Extended File System Metadata Management with Relational Databases

Michael McThrow

Storage Systems Research Center
University of California, Santa Cruz

Abstract

Modern file systems need to handle extended metadata. Existing file systems are not equipped to handle managing metadata in the amount of files and the diversity of files that these file systems are now supporting. Users need better searching and querying capabilities. Metadata within files still remain application- and file format-specific and is often proprietary, which makes searching difficult. We propose a file system that combines the power and proven technology of relational databases with the Linking File System's (LiFS) ability to handle extended metadata.

1. Introduction

Over time, file systems have dealt with multiple types of files, such as text documents, spreadsheets, photos, music, and movies. As file systems have to deal with a growing amount of files and file formats, the need for file systems to manage extended metadata has also grown. For many years, file systems didn't have a common API for handling extended metadata. Because of this, applications were forced to put their file metadata inside of the file. The problem with this approach is that there wasn't a common interface and common format for extended file system metadata. File metadata was often only accessible to the application, and the file format was usually proprietary, which made searching for files based on their metadata difficult.

The Linking File System (LiFS), an earlier research project (and will be known as "the existing file system" throughout the paper when comparing it to our new research) addressed these problems [1]. It addressed these problems by creating an file system that supports extended file system attributes and relational links between files. Extended file system attributes are application- or user-defined values consisting of a *key* and a *value* [1]. For example, an MP3 file might have a *key* called "Artist" and a *value* called "The Beatles." Relational links also provide a rich way of describing files. For example, to continue using The Beatles music as an analogy, instead of organizing all songs from the album *Abbey Road* in a directory, each song can have a relational link with the other files. Relational links also have extended attributes with keys and values. For example, each of those relational links would have a key called "Album" and a value called "Abbey Road." With relational links, a user does not need to organize files in directories, although for convenience and backwards compatibility, directories can be emulated by creating a file of 0-byte length and setting links to other files [1].

LiFS did a great job at handling extended file system metadata. However, we wanted to go further. We were very interested in replacing the traditional data structures with a relational database. We want to know if relational databases are the best way to manage extended file system metadata.

2. Motivation

There are a few key reasons why were are interested in relational databases for extended file system metadata management. One reason is that relational databases are a well-established and proven tool for managing data. Relational databases have been around since 1970 [6], and relational databases have over three decades worth of improvement and optimization. Because data management is an important part of the LiFS, we would like to take advantage of the advantages and years of optimization that relational databases have to offer.

Another reason why we are interested in relational databases is because they are queryable. Instead of coding special searching functions for relational links and attributes with traditional file system data structures, we can use a querying language, such as SQL, to search for extended metadata. By making a file system queryable using relational databases and queries, it becomes easier to search for files.

We want to know if a relational database is the best way to manage extended file system metadata. Relational databases are a proven technology with many capabilities such as powerful querying languages. The following sections will describe the design, implementation, and the results of our research.

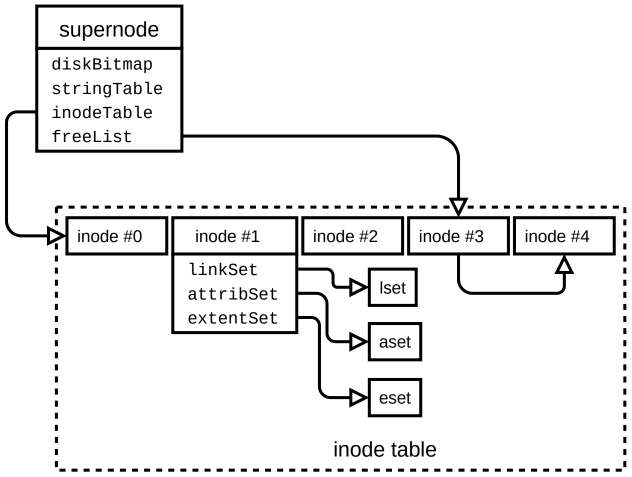

Figure 1. Structure of the existing file system's supernode

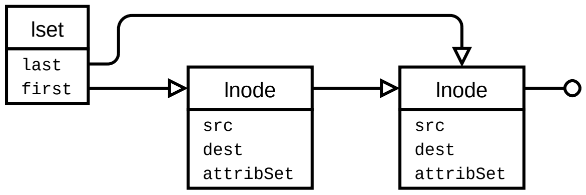

Figure 2. Structure of a set of links

3. Design

3.1. The Existing File System

The existing file system uses native data structures optimized for low latency storage class memories [1]. By the term *native data structures*, we mean traditional data structures used for file systems, such as linked lists and trees. Since the focus of our research is about extended file system metadata management, we will only focus on the data structures used in the existing file system that pertain to that topic. The existing file system contains inodes, link nodes (*lnodes*) and extended attribute nodes (*anodes*).

As shown in Figure 1, the file system is contained into a large node called the *supernode*. The portions of the supernode related to extended metadata management consist of a linked list of inodes and a table of strings used to eliminate duplicate copies of strings when used to search for extended attributes [1]. Each inode consists of traditional inode data determined by the POSIX file system specification, as well as a linked list of lnodes and anodes. As shown in Figure 2, each lnode consists of a source inode, a destination inode, and a linked list of anodes. Figure 3 shows a linked list of anodes. Each anode contains a *key* and a *value*, both pointers to the string table entry for those values [1].

Several new file system calls were created for the use of the existing file system [1]. These system calls consist of relational link creation, relational link removal, setting link attributes, returning the set of corresponding links to a file, and creating a directory structure. The setting and getting of file attributes were implemented using the getxattr() and setxattr() system calls [1].

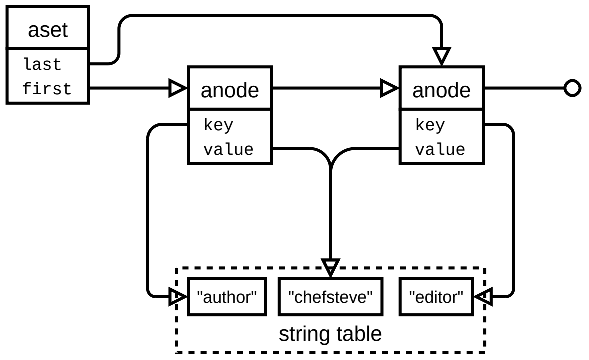

Figure 3. Structure of a set of attributes

3.2. The Relational Database

Figure 4 shows the relational database schema of the aforementioned native data structures, and also shows how each attribute in each schema relates to other attributes. The database schema contains four schemata: *inode* that represents a table of inodes, *lnode* that represents a table of links, *i_anode* that represents a table of extended attributes for inodes, and *l_anode* that represents a table of extended attributes for links. The inode schema contains an attribute containing the inode number, as well as individual attributes containing traditional inode data determined by the POSIX file system specification. The lnode schema contains attributes containing a source inode number, a destination (target) inode number, and a ID value for the link used for relating each lnode tuple to l_anode tuples. The i_anode schema contains an attribute representing the cor-

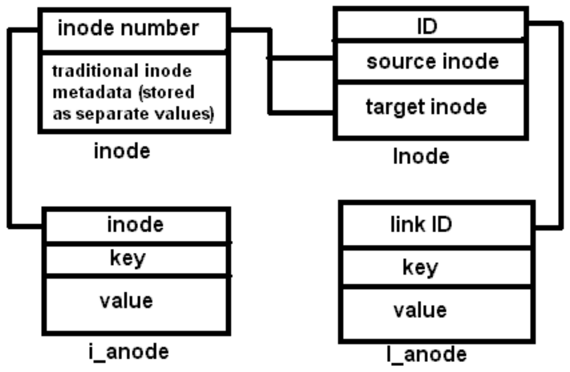

Figure 4. Relational database schema

responding inode number, as well as an attribute each representing a *key* and *value*; both attributes being character strings. The l_anode schema only differs from the i_anode schema by the fact that the corresponding inode attribute is replaced with a corresponding lnode attribute.

Each schema in the relational database schema contains an index on the attributes that represent their ID value (e.g., the *inode* schema has an index on the inode number). Indices vastly accelerate queries in which the attribute that is indexed is specified [6]. Since the queries that we have implemented in our test cases rely on the relations between different schema and their attributes, and since ID values are used in the database to relate schema, indices will be a requirement in the design of our database. The drawback to indices is that they increase the time of insertions, deletions, and updates of those relations [6]. However, we anticipate that the file system will be queried more often than users would insert, delete, or update files.

Each system call in the existing file system has been rewritten as an equivalent using SQL queries. For example, adding an extended attribute to a file requires two tasks: an inode lookup for the file path given, as well as an INSERT statement to the *i_anode* table, along with a *key* string and a *value* string.

4. Implementation

The relational database is implemented using SQLite. SQLite is a lightweight, public domain C library that does not require any configuration, can easily be embedded into an application, supports databases up to two terabytes in size, and has a footprint of less than 250 kilobytes [5]. SQLite also supports in-memory relational databases, which is a requirement for our research. We feel that SQLite's support for in-memory relational databases, as well as its small footprint and public domain status, is the best choice of database API for our research.

We decided that we will test the performance of the various SQL queries needed to emulate the system calls before actually implementing the file system. By analyzing the performance of the SQL queries in the relational database before implementing them in an actual file system, we will be able to optimize our database queries and schema in our future research. Therefore, our results are not based on an actual file system; they are based on the speed of each system call implemented with SQL queries.

5. Results

The relational database was implemented and evaluated on a Sun workstation running the Linux kernel 2.6.9-ac11. The system is configured with an AMD Opteron 150 processor running at 2400MHz and one gigabyte of RAM. The version of SQLite used for the relational database is SQLite 3.3.6. The relational database was compared with the tests of the existing file system, which was tested on the exact same system [1]. All of the relational database tests were ran using an in-memory database, and for each test, the database was freshly created.

5.1. Standard File System Operations

We tested the relational database for six standard file system operations: creating a directory tree, creating files, setting extended file attributes, and retrieving extended file attributes. In order to directly compare these tests to the existing file system, we created trees of *k* directories, *n* files per sub-directory (there are no files in the root directory, however), and a depth *d*.

Figure 5 shows a graph comparing the tests of the existing file system to the tests of the relational database. In the area of file creation, the relational database is competitive with the existing file system. Directory tree creation is slower in the relational database than with the existing file system. However, we would like to note that in the relational database, creating an empty directory uses the exact same steps as creating a file. The setting and retrieval of file attributes, however, is much slower with the relational database than with the existing file system.

Why are the tests for setting and retrieving file attributes are magnitudes slower than their native data structure counterparts? In those tests, the directory tree must be traversed in order to obtain the inode numbers of the subdirectories and the directory's files. In order to get information about subdirectories and the files of the current directory, we have to retrieve a table of all of the links of the current directory (which is done by calling the sqlite3_get_table() function and sending it a SELECT query, which dynamically allocates a string table containing the tuples that match the query), and check the links to make sure that link is either a subdirectory or a file within

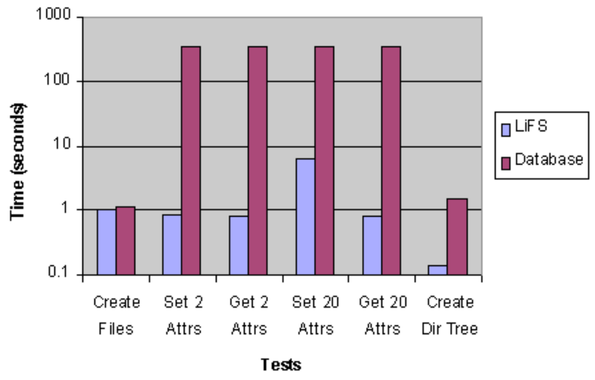

Figure 5. Time required for various tests on a directory tree where $k = 5$, $d = 5$, and $n = 4$. 16,520 total files were created.

that directory. To make these tests easier, we didn't combine these links with any relational links, and we took advantage of the fact that the first *k* links of the directory are subdirectories, while the last *n* links of the directory are files. Still, the combination of dynamic allocation of memory for retrieving tables obtained through SQL queries, as well as processing that data, is the reason why our tests are much more expensive than the tests for the native data structures.

5.2. Link Operations

We attempted to test the relational database for relational link operations: creating random links between files, setting link attributes, and deleting links. However, we ran into some major performance issues related to directory traversal. In order to convert a path name into an inode number, which is done in all of our relational link operations, the directory tree must be traversed. For each directory in the path (including the root directory), the database does a query on that current directory, and then moves on to the next directory in the path. We have calculated that the average time of each SQL query is 21.5 ms. In a directory tree where $k = 5$, $d = 5$, and $n = 4$, there is an average of 5.75 queries per path, which means that it takes an average of 123.625 ms to execute the SQL queries for the average path. Since there are 3,906 directories in this directory tree, and since each relational link test operation requires that the inodes for both a source path and a target path are found, the total estimated time that it will take to perform inode lookups for 15,620 links is 965.7585 seconds, or 16 minutes and 5 seconds.

Table 1 compares the performance of relational link operations of the existing file system to the estimated performance of the relational database. We ran these tests on the relational database without having to convert paths to inodes (we stored all used inodes in an array in this test), and then added the results of those tests to the aforementioned estimated total time it takes to perform inode lookups of these links. If the path name conversion overhead is ignored, then the tests of the relational database are mostly competitive with the existing file system's tests. This shows the overwhelming impact that inode lookups have on the performance of the relational database.

Attrs	Test	LiFS	No Traversal	With Traversal
2	Create Links	1.073	3.140	968.899
2	Create Attrs	1.148	1.094	966.853
2	Remove Links	1.086	2.014	966.845
30	Create Links	1.054	3.154	968.913
30	Create Attrs	2.751	17.180	982.939
30	Remove Links	1.288	2.938	968.697

Table 1. Time in seconds to create 15,620 random links over a directory tree ($k = 5$, $d = 5$, $n = 4$) with 2 and 30 attributes on each link. "No traversal" is the relational database tests without directory traversal, and "with traversal" is the relational database tests with the overhead of directory traversal.

5.3. The Impact of Dynamic Memory Allocation

The very slow performance of much of our tests have been affected by the amount of dynamic memory allocation required. Retrieving tables from an SQL SELECT query requires dynamically allocating memory to create a string table. As shown in the link operations, these queries can take as long as 15 ms. This not only includes the time required for dynamic memory allocation, but also the time the SQLite library uses to parse, compile, and execute the SQL query. There also seems to be no way to avoid dynamically allocating memory for tables retrieved by SQL queries, since there is no support for pre-allocated arrays if the application knows how many entries are in the database in advance.

6. Related Work

One attempt at using databases for handling metadata in file systems was the Inversion File System. The Inversion File System was a database file system that handled both file data and file metadata [4]. The file system was implemented with the POSTGRES 4.0.1 database. The Inversion File System also supported features such as transactions, fine-grained time travel, fast recovery, and *ad hoc* file queries on both the metadata and the file data. It performed at between 30 and 80 percent of the speed of the native ULTRIX file system over NFS, when tested for the same operations [4].

Apple Spotlight is an example of a database on top of the file system that supports extended metadata [3]. Spotlight indexes files in the file system and stores their metadata in a database. The metadata stored by Spotlight is dependent on importers that read the file format and "imports" the metadata to the Spotlight database. There are two big differences between Spotlight and our research. One major difference is that Spotlight doesn't support relational links between files. Another difference is Spotlight is built on top of the file system in Mac OS X, while our research is interested in building a file system around our database (although the actual file data won't be stored in the database). Since Spotlight is built on top of a file system rather than an integral part of a file system, Spotlight doesn't have to deal with directory traversal, file and directory creation, and other file system-level operation levels that our relational database has to support. The Spotlight database is implemented with Apple Core Data, an API that "provides a general-purpose data management solution developed to handle" data for various Mac OS X applications [2]. One of the databases that Core Data supports is SQLite, which is the exact same database that we are using in our research.

7. Future Work

We need to optimize the performance of our database. First, we need to find the optimal schema for our database. Next, we need to review our SQL queries and also optimize them. We also need to further investigate the impact of dynamic memory allocation and find some ways to bypass or mitigate the effects of it. After we have finished optimizing our database, then we will implement it as an actual file system and compare its performance to other file systems. We would like to see how a file system created based on our research would perform in real-world situations.

8. Conclusion

Despite some setbacks with performance, our research still looks promising. The need for modern file systems to handle extended metadata is still great, and we still feel that databases are the way to go. However, we have a lot of optimization to do before we are able to proceed further with this research. The performance penalties are much too great for this research to be used in a serious project as of now. However, after optimizing our database schema, we feel optimistic that our results would be more competitive with the existing file system and other current file systems.

9. Acknowledgments

I would like to thank Carlos Maltzahn for his extensive amount of suggestions and support with my research. I would also like to thank Scott Brandt for his suggestions and his support, as well as Kevin Greenan for answering all of my questions about the test cases used in the existing file system.

This work was completed as part of the University of California, Santa Cruz's SURF-IT (Summer Undergraduate Research Fellowship for Information Technology) program, an NSF CISE REU Site. This paper is based upon work supported by the National Science Foundation under Grant No. CCF-0552688.

References

[1] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hoffman, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. LiFS: An Attribute-Rich File System for Storage Class Memories. In *23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, May 2006.

[2] Apple Computer. Developing with Core Data. http://developer.apple.com/macosx/coredata.html, March 2006.

[3] Apple Computer. Working with Spotlight. http://developer.apple.com/macosx/spotlight.html, June 2006.

[4] Michael A. Olson. The Inversion File System. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, January 1993.

[5] SQLite. SQLite Home Page. http://www.sqlite.org/, September 2006.

[6] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall, 1997.