less forced to stick with it, as Halverson's code is a modification of it.

4.1 The Tsearch2 Module

The tsearch2 module was designed to provide search capabilities for text documents within the Postgres database. The essential type it provides is "tsvector" The related "to_tsvector" function createsa a tsvector element from an element of type text. For example: to_tsvector('Hi mom, hi dad.') creates the following tsvector: 'hi':1,3 'mom':2 'dad':4. This tsvector indicates that 'hi' appears at positions 1,3 in the sentence, while 'mom' and 'dad' appear at positions 2, 4 respectively. This position information is important for ranking search results. The index that the tsearch2 module provides utilizes the words as search terms along with a reference to the file in which the word occurred and the position at which the word was found.

So why is this faster than `grep`? A utility like grep scans through files on disk, searching for the specified

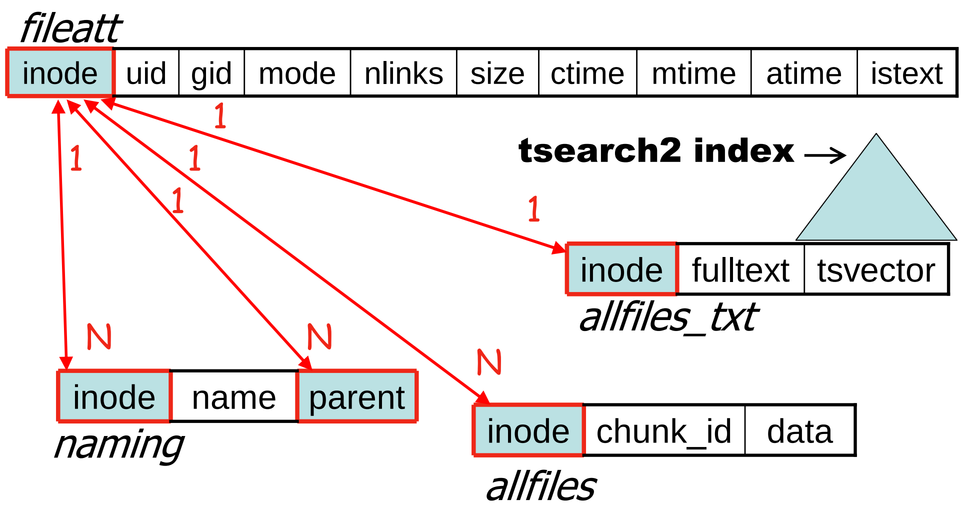

Figure 2: Database schema for prototype implementation

search string. Recall that each element of type text is reduced to a tsvector. The space savings we get from throwing out the stopwords and storing duplicated words only once is partially offset by the positioning information that is kept for each word occurrence. The real savings comes thanks to the index, which is built on top of Postgres' Generalized Search Tree (GiST) template for building indexes for user-defined types. GiST search trees are structured to be similar in structure to B+ trees, the most common index format in most databases. In the case of the tsearch2 module a GiST tree was customized for the purposes of storing the mapping from words to the files in which they occur. Thus using a search word as a key in the index, the occurrences of the search word will be found in no more than logb(N) disk accesses, where b is the fan-out of the tree and ranges from 100 to 1000 depending on the data it indexes. Compare this to the cost of reading all text files to seek out potential keyword matches.

Tsearch2 incorporates some common information retrieval tricks in performing its text search. As seen in the "Hi mom, hi dad." example, it ignores capitalization. It additionally uses a stopwords file which contains the most frequent, and therefore the least informative words (for search purposes), in the English language. No stopwords are stored in tsearch2's tsvectors or in its index. Some common stopwords are "a", "the", "of", etc. Stemming is also performed, which entails the reduction of "walks", "walked", and "walking" to "walk". For the purposes of search, all variations are considered equivalent, and the same stemming is also applied to the search terms. These are useful search tricks, especially when the user is unsure of the exact wording in a file, and will always return at least as many results as a `grep` query would, which is good for users who are still learning the semantics of the search functionality.

The tsearch2 library implements a ranking function with some tunable parameters. It incorporates the frequency of each search term in its scoring for each file that it considers. In addition, if there are multiple search terms, it uses the proximity information stored in the tsvector to influence the score. This is particularly useful, as the user is likely to search based on terms that occur together in a sentence, or may include an individual's first and last names in a search. The tsearch2 module also allows for the inclusion/exclusion of the inverse of document length in its ranking score. Intuitively, if a search terms appears the same number of times in a long document and in a short document, the shorter document probably features the term more prominently.

In our system, the tsearch2 ranking function will run over a collection of personal files and can afford to be simpler than the ranking function of a search engine. This is true because the number of hits will usually be orders of magnitude fewer in our system, which makes ranking easier and slightly less important, the emphasis instead being that all files that match the search terms must appear in the results. Furthermore, a commercial search engine must deal with documents that seek to bias their rating by exploiting knowledge of the search engine's ranking function, which is unlikely to occur in our implementation.

4.2 The Database Schema

Figure 2 illustrates the schema that our database utilized. The "fileatt" table keeps track of all the standard filesystem metadata. The "nlinks" field keeps track of the number of hard links pointing to a file. We use hard links extensively when creating semantic directories and can only delete a file once nlinks reaches 0. The "naming" table keeps track of the directory structure and contains a tuple for each file or link in the filesystem. All of the text files that the user would like to index are in the "allfiles_txt" table. The remainder of files are not indexed and are placed in the "allfiles" table.

Our schema differs from the one-table schema of [7] in two respects. The first difference is the "istext" attribute in the "fileatt" table and second is the "allfiles_txt" table which did not exist in their implementation. Currently text files are identified by the presence of a ".txt" extension.

A relational database will not allow a subset of the pages that comprise an attribute in a tuple to be modified. For this reason, each non-text file stored in the "allfiles" filesystem is subdivided into 4096 byte chunks and an entry for each chunk is placed in the "allfiles" table along with the file's inode. When a file is read the chunks are collated together and sent across to the NFS client. When an NFS write occurs to a subset of the bytes of a file, the write is directed to the affected chunks and an entire rewrite of the file is avoided. An additional benefit to the division of these files into even-sized chunks is that they made block-level server-side caching particularly easy to implement. The server-side caching allows the proxy to fill NFS_read requests for data without having to invoke the database each time.

The allfiles_txt table contains the necessary additions for text search capabilities. The tsvector field stores the word occurrence information in a format amenable to the text index. A trigger is set on the fulltext field of the allfiles_txt table so that whenever the file is updated, the corresponding tsvector entry is recomputed using the to_tsvector function and the index is updated.

The "allfiles_txt" and "allfiles" tables also differ in that files in "allfiles_txt" are not divided into chunks, rather the full text of the file is stored in a single field. The decision to do this was driven by the workings of tsearch2 module. It would not work to create a tsvector for every chunk of a file because words lying across chunkID boundaries would get cut in half and not get indexed. Another problem with this approach is that, boolean searches such as %alice & bob% would fail in the event that "alice" is in one file chunk and "bob" is in another. A viable alternative would be to add another table "text_chunks" in which the file is broken into chunks as in "allfiles" and to add a boolean "isStale" column to allfiles_txt. Reads and writes would then occur to "text_chunks" and the isStale flag would be set to true on writes to chunks int the "text_chunks" table. Unless stale data is acceptable, this still requires a lot of work to be done for searching since the entire fulltext field needs to be updated when even a small write occurs. The most efficient solution would be to customize the tsearch2 module so that it would work on files that are split into chunks.

5 Related Work

Over the years, there have been a number of projects and papers related to semantic file management, personal information management, text searching, and database-backed file systems. None of these individual elements is new. However, to the best of our knowledge, this is the first time a file system has attempted to expose user-defined file types, implemented in an object-relational database system, using a standard interface (NFS) and virtual directories.

The prior work that is most closely related to ours is the Inversion File System [9], which was also constructed on top of Postgres. This system allowed the user to implement new file types, which defined the kinds of meta-data associated with each type of file, new functions to operate on these types, and new types of indices. These user-defined types and functions were loaded automatically into the Postgres database manager. The system then allowed users to issue arbitrary queries over the meta-data associated with the various Inversion files.

The most obvious difference between the Inversion system and our work is the nature of the exported interface. The interface to Inversion included a set of special-purpose libraries, and the query interface was Postquel, the first query language supported by Postgres. For this reason, in order to access Inversion files, applications were required to use these libraries, and in order to leverage the query functionality, users were required to write queries in the (somewhat cumbersome) Postquel language.

The designer of Inversion mentioned its interface as a potential shortcoming. Indeed, in the years since the Inversion paper was published, we suspect that this problem has only grown. NFS continues to be an almost ubiquitous standard, which is relied upon by many existing applications. Also, because the number of people using computers has grown rapidly over the past twelve years, and includes many people not trained in computer programming, we suspect that an interface based on a query language such as Postquel or SQL would now be even more difficult for the typical user.

The interface exported by our prototype is more similar to that used by the Semantic File System [6]. As we mentioned previously, the main premise of this paper was that declarative queries could be integrated into an existing tree-structured file system by extending the naming semantics of files and directories, using a paradigm called "virtual directories." Otherwise, the existing interface (NFS in this case) was preserved for existing applications.

To a lesser extent, the Semantic File System was also extensible. User-programmed "transducers" were used to extract meta-data from files of various types, and the meta-data was then indexed in the file system. However, the indexing component was based solely on B-Trees, and could not be extended to new index types. Additionally, the system used a custom-written query processing module (and query optimization heuristics) rather than taking advantage of existing technology found in modern database systems.

In more recent related work, our database-backed NFS implementation closely follows the design used by Halverson and Samios's project at the University of Wisconsin [7]. However, as mentioned previously, this work did not take advantage of the query power available in the underlying database. For this reason, our work can be considered complementary.

Finally, a number of recent and forthcoming commercial products, including Apple Spotlight [2] and WinFS [1], seek to integrate text management, content indexing, and search into personal file management. However, to the best of our knowledge, none of the existing products can be used in conjunction with common distributed file system protocols, such as NFS.

6 Summary and Conclusions

In this paper, we presented a prototype architecture for building extensible semantic file systems. This architecture has two key sets of advantages that have not previously been implemented in a single system. First, the exported interface exposes query functionality using virtual directories, as proposed in [6], while maintaining "backwards compatibility" with the NFS interface. Second, by building the system on top of an object-relational database, as proposed in [9], we obtain easy extensibility for new file types and indexes, and we are able to take advantage of existing query processing functionality. As an added benefit, the database-backed architecture also allows for strong transactional semantics and recoverability.

We demonstrated the feasibility of this architecture using an important sample file type: text. Full-text indexing and query support is implemented as an extension to Postgres, and we were able to expose two key pieces of functionality using the existing NFS interface and virtual directories: boolean keyword search, and document rank. Indeed, our implementation required no modification to an existing NFS client.

There are several interesting conclusions to be drawn from this work, the most important of which is simplicity. The three-tiered proxy architecture provides a great deal of flexibility. By combining existing components, including an existing extensible database system and an existing NFS client, it was relatively easy to expose new text-search functionality to the user, without disturbing the existing file system interface or the core database implementation. Additionally, the extensible database system provides a flexible interface for defining new file types, which can be exposed to the user.

7 Future Work

There are a number of promising areas for future work, many of which have been mentioned throughout the paper. In the short term, there are two important issues to be addressed. The first issue is incorporating block-level text writes and block-level caching into our prototype. At present, when a single block of a text file is written, our implementation requires the entire document to be re-written. An enhanced implementation would support block-level writes.

The second near-term problem is performance benchmarking. Due to time constraints, we were not able to perform a thorough performance analysis of our prototype. In the near-term, there are three important questions related to the performance of text indexing, query support, and virtual directories:

- **Dynamic index maintenance** In the prototype, text indices are maintained automatically by the database, using triggers, when files are created, updated, or deleted. In the current implementation, we expect this to be the main source of performance overhead related to full-text management. However, this overhead is likely to vary tremendously based on the workload, and has not yet been quantified.

- **Virtual directory creation and text querying** Because of the full-text indices, we expect the performance of full-text query execution to be pretty good. An experiment would measure this cost, as compared to a "strawman" utility that processes these queries by simply scanning all files, like `grep`.

- **Performance comparison of alternative semantics** As mentioned in Section 3.1, there are a number of alternative semantics for virtual directories, two of which re-process the associated query, either when the directory is read, or when files are updated. We hypothesize that this would cause an additional performance burden for the database backend, as well as the network, but a set of experiments would quantify this cost more precisely for certain types of workloads.

In the longer term, there are several additional issues to be considered. The first relates to the three-tier architecture. Exploiting this design, we suspect that it would be possible to track use patterns, inside the NFS proxy, to enable adaptive reorganization of indices and storage. File systems in general, as well as our query utilities, can be used in a number of ways. We suspect that this workload would influence the design of schemas and choice of indices for both the meta-data and data files in the database. Unfortunately, these workloads may not be well-understood, particularly because few file systems have been implemented in this way, and none is in widespread use. Further, following our theme of transparency, it is important to hide database performance tuning from the NFS client.

Finally, integrating text-search into an existing file system is an interesting problem. However, as mentioned in the introduction, text management serves as just one example of a new type of file system that is easily extended to manage new file types based on semantic content, with minimal modification to the existing interface. An important future step is to extend the existing prototype to handle indexing and querying meta-data and content of other types of files, such as images and music.

8 Acknowledgements

We wish to thank Alan Halverson for providing a large amount of useful source code from his 2002 project, Remzi Arpaci-Dusseau for a number of useful comments, and David DeWitt for lending us an unsupported machine on which to run our prototype.

References

[1] http://msdn.microsoft.com/data/winfs/default.aspx.

[2] http://www.apple.com/macosx/tiger/spotlight.html.

[3] http://www.postgresql.org.

[4] O. Bartunov and T. Sigev. Tsearch2 full text extension for PostgreSQL. http://www.sai.msu.su/ megera/postgres/gist/tsearch/V2.

[5] R. Boardman, R. Spence, and M. Sasse. Too many hierarchies? The daily struggle for control of the workspace. In *Proc. of Human-Computer Interaction International*, 2003.

[6] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic file systems. In *Proc. of the 13th Annual Conference on Operating Systems Principles*, October 1991.

[7] A. Halverson and B. Samios. NFS meets data bases. In *Proc. of the First Annual USEDNIX Conference*, December 2002.

[8] O. Kirch. UNFSD - The Universal User-mode NFS Daemon. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir.

[9] M. Olson. The design and implementation of the inversion file system. In *Proc. of the 1993 Winter USENIX Conference*, January 1993.