Design for a Tag-Structured Filesystem

Adrian Sampson

May 12, 2008

Abstract

Tagging is an organizational system commonly used as an alternative to hierarchical systems. Many authors have recognized the desirability of a filesystem accessed with a tagging interface, as opposed to or in addition to a traditional directory interface. Existing tagging filesystems universally use conventional, hierarchical filesystems as a backing store. This paper examines the challenges in designing interfaces, on-disk structures, and file layout in tag-structured filesystems and proposes various solutions.

1 Introduction

Tags, sometimes called keywords, are arbitrary metadata strings used to organize collections of objects. In a tagging system, objects are annotated with an arbitrarily large set of tags; the collection can then be queried to find all objects possessing a certain tag. In more complex systems, more sophisticated queries are possible: tag intersections query the collection for objects matching several tags and so on.

Tags have recently gained popularity through their use in social Web applications such as del.icio.us [1] and Flickr [2]. In these applications, a tagging system can be seen an alternative to a hierarchical system of organization. In fact, from the point of view of a desktop user, hierarchies present many usability problems that are not present in tagging organizational systems [5, 17, 18]. For instance, tagging systems do not require that objects have a single "location" like a pathname and can allow objects to be grouped in multiple meaningful ways.

That filesystems are traditionally organized in a directory structure encourages exploration of filesystems organized primarily by tags instead. While work on filesystems with extensible metadata is not a recent phenomenon [4, 10, 11, 19], the use of tags as a primary mode of access is a somewhat less-explored area. In particular, prior work primarily concerns itself with developing interfaces to tagging filesystems. To the best of our knowledge, no work has been published examining the design of a filesystem's on-disk layout and data structures according to a tag-based organizational strategy. This paper synthesizes previous work on tagging filesystem interfaces as a motivation for exploring the deeper design issues in creating efficient implementations for those interfaces.

We will henceforth refer to filesystems with tag-based organization interfaces as *tagging filesystems*. Filesystems whose block-level storage design reflects this type of organization (i.e., the type of filesystem proposed in this paper) will be referred to as *tag-structured filesystems* (TSFS).

1.1 Related Work on Tagging Filesystems

SFS [10] and HAC [11] are early examinations of several concepts used in tagging filesystems. In particular, they introduce the concept of the *virtual directory*, a pseudo-directory that "contains" the list of files matching a query defined by the directory name. By using virtual directories, users and programs accustomed to a directory-based organizational scheme can easily query the filesystem's metadata. These systems also promote the use of virtual directory hierarchies for query refinement.

More recently, SemDAV [17] and TagFS [18] have directly addressed the construction of interfaces to filesystems based primarily on metadata queries.

A wide variety of work on tagging filesystems has recently occurred outside academia. Several implementations of tagging filesystem interfaces have been published [3,6,13–15]. Each system includes a method for encoding tag queries as virtual directory names. Universally, the systems use a traditional filesystem as a backing store for files and an auxiliary database to store tags.

2 A Tagging Filesystem Interface

In the design of our TSFS, we will assume that the system may be queried with arbitrary logical expressions on tag strings. In particular, we assume that the filesystem exposes a native, non-POSIX-like interface for these queries that allows them to be arbitrarily complex. A secondary, POSIX-compatible interface may also provided, but it will be at most equally as expressive as the native interface.

We will further assume that no directory structure exists alongside the tag-based organizational structure. While such a hybrid model may be useful in some settings, this simplification frees us from the complexities of considering the two models' interactions.

Furthermore, our TSFS will implement no traditional metadata outside of the tagging system. Tags act as arbitrary metadata fields and storing certain types of metadata separately is unnecessary. The modification or creation timestamps traditionally associated with files, for instance, may be easily encoded as strings and stored as tags. The operating system could enforce special treatment of certain tags using prefixes; it might, for instance, store modification times as tags beginning with `mtime:` and prevent user-space tools from modifying tags with this prefix. Separating this logic from the on-disk structure of the filesystem permits compatible changes to the set metadata fields in future or specialized systems. It also provides uniform methods of access to metadata, simplifying the structure of metadata queries.

The filename attribute is particularly difficult to support in a tagging filesystem. Because no directory structure exists to distinguish files with the same name, filenames would need to be globally unique. Such a requirement is unnecessarily restrictive. Filenames can be easily approximated in tags if necessary by using "singleton" tags. Directory structures themselves can also be simulated using tag conjunctions. Because the set of files with both of a pair of tags, T_1 and T_2, is a subset of the set of files with one of the tags, $T_1 \wedge T_2$ can be seen as a "subdirectory" of T_1 or T_2. By building up these structures, the user can systematically identify files without filenames.

Based on this interface, our TSFS design will attempt to optimize the following basic tag operations:

- *Inode-to-tags.* Look up all the tags associated with an inode.
- *Tag-to-inodes.* Find all the inodes associated with a tag.
- *Add tag to inode.* This must be supported efficiently whether or not any other inodes in the filesystem have the same tag.
- *Delete tag from inode.* The filesystem must consider the possibility of "orphaned" tags, those associated with no inodes. These must be cleaned up efficiently where necessary.
- *Tag conjunction and disjunction.* In a less common usage scenario, the intersection or union of the sets of inodes associated with multiple tags may be requested.

3 Metadata Storage

In designing the data structures for our proposed TSFS, we will assume that tags are short strings. As they are intended to be efficiently matchable and are not intended to replace more general metadata like extended attributes, we will assume that their length is on the order of that of filenames.

3.1 High-Level Tag Association Model

In a tagging filesystem, two types of objects must be stored: inodes, which list the addresses of blocks containing a single file's data, and tags, which represent files' metadata. A tag-structured filesystem must have a mechanism for storing tags, inodes, and the relationships between them. Three basic, high-level approaches to storing this data exist:

- *One table.* The inode table is the only table. Each inode retains (that is, contains or refers to) a list of tags that apply to the inode.
- *Two tables.* In addition to the inode table, a tag table is stored. Entries in the tables refer to one another in order to associate inodes with tags. Two basic two-table models exist.
  - *Two tables with inode-side references.* Each inode retains a list of references to entries in the tag table that apply to the inode.
  - *Two tables with tag-side references.* Tags retain references to inodes they apply to; no reference exists from inodes to tags.

- *Three tables.* Neither tags nor inodes contain references to one another. Instead, a third table associates tags with inodes. Each entry in the tag/inode association table contains a reference to a tag and a reference to an inode.

Each model has advantages and disadvantages for performance. The one-table model and the two-table model with inode-side references both allow fast inode-to-tag operations. In both, however, tag-to-inode lookups must search the entire inode table for matching inodes; the tag-to-inode run time is linear in the size of the inode table. Conversely, the two-table model with tag-side references allows fast tag-to-inode operations but prohibits fast inode-to-tag lookups for systems with many tags. In the three-table model, both operations must perform lookups in the tag/inode association table and are bounded by that table's lookup speed.

The approaches also vary in their efficiency with respect to storage space. The one-table model, for instance, can be seen as inefficient because it must store tag names multiple times—once for each inode to which a given tag applies. Because tags are short, however, this overhead is small.

To efficiently support all common tag operations, a hybrid approach must be used. In particular, we propose a two-table model with both inode-side references and tag-side references. The inode-side references to tags take the form of the tag string itself; because tags are assumed to be short, we contend that the overhead incurred by the duplicate storage of tags will be small.

We will evaluate the merits of this approach and its implementation details in section 3.2.3.

3.2 Low-Level Data Structures

In this section, we will propose on-disk data structures to store the two basic units in a TSFS: the inode and the tag.

3.2.1 The Inode Structure

The inode data structure in the proposed TSFS resembles the one in the UNIX File System (UFS) [16] and contains:

- The inode number.
- A constant number of fields that may contain block addresses that in turn contain the file's data. If the number of data blocks is less than the number of fields available, the unused fields are set to zero.
- A constant number of fields that may contain block addresses. The first such block is a singly indirect block, containing a list of block numbers which in turn contain file data. The second block is a doubly indirect block, containing a list of addresses of indirect blocks. Each subsequent field introduces an additional level of indirection.
- A fixed number of tag strings applying to the inode.
- Addresses of blocks with increasing levels of indirection that contain lists of tag strings associated with the inode, as above.

The ability to store tag strings directly in the tag structure optimizes for the case in which few tags are associated with a given inode; that external blocks are also possible allows for larger numbers of tags to be associated with a given inode.

Inodes and data blocks are stored on disk as in the Berkeley Fast File System (FFS) [12]. At filesystem creation time, the disk is divided into "cylinder groups" (ranges of contiguous blocks) and space is allocated at the beginning of each cylinder group for inodes. Data blocks are allocated in the non-inode space of cylinder groups; blocks are marked as free or used by bitmaps present in each cylinder group.

3.2.2 The Tag Structure

The tag data structure must contain a tag string and references to the inodes to which the tag applies. We propose a constant-size data structure that contains:

- The tag name, stored in a constant-size string field. If the tag string is shorter than the size of the field, it is null-terminated.
- A constant number of fields that may contain inode numbers of the files to which the tag applies.
- A constant number of fields that may contain block addresses of increasing levels of indirection. The blocks store additional inode numbers to which the tag applies.

As in the inode structure, this structure allows efficiency with small numbers of tags while still permitting larger tag sets.

Several efficient alternatives exist for efficiently storing the system's tag structures. Extensible hashing [9], for instance, would allow fast lookups at the expense of occasional slow expansions and a complex implementation. For simplicity, we propose the use of a B-tree (or a variant) indexed by the tag string. A B-tree supports efficient lookups and insertions while not requiring that the tags be stored contiguously. This flexibility is important because the filesystem has statically allocated data, such as the cylinder group headers and inodes, that must be accommodated by a dynamically allocated tag table.

3.2.3 Performance

As a rough evaluation of our design choices, we will briefly discuss the steps required to perform each of the common tagging operations defined in section 2.

- *Inode-to-tags.* The inode is read from disk. If the number of tags is small (less than the number of tag string fields in the inode), they can be read from the inode itself. Otherwise, additional blocks must be accessed; the number of additional accesses depends on the number of tags and the filesystem's block size. Appropriate selection of the number of direct tag string fields can optimize this operation for an expected number of inodes.

- *Tag-to-inodes.* The tag structure must be looked up in the tag B-tree. Then, a process similar to the inode-to-tag operation occurs: for a small number of associated inodes, the inodes may be read from the tag structure itself; otherwise, additional blocks must be accessed. Again, this operation may be tuned by adjusting the number of inode fields in the tag structure.

- *Add tag to inode.* The inode is looked up. Depending on the number of existing tags, the new tag is written to the inode itself, a new block is allocated for the tag, or the tag is written into an existing external block.

- *Delete tag from inode.* The inode is read from disk. If the tag string is present directly in the inode structure, it is erased. Otherwise, a search through the inode's tag string blocks occurs until the tag can be found and erased. If the erased tag was not the last tag on the inode, the last tag must be moved to the newly erased position to maintain consistency. This may require additional lookups into tag string blocks. Finally, the tag structure is looked up in the tag B-tree and the inode in question removed from it in a similar manner. If the tag no longer has any inodes, it is removed from the B-tree.

- *Tag conjunction and disjunction.* Two independent tag-to-inodes operations must occur; the results are then manipulated in memory.

The first three operations are clearly supported efficiently. Deleting tags, conjunctions, and disjunctions are areas in which the design may be improved in the future.

4 Inode and File Allocation

One major innovation of FFS is its attempt to place data on disk nearby other data that might be accessed at nearly the same time [12]. It accomplishes this by, for instance, placing inodes together in the same cylinder group when they are allocated in the same directory. Directories give strong hints about the temporal locality of file accesses.

This paradigm does not easily extend to tagging filesystems because they have no single most prominent determiner of file "relatedness" like the directory. Inodes may have more or less similar sets of tags; some tags may be more or less important to users when determining locality of file access. Interpreting users' selection of tags to infer file relatedness is clearly a large and difficult problem. For this reason, we propose several design alternatives that warrant closer examination.

Note that, because these policies do not affect the on-disk format of the filesystem, systems could have different policies but remain interoperable when reading and writing TSFS volumes.

1. *Single tag.* The user is allowed to tag each file with at most one tag beginning with a predefined prefix such as `groupby:`. Files are related if they share this tag; when the tag is added or changed, it is looked up in the tag structure B-tree to determine which inodes are related and should thus be placed nearby. This allows files to be clustered by a single criterion as they are in FFS. To optimize performance, the user must carefully select this specially designated tag for each new file. If the user does not use `groupby:` tags, files will be placed arbitrarily, possibly randomly, into cylinder groups. This

behavior may be acceptable in some systems with large or unpredictably accessed files.

2. *Raw tag similarity.* When an inode is created or tags are added to it, the inode's tags are each looked up in the tag structure B-tree. A list of inodes that share tags is constructed. The new inode or file data is placed as near as possible to the inode that shares the most tags with it.

3. *Manual tag priority.* The user provides the filesystem with a ranking of tags according to their importance when determining file relatedness. The process proceeds as in the previous design but the relatedness of inodes is weighted by the provided ranking.

4. *Automatic inference.* Systems have been proposed that observe user behavior to optimize file layout [7,8]. Such a system could be used to analyze which files are accessed together most often. The tags these inodes share could then be inferred to be strong determiners of file relatedness. The tag priority that is manually specified in the previous design might then be automatically determined.

5. *Wait for SSDs.* The advent of solid-state, random access storage technologies limits the utility of optimizing file locality. If a TSFS is designed with such technology in mind, no locality mechanism must be implemented.

Clearly, all but designs 1 and 5 will incur significant performance overhead when creating inodes and manipulating tags. Designs 3 and 4 especially can be expected to require large numbers of lookups in the tag structure B-tree. While some of this performance overhead may be hidden by caching the results of tag-to-inodes and inode-to-tags lookups, these solutions may be best suited to situations in which slow tag manipulation is acceptable. In particular, this might be acceptable in desktop settings wherein only user can be expected to issue tag-manipulation commands. Without careful optimization, sophisticated similarity algorithms will be impractical in settings where tag operations are likely to be automated.

5 Future Work

Many avenues for future work in the design of tag-structured filesystems remain. Design choices in this space will remain vague until more information is made available about the ways in which tagging filesystems are used.

While tagging organizational systems have clearly succeeded in some spaces including modern Web applications, it is not clear how users and developers will interact with a tagging filesystem. User studies should be conducted to measure the usability benefits of tagging filesystems and to determine which operations users tend to invoke most often. The habits of users could then be used to make decisions about both the interface and the structure of new filesystems. Similarly, efforts should be made to develop or port applications that take advantage of tagging filesystem interfaces. Profiles and traces of these applications will help to inform the design of optimized tag-structured filesystem.

With this new information, deeper exploration will be possible into mechanisms for predicting temporal locality of access in tagging filesystems and selecting cylinder groups for file allocation. In section 4, alternatives are presented that portray tradeoffs between performance and sophistication of file allocation techniques. With more information about tagging filesystems' use patterns, assumptions may be made that help determine the importance of performance and prediction accuracy in different situations. These assumptions will help simplify this particularly difficult aspect of TSFS design. A simple first step in this area might be to develop simple, efficient heuristics for locating files with similar sets of tags (with some similarity metric). Even if these techniques are not deterministic, their effectiveness in predicting locality of access may be assessed.

Further exploration is also needed into the optimization of logical operations on tags. With simple TSFS designs, including the one outlined in this paper, finding the intersection of two tags requires two complete tag-to-inodes operations followed by an in-memory conjunction. If intersections are often used with common tags to identify small groups of files, this design is wastefully inefficient. Algorithms, data structures, or both must be developed to support this kind of operation efficiently. Inspiration for such solutions may be found in database techniques that attempt to efficiently support logical operations in queries.

6 Conclusion

As previous authors have observed [17, 18], tagging filesystems offer desirable advantages to desktop users

as an alternative to traditional, hierarchical filesystems. Tag-structured filesystems must be created in order to make a tagging interface practical. However, the design of a TSFS presents many interesting challenges, most notably the policy for optimizing temporal locality among inodes and data.

References

[1] del.icio.us. `http://del.icio.us/`. Accessed May 2, 2008.

[2] Flickr. `http://www.flickr.com/`. Accessed May 2, 2008.

[3] Tagsistant. `http://www.tagsistant.net/how_works.shtml`. Accessed May 2, 2008.

[4] A. Ames, C. Maltzahn, N. Bobb, E.L. Miller, S.A. Brandt, A. Neeman, A. Hiatt, and D. Tuteja. Richer file system metadata using links and attributes. *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference on*, pages 49–60, 2005.

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

[6] Stefan Berndtsson. LAFS. `http://www.nocrew.org/~stefan/lafs/lafs-1.0.1.tar.gz`. Accessed May 2, 2008.

[7] S. D. Carson. A system for adaptive disk rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[8] P. Eaton, D. Geels, and G. Mori. Clump: Improving file system performance through adaptive optimizations, May 2000.

[9] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[10] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic file systems. *SIGOPS Oper. Syst. Rev.*, 25(5):16–25, 1991.

[11] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 265–278, Berkeley, CA, USA, 1999. USENIX Association.

[12] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[13] Stanislav Ochotnický. μTagFS. `http://www.kmit.sk/devel/utagfs/`. Accessed May 2, 2008.

[14] Pts Oldalai. movemetafs. `http://freshmeat.net/projects/movemetafs`. Accessed May 2, 2008.

[15] Mayuresh Phadke. dhtfs - readme. `http://code.google.com/p/dhtfs/source/browse/trunk/README`. Accessed May 2, 2008.

[16] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[17] Bernhard Schandl and Ross King. The SemDAV project: metadata management for unstructured content. In *CAMA '06: Proceedings of the 1st international workshop on Contextualized attention metadata: collecting, managing and exploiting of rich usage information*, pages 27–32, New York, NY, USA, 2006. ACM.

[18] Simon Schenk, Olaf Görlitz, and Steffen Staab. TagFS - tag semantics for hierarchical file systems. In *Proceedings of I-KNOW. 6th International Conference on Knowledge Management.*, pages 304–312, 2006.

[19] Zhichen Xu, Magnus Karlsson, Chunqiang Tang, and Christos Karamanolis. Towards a semantic-aware file store. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 31–31, Berkeley, CA, USA, 2003. USENIX Association.