Exploring the Design Space of Metadata-Focused File Management Systems

Richard Watson
University of Southern Queensland
Toowoomba, Australia
rwatson@usq.edu.au

Stijn Dekeyser
University of Southern Queensland
Toowoomba, Australia
dekeyser@usq.edu.au

Nehad Albadri
University of Southern Queensland
Toowoomba, Australia
u1023136@usq.edu.au

ABSTRACT

Operating systems both old and new are reliant on the venerable hierarchical file system. For some time now, however, attempts have been made to either define new file systems or to bolt on applications that offer much improved functionality to attach and use metadata. This is because researchers have shown that traditional file systems are not able to meet users' needs in terms of organising large numbers of files effectively, and to support expeditious retrieval of those files when they are needed at a later time.

Numerous proposals for post-hierarchical file management systems have been described in the literature; researchers focus on different dimensions of such systems in order to solve or reduce identified limitations. In some cases this leads to significantly different file system architectures, while in other cases new functionality is added on top of a traditional system through special purpose user-space applications. Orthogonally, some proposals focus on tags while others favour named attribute-value pairs. Still other choices are, seemingly, made in an *ad hoc* and often implicit manner.

This paper investigates the different dimensions and associated choices that participate in the proposal of new approaches and that affect their ability to improve on current systems. The Cartesian product of those dimensions and options forms a large design space; we map some of the existing literature onto that design space and discuss approaches to evaluate new proposals.

Keywords

File systems, metadata, classification, tags, attributes.

1. INTRODUCTION

Users' computers systems, mobile and cloud-based as well as desktop-oriented, contain more files than ever before. Regardless of whether those files store scientific data generated by increasingly sophisticated instruments and computer models, work-related documents, or more personal collections of media and artefacts of our digital life [14, 21], the effect of this growth is that it is becoming progressively more difficult for users to manage their files. Informally the key problem can be stated thus: "*How can I store file X so that some time in the future I (or someone else) can easily find it?*"

ACSW '17, January 31-February 03, 2017, Geelong, Australia

DOI: http://dx.doi.org/10.1145/3014812.3014833

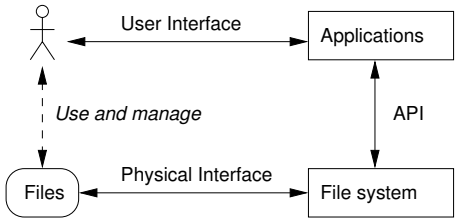


Figure 1: Interface relationships

The technique of locating files or other text-based objects by *searching* for keywords that have been extracted from file *content* has become widespread and may even be a dominant access mode for some users. However it has many problems such as computational cost, efficacy (ability to actually find the required file), inability to handle binary files, and (typically) dependence on relatively unsophisticated queries.

For these reasons we believe that the task of locating a required file should be be conducted in terms of file system level metadata associated with files rather than the file content. We assume that this file level metadata is deliberately assigned by the file creator and so will be potentially more useful than the keywords typically extracted by file indexing systems. (We note that user supplied file system level metadata may be augmented by metadata extracted automatically from file content; this would give access to some of the benefits of file content indexing while retaining especially the power, speed and universality of file system based metadata inspection operations.) The structure of file management system metadata, and the services provided to manipulate and use that metadata, becomes a key contributing factor to the efficacy of any file locating process.

We use the term "file management system" as a grouping of both specialised file systems as well as special-purpose applications, designed to improve on the metadata-related deficiencies of traditional hierarchical file systems. One such problem is the fundamental disconnect between a user's needs in managing and using their files and the decisions that file system designers face in developing a set of file system services. This disconnect arises because users interact with application programs which in turn rely on file system services—that is, there is an indirect relationship between user requirements and file system design. Figure 1 depicts this relationship.

The challenge for designers of systems that manage collections of files , which this paper addresses, is to determine a set of core ser-

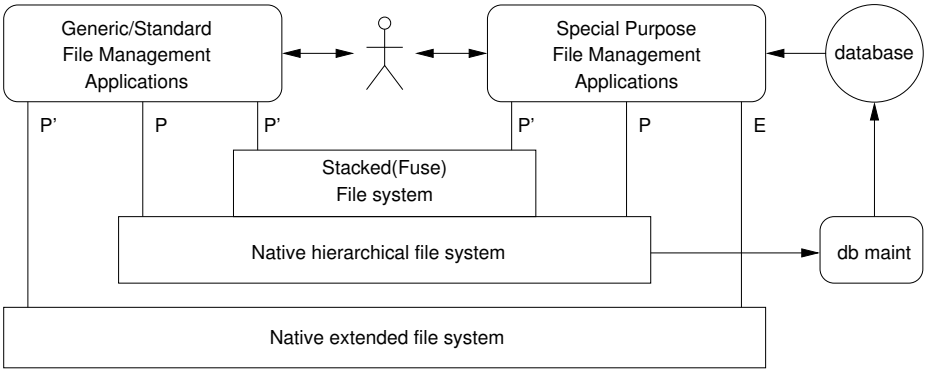


Figure 2: Implementation options for File Management Systems

vices, together with associated key physical design elements, that support users' requirements by enabling the creation of novel user interface applications that will advanced file management functionality. While user level applications are key to meeting user requirements, we do not address their design in detail, though do consider generic functionality at the user level.

We are mainly interested in services that support *user* generated metadata, as distinct from the system metadata such as file size and timestamps, and auto-generated metadata such as that provided by content indexing systems or by physical devices. This is based on the assumption that users will organise metadata to support their personal view of the files they manipulate rather than use either a generic file classification system, or one generated by an algorithm. Metadata that is not directly created or manipulated by users should naturally still be captured and utilised to support system functionality as well as user processes.

The motivation for this paper is built on the problems associated with traditional hierarchical file systems in terms of satisfactorily addressing file management and the task of locating a previously saved file. These are summarised in section 3, after first defining in section 2 what we mean by a 'file management system', and surveying some implementation and performance related issues. The main contributions of the paper lie in the exploration and evaluation of the design space for metadata structure and services within alternative file management system designs (sections 4 and 5). In section 6 we also map the numerous proposed alternatives, none of which have to date succeeded in supplanting the well-entrenched HFS model, to the identified design space. We conclude with a discussion in section 7.

2. IMPLEMENTING FILE MANAGEMENT SYSTEMS

Users manage files using application that in turn rely on a file system. We refer to this ecosystem as a *file management system* (FMS). Note that file management systems include and possibly extend the standard file system concept.

Applications fall into two broad categories. There are a set of well established generic file and directory management tools such as the `ls`, `mv`, and `rm` Unix command line tools or the equivalent GUI file manager and browser applications (OSX Finder, Windows Explorer). The ubiquitous file browser windows that pop up within domain specific applications like word processors and spreadsheets whenever an 'open' or 'save' operation is required also fall into this category. These venerable tools have been augmented more recently by special purpose applications, often related to file searching, such as OSX Spotlight or the simple but powerful Unix `locate` system. Typical of this group, both these applications make use of an application-specific database rather than file system data structures.

Adding new metadata-based functionality to a FMS can be achieved in a number of ways as illustrated in Figure 2. The diagram shows user applications of the two kinds just described communicating via an API with the underlying file system. The API annotations are described below. Current FMSs are shown in the diagram as interacting directly with a HFS using a standard (most likely POSIX-based [1]) API, denoted by the P annotation.

The following implementation strategies are identified.

1. Use the established strategy of adding new special purpose applications to address shortcomings of tradition FMSs.

2. Extend the HFS using a stacked implementation. Fuse [11] is a very common framework for this. Such a system must use the same (virtual) file system API, which means that all the standard generic applications are available 'for free'. However the *semantics* of the file system operations may be significantly different from the HFS-based POSIX standard. For instance an `opendir()` path specification could denote a query in some extended file system. For this reason we label the API as P'.

   New special purpose applications may still be needed in cases where the generic UI applications are unable to support the enhanced semantics of the extended file system.

3. A new native file system may be implemented as an alternative to a stacked file system. This would achieve some modest performance gains, but at the significant cost of having to provide the physical bit management and IO services that the HFS already provides in the stacked implementation. Such a system could provide a truly extended API (labeled E in figure 2) though it would only be available to special purpose applications that are targeted at that file system.

The stacked file system seems to be an ideal option for exploring and exploiting novel file organisation and management. This approach has modest development cost and performance penalty [22] and the ability to employ existing applications without any modification (if the operating system supports virtual file systems).

Although user space file systems have been shown to perform well compared to native in-kernel systems under heavy I/O loads they do suffer correspondingly higher overheads in the presence of

a high volume of metadata operations [22]. This would suggest that a native file system implementation, while more expensive to implement that a stacked file system, would be a clearly advantageous option following prototyping, evaluation and eventual acceptance of an advanced alternative design.

Writing special applications is a poor long term solution. Users are unable to take advantage of new features via their usual tools, and the applications are often OS-specific. This lack of universal applicability slows adoption and development of new standards.

The design considerations presented in this paper are independent of a particular implementation. It would be expected that performance would be better in a monolithic kernel-based system but we contend that performance of metadata operations is not the major issue here. Of much greater importance is providing a uniform (meta)data model and consistent user interfaces that encourage and empower users to organise their files in natural and useful ways to facilitate later retrieval.

3. HFS PROBLEMS

Though not the first authors nor the last ones to do so, Seltzer and Murphy in their aptly titled USENIX paper [26] have argued that "*hierarchical file systems are dead*". Supporting their assertion, our wider research project focusses on the limitations HFSs have in terms of capturing and utilising metadata.

Traditional file systems employ a model where files reside in a tree of directories. As such, HFSs support the creation of a user-defined classification system. Classification is a natural human activity that seeks to manage and understand complexity by recursively grouping classes of entities (e.g. files or plants that share common properties) into subclasses. As the classification tree is descended, associated entities have more inherited properties, and the number of members of the subclass decreases.

In the file system instance, the searcher iteratively descends the directory (classification) tree, at each step choosing one directory from the children of the current directory node based on its name (which should reflect its categorical relationship with its parent and siblings). Each step reduces the search space until a relatively small selection of files is presented for selection.

The basic support provided by HFSs to organise files in directories and facilitating iterative, navigational search is one reason for their longevity. However, while simple hierarchical directories may have been sufficient in the past, ever-growing collections of files mean that HFSs are not able to meet modern users' needs in terms of organising and retrieving information. In previous work [2] we detailed HFS problems in this context. The following is a summary of these issues.

1. *Artificial hierarchies*

   Generally, the properties of files do not shape a natural subclass relationship, resulting in artificially constructed hierarchies. Consider files associated with university courses that have properties 'course code' and 'year of offer'. Either of the hierarchies shown in Figure 3, year-of-offer and then course-code within year or vice versa, could justifiably be used to group course files.

2. *Single Classification*

   In a hierarchy, items can often belong to more than one subtree. Assume that the hypothetical course files introduced above are organised by year then course. Now imagine that a file should be included in both courses: in which directory should this file be placed? We can either select one directory to place the file (Figure 4 (a) and (b)), keep duplicated copies

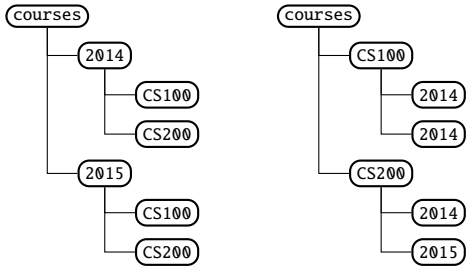


Figure 3: Alternative classification hierarchies

   (Figure 4 (c)), or keep one copy and place a hard or soft link to it in the sibling course directory. None of these solutions are practical and efficient (more details can be found in [2]).

3. *Problematic Pruning*

   This problem is a consequence of the single classification problem where orienteering through an imperfect classification hierarchy leads to users not finding files they are looking for. So if a user branches the 'wrong' way while navigating through the directory tree they will never find the file.

4. *Metadata management*

   In HFSs, bulk updates of metadata are inefficient. For example, if a user wishes to add or remove metadata, usually because the classification needs to be modified to better reflect reality, it often requires a sequence of non- trivial directory create, delete, and rename operations which must be carried out in the correct order.

5. *Native query support*

   The traditional file system API (e.g. POSIX) has very limited query ability: either to find a single file given a path (e.g. stat, open) or to open and read the contents of a single directory (opendir, readdir, scandir). This limited query capability supports an orienteering style of search, but does not support file system wide queries of the kind that are provided by the special-purpose applications that are layered on top of the file system.

   We argue that a generic, powerful query mechanism will assist users in understanding the existing organisation of their file system instance and help identify (and hence help rectify) occurrences of incorrect classification.

6. *Unconstrained directory membership*

   A consistent classification scheme is needed to ensure that the process of locating a file by orienteering through a file system directory tree will succeed. Inconsistently classified files will never be found. While HFSs provide a mechanism to build classification organisations for files using hierarchical directories, to ensure a consistent classification scheme all of the files in each directory must have a common subset of properties that is disjoint from the properties of files in sibling directories. These propertes are usually indicated or implied by the appropriate choice of directory name. (The directory name CS100 indicates all files in that tree are associated with the course CS100.) However the HFS has no means of formally defining directory semantics–properties that all resident files should posess–much less a mechanism that requires that all files added to a directory have such properties.

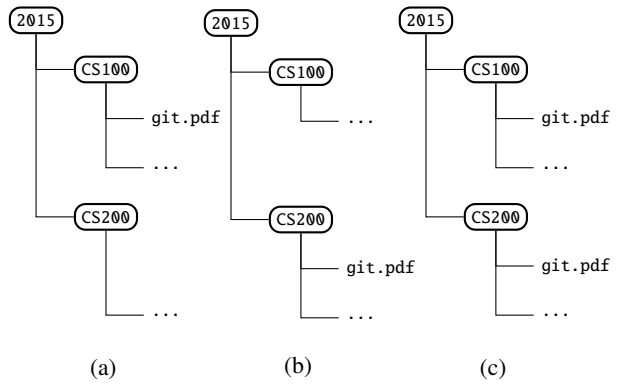


Figure 4: Multiple classification choices

4. REQUIREMENTS FOR METADATA FILE SERVICES

Imagine we are designing a new file management system—we have a completely blank slate—what file system services, exposed by the API, best support users' needs in storing and re-finding files? In this section we venture to identify a minimal but essential set of generic or abstract services that either create, update, or use file system metadata. These services are independent of specific metadata structures or implementation constraints; implementors must make design choices that provide this key functionality.

A few metadata services seem to be axiomatic. They allow a user to create, access, and manage a single file. Operations maintain the property that every file has a unique set of metadata, so it is always possible to lookup a file.

Two further groups of services can be identified: query-style read operations that return a group of files, and advanced metadata manipulation operations. The need for both of these groups is justified by the need to *classify* files in order to effectively manage file metadata and to later retrieve files.

Classifying things into categories that share common properties is a fundamental human activity [27, 13] that supports us in understanding the complex world in which we live. Highly refined classification systems exist in many domains such as the taxonomies of the biological sciences and the cataloguing systems of library and information science. These systems are usually hierarchical where a class of entities is recursively subdivided into disjoint sub-classes based on the values of attributes.

Organising files (or books or birds) into classes is incredibly useful. When adding a file to a class-aware file system instance, the class structure can provide a guide to what metadata to assign to that file. Specifically, an operation that adds a file to a class will automatically assign all metadata that is common to members of the class–the user only needs to identify the class and then add a small amount of unique metadata to distinguish the file from other class members. When searching for an individual file, class characteristics (i.e. class metadata) can guide the search process.

Traditional taxonomies are single-classification systems[1]; if an entity belongs to two distinct classes c_1 and c_2 then either $c_1 \subset c_2$ or $c_2 \subset c_1$. This property is handy when building a library classification system for physical books as a book can only reside at a single shelf location, but is limiting because in general many entities, especially files, have properties that violate this constraint; they may belong in two classes that are not in the ancestor relation. The problems that arise from single classification file systems are explored in detail in [2] together with the solutions offered by multi-classification systems.

We assume that the file search process will support observed user patterns, particularly but not exclusively the orienteering approach [28]. Orienteering as applied to existing file systems is an incremental approach where the user navigates through class-like directories at each step reducing (if step is downward) or redefining (if step is sideways) the set of candidate files. The traditional directory is not a class because traditional file services expose only the contents of a directory not the results of recursively traversing subdirectories.

Required Services.

From the above discussion we posit that a metadata-focused file management system must support the following services:

1. *Create* a file together with associated metadata.
2. *Identifying files*
   (a) *Lookup* a single file using a unique combination of metadata values. This is usually not exposed directly, but used by any service that needs to access an existing file (say to access or update metadata, or to delete the file).
   (b) *Query* the file system to return a group of files that meet class membership conditions. The query language must support relatively complex conditions such as union and intersection of classes.
3. *Modifying files' metadata*
   (a) *Update* an item of metadata for a single file, for instance to change a name or a tag.
   (b) *Reorganise* a selected group of files by systematically applying possibly complex changes to those files' metadata, thereby potentially reclassifying the files.

A query is closely related to the virtual directory concept [12]. There are many issues and design choices associated with virtual directories, which we defer to the following section as we consider them to be outside the 'minimal' set of file system services.

The reorganise service is required because of the dynamic nature of file system classification structures. Unlike scientific taxonomies and library classification systems that are standardised and relatively static, file system classifications are defined by individuals or groups of users and are typically unique to the user or domain, and are constantly evolving to suit organisational needs. As logical classifications change, potentially large groups of files are to 'move' between classes, necessitating the existence of operations that do so efficiently. It is possible that some updates requested by this service may fail to meet the metadata uniqueness constraint. Hence the reorganise service is a transaction—either all updates succeed or none do.

5. DESIGN SPACE DIMENSIONS

Suppose a developer wants to create a new, metadata-focused file management system that meets the service requirements defined in the previous section. Such a developer will need to make a number of key decisions based on overall goals to be met. For the moment we discount other important considerations such as performance and efficiency.

[1]We note that faceted library classifications systems have been developed that do support multi-classifaction. Ranganathan's colon classifaction system [23] is widely used in Indian libraries.

In this section we list and briefly discuss a range of dimensions and associated choices that our developer will need to consider. While some of these we have diligently derived (as they are not always explicitly stated) from existing research proposals as well as proof of concept implementations, others are newly identified and hence constitute one of this paper's contributions together with the careful enumeration and analysis of them all in one place.

Note that we consider only metadata that is managed by the file system, as opposed to metadata that is present within files (e.g. EXIF embedded in JPG images).

5.1 File Metadata

- *Kind of metadata*

  Two possible kinds of metadata exist: tags and name-value pairs. Tags are untyped and may be represented as atomic, unlimited length strings. They can be seen as a generalisation of the file (and directory) names in current HFS implementations. Attributes, on the other hand, have names that convey meaning, and values that may be typed. For example, *duration* would expect values of type *time*.

- *Cardinality*

  For both tags and attribute systems, the cardinality of values could be either single or multi-valued. Normally a tagging system will allow multiple tags, lest it defaults to current file (and directory) names. In contrast, for simplicity attributed systems may choose to impose single values for each attribute. Note that a multi-valued attribute system can trivially subsume a tagging system by providing a single multi-valued "tags" attribute.

- *Names and values*

  Are the values for metadata and the names of attributes managed by the file system? For instance should a tag or attribute name be defined before it can be used, and should the file system support a set of builtin types such as integers and dates that can be associated with an attribute? Requiring predefined values can reduce incorrect categorisation, and is relatively straightforward to implement, but adds a potentially significant burden to users.

- *Relationships*

  Does the file system support the many-to-many association of files to other files or objects in the system? Does it allow any pair thus associated to additionally have properties? For example, can we link movie files with music files, and record at what time point in the movie the music starts? Such functionality is distinct from ordinary group containment (see below) because of the many-to-many requirement.

5.2 Grouping Files

- *Existence of Groups*

  Should the file system implement file container objects, similar to the familiar directory? If not, then files live in a flat structure and can be distinguished only by their own metadata values. If files themselves have a sufficiently rich collection of associated metadata (rather than metadata associated with groups) then classifying using views (§5.3) and orienteering-style searching using iteratively refined queries is feasible, and hence groups may not be needed.

- *Group Structure*

  If groups can be created, can they be nested? This can support the classical hierarchical directory concept. Alternatively, groups could be organised in a directed acyclic graph which supports nicely the multi-classification file organisation.

  We observe that file system designers have in the past eschewed graphs but provided a symbolic link to simulate a graph structure. The unidirectional link is problematic to maintain; we believe that it should be possible to implement safe and efficient true graph structures.

  Support for groups admits the possibility of creating a generic, predefined hierarchy of groups that may be invaluable to allow a number of users to share a classification system.

- *Group Metadata*

  Groups themselves have identifying metadata; some of the same considerations as with files will be relevant here: are they tags or attributes, and single or multi-valued? To provide end-users with consistency, it is likely that the choices made for files will be duplicated for groups; however, that is not a necessity.

  Importantly, does the metadata associated with groups also apply to their members files? In other words, do files explicitly or implicitly inherit properties from the group they belong to? This question becomes more complex when considering recursive membership.

- *Membership*

  Do files belong to exactly one group, or potentially to many? On one level, the question here is whether a file in a directory is a member of its direct parent group only, or whether the system also considers it to be a member of that group's ancestor groups. In other words, if a user wants to see all file members for a group, is that membership calculated recursively, or not? The point is important given the requirement of supporting classification; in traditional (e.g. biological) classes, membership is recursive. In contrast, that concept does not carry over well to existing HFS implementations.

  On a second level, the question refers to whether or not membership of groups anywhere in the hierarchy (or in the directed acyclic graph) is disjoint or potentially overlapping. Seen from the perspective of an individual file, the question is whether that file can be a member of more than one, non-hierarchically related group.

  Plurality of membership is but one way to provide multi-classification in a file system; a combination of specific choices for some of the previous (and following) dimensions may also provide that desirable functionality.

5.3 Views

- *Existence*

  In section 4 we have argued that querying is essential for a metadata-focused file system. However, is such querying to be performed on a purely *ad hoc* basis, or can queries be "saved" to essentially become views? Do stored views have names, or only a query definition?

- *Structure*

  Can users organise stored views in a structured form, or is a hierarchy of views derived automatically based on the files

that it contains? If the latter, the expense for a system to automatically derive and maintain such a hierarchy will depend strongly on the power of the query language; in some cases it will be sufficient to check only the query definition, in others a large part of the file system instance may need to be checked periodically.

To what extent can the file system treat views as groups (referred to as virtual directories in some systems)? Group-like behaviour supports user level applications that enable search by orienteering, but has significant consequences if views are to be updatable (see below).

- Dynamic *vs* Materialised

  Are views run dynamically when a user accesses them, or is the file membership of views materialised? The latter can provide substantial efficiencies at query time, but will incur a 'view maintenance' cost when files are added, removed or updated in the underlying file system.

- *Updatability*

  A critical service required for metadata-focused file systems is *reorganising* an existing classification. A highly powerful approach is to allow views to be updatable: a set of files "moved" to a view automatically obtain the metadata necessary for the files to be in the result of the view's query [10]. Such functionality, while exceedingly powerful, will be necessarily limited in scope, as there will be cases where the view is theoretically not updatable, and there will be other cases where the view may be updatable, but the files moved into it cannot obtain the required metadata (e.g. when that requires a change in value of a system-defined attribute such as *file type*).

5.4 Behaviour (Event Handling)

In addition to structural issues, some researchers [10, 12] have proposed *active* components; the file system may run code when an event is fired (e.g. file creation or update). In other words, behaviour is associated with files and or groups.

- *File Transducers*

  A transducer is a program that extracts metadata from files when they are created or modified. A keyword indexer is one example. A key question is whether or not transducers are associated with file types, or with files in general.

- *Group Transducers*

  Does the file system allow users to place files in any group they choose, or are there membership conditions? For example, in a tagging system that has a group tagged as "Movies": can a user drop a text file in it? One could take the position that a tag is not a strict property and the user should be allowed to perform the operation. In that view, the tag "Movies" has different semantics than an attribute file-type with value "avi". In general, a group transducer can enforce constraints on the files that are members of the group.

- *View Transducers*

  This is similar to a group transducer and manages the view updatability process described above.

- *Relationship Transducers*

  Analogous to the other transducers, such a program would for example be able to set properties of the relationship, or enforce conditions on its members.

- *Object Oriented Classes*

  The discussion of behaviour linked with file types, groups, and relationships, leads to the question of whether the file system should become an object store [24], and the objects are instantiations of object-oriented classes while transducers are class methods. Even discounting behaviour, a combination of choices for the other dimensions described above may imply that a given file system architecture may be more accurately described as an object store. Groups and Views can then be modelled as classes.

5.5 Other dimensions

While some are only tangentially related to metadata issues, there are other considerations in terms of design choices that are still distinct from important objective requirements such as efficiency.

- *Add-on vs API integration*

  Some of the services identified in section 4 are implemented in some solutions as add-ons above a traditional HFS. As argued in section 2, this has substantial advantages (e.g. cost, file I/O and storage efficiency) but also important drawbacks. Most importantly, add-ons do not make the new functionality uniformly and consistently available to other user-space applications. This means that the effort of managing metadata and creating classifications may not deliver the expected payback, thus reducing the time users may choose to invest in the add-on.

- *Backward Compatibility*

  POSIX-type file systems are so entrenched in computing that users may only be willing to take on new systems that are backward compatible. Systems that are very different and perhaps much more powerful will not necessarily be adopted if they require users to undergo a paradigm shift.

- *Interoperability*

  Any new system will need to coexist with traditional HFS implementations. This requires an ability to import and export files, while maintaining as much metadata as possible. That may mean that a tool is needed to semantically map concepts from one model to another; such tools could for instance make use of RDF [18].

- *Access Control*

  The presence of richer metadata structures and even executable objects as part of a file system raises questions of access control [15]. For instance, who can edit file system metadata values? Is there a distinction between (a) system-controlled, (b) automated/indexed, and (c) user-defined metadata? How should file *content* permissions relate to file *metadata* permissions? Can users install their own event handlers to customise system behaviour? Can users define attribute names and value types?

  Prohibiting users from modifying file metadata need not violate the requirement of allowing user-created classification; this could be accomplished by supporting suitably expressive groups or views.

5.6 Examples

At this point we note that the number of dimensions listed above, with the number of choices for each of them, creates a multidimensional design space that can accommodate a multitude of possible file system architectures.

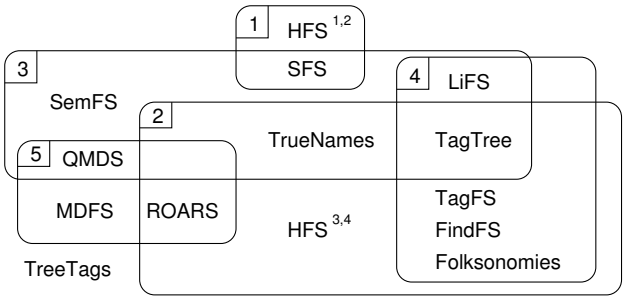
Structure	Attributes				Membership
	1 (anonymous)		N (named)		
	Values		Values		
	1 (name)	N (tags)	1	N	
Flat		Folksonomy[29]			1 (disjoint)
Tree	HFS[1]	TreeTags[2], TagTree[30]			1 (disjoint)
	HFS[2]	FindFS[9]	SFS[12]	LiFS[4], SemFS[16]	N
Graph				MDFS[10]	1 (disjoint)
	HFS[3]	TagFS[25]	Truenames[20], HFS[4]	QMDS[5], ROARS[8]	N

Kinds of Hierarchical File System
[1] without file or directory links
[2] with file links
[3] with file links and directory links
[4] with file links, directory links and extended attributes

Table 1: Design space populated with existing systems and proposals.

As one relatively extreme but nevertheless didactic example, consider a file system that supports files having attribute-value paired, multi-valued metadata that is either system-generated (e.g. *timestamp, file type, user, ...*) or automatically obtained through an indexing file transducer (e.g. *keywords, document structure, ...*). Suppose that it does not allow end users to provide other metadata for the files and also does not allow users to create groups. In other words, the transducer is essentially sophisticated enough to generate a set of name-replacing metadata that delivers unique as well as meaningful file identifiability. The file system does allow creating materialised views, however, that are defined through conjunctive queries. Those materialised views are stored as user-defined queries (which supports user-initiated classification) and may have a user-allocated name to provide an additional method for finding files. The views form a hierarchy that is derived automatically through query containment checking (users are unable to change the hierarchy of the views). Given the limited power of the query language, the algorithm need only check the query definitions at the time the view is created, not the file system instance which may change rapidly.

Clearly such a system is significantly different from existing systems, and it is also distinct from other research proposals that have made alternate choices for the various dimensions. Its efficacy in terms of usefulness, efficiency, easy of use, etc would need to be determined; we will explore approaches for doing so in section 7.



Groups:
1 Single classification
2 Multi-classification, no query support
3 Metadata managed by system
4 No ability for multi-file metadata updates
5 Non-traditional data management systems

Figure 5: Classifying systems

6. MAPPING EXISTING WORK

An interesting exercise made possible by the identification of the various design space dimensions and their possible values, is the categorisation of existing systems and research proposals in the multidimensional space created by these dimensions.

For example, the traditional hierarchical file system can be easily located in the design space. Proposals that have attempted to replace hierarchical directory structures altogether [3, 5, 7, 10, 12, 17, 19, 24, 26], in favour of relying on rich collections of metadata, can also be mapped in terms of their particular choices.

Similarly we can locate approaches that have built on existing HFSs by either improving navigation to a directory [6], enhancing use of symbolic links [9], or utilising tags while explicitly introducing a query language [2].

For clarity and to reduce the sparseness of the multidimensional space when populated with citable systems, we focus on a subset of dimensions, those that relate to file metadata and group structure, and project away the others. The result is Table 1. There are four entries for HFS that reflect particularly the presence of linking features. The presence of links between directories forms a graph structure but the common symbolic link implementation (unlike the hard link) is easily broken and such implementations do not reliably support a graph structure.

The table shows group parameters vertically and file metadata parameters horizontally. The group structure values can be flat, tree, or graph, and the membership column lists the number of groups to which a file may belong (which must be one in the case of a flat, ungrouped structure). Attributes can be either tags (anonymous) or named attributes. Either can be single or multi-valued.

The table discounts, among others, the "relationships" dimension; however, several proposals (LIFS [3, 4], MDFS [10], and QMDS [5]) are focused precisely on that dimension.

We place existing proposals in the cells of the table shown; some cells contain more than one system—these systems are still quite distinct even though they seemingly occupy the same spot in the design space. This is an artefact of projecting only some dimensions to improve clarity. For example, TreeTags [2] is functionally distinct from TagTree [30] although their names indicate common design choices.

It is clear that there has been significant attention paid to tagging (table column two) and to graph-based variants (bottom right). The upper right quadrant of the table, that combines structural simplicity with attributed data, is interestingly empty; in the example given in section 5.6 we have outlined a possible system to fill the gap.

It is instructive to review the systems shown in Table 1 with respect to the problems identified (section 3) and required metadata

functionality (section 4). In Figure 5 we have grouped the systems into intersecting sets according to some key requirements: mainly their ability to form classification structures, and their metadata management abilities.

Interestingly, the overwhelming majority (12/14) of systems support multi-classification, which confirms the relevance of this requirement. However, about half of the multi-classification file systems lack an inbuilt query system that facilitates advanced use of such classifications. Note that multi-classification systems *with* query support appear on the diagram as systems not included in sets one or two.

In some systems (group three) metadata is managed to some extent by the file system, compared to systems where metadata manipulation is under complete user control. The fourth group contains systems that are only able to update information for one file at a time; that is, all metadata update operations are file-based (rather than updates at directory or collection level).

Most systems are based on a familiar underlying file system model. A few, members of the fifth group, use novel non-traditional approaches; some of these may be more correctly seen as data or information management systems. One system (TreTags) fits none of these groupings as it is a multi-classification system that is based on a HFS model but also supports queries and multi file update of user managed metada.

There is no strong correspondance between the classification seen in the data model design space (Table 1) and the functionality-based classification of Figure 5. We do see that most of the tag-based systems (second column of table) belong to the intersection of the multi-classification set and the file-at-a-time metadata update set. So while popular tagging systems have the power to support multiple classification schemes, they lack the metadata management operations to facilitate easy re-classification. Not surprisingly, the final 'advanced' group lives in the bottom right part of the design space as these systems have the richest matadata structures.

7. DISCUSSION

Numerous design options were identified in section 5. Implementation of a new file management system design requires choice between design alternatives, and selection of optional design elements. We propose some high level criteria that can be used to guide these choices. We identify some key invariant properties that a file management system must possess, as well as some desirable criteria. The influence of these criteria on design choice is discussed.

The fundamental services identified in section 4 are justified by users' need to build a file management system instance that classifies files to support later retrieval. We believe that multi-classification support is an invariant property of any advanced file management system. The second fundamental design criterion is simplicity: no system should be more complex than needed to meet requirements. This second criterion supports users, as well as system implementors, in building simple and clear mental models of the system.

Support for multi-classification, advanced query, and powerful file metadata reorganisation requires the following design elements. Structurally, either a graph of groups with a single name, or a hierarchy of groups with multiple tags supports multiple classification. An alternative is a flat groupless organisation where files have multiple attributed metadata. (The simpler flat tag organisation or single-named group hierarchy are insufficient). If the flat attributed metadata model is adopted, then the system would have to support views so that classification structures can be described (using a query) and used in API operations.

As already noted, there is no evidence of the application of a *purely* flat model, although both MDFS [10] and WinFS [24] subsume such a model. This is perhaps because of the complexity of managing views/virtual directories that would be mandatory in supporting file classification; the cited systems arguably suffered from precisely that complexity. Enhanced HFS models that meet the requirements defined in section 4 can deliver better file classification without the added complexity of views. However the flat approach should not be discounted. Notwithstanding the implementation challenges, it offers one huge advantage: the file classification structure is defined as a set of queries that are much more easily and cheaply updated than physical directory membership; this facilitates large scale reorganisation.

We therefore put forward two conjectures: Firstly, a system where most metadata is allocated to richly structured groups (not files) is equivalent, in terms of classification ability, to a flat system where files themselves are associated with multiple attributed metadata and users can create views. Secondly, the latter system (while complex to implement) is superior to the former in terms of reorganisation capability.

There are other desirable evaluation criteria, but their importance is less universal and perhaps driven more by designers' personal views, experience and beliefs or by consideration of the characteristics of the users and their file system usage patterns.

For instance, it may be argued that a user should be prohibited from classifying a file incorrectly (equivalent to placing a file in the incorrect directory by mistake–see section 3, problem 6). This could be supported by suitable event handlers, though at a significant increase in complexity both of implementation and configuration. It probably also dictates file system management of metadata names and values (see section 5.1).

Another possibly desirable criterion is the support for defining classification data structures that are independent of the presence of files. The traditional directory hierarchy is such an example. If a flat non-grouped structure is adopted the classifications can be defined using views, but this requires the existence of a file system maintained dictionary of attribute names with which to construct valid view definitions.

Integrating file system generated (e.g. timestamps), user assigned, and third party (e.g. content indexes) metadata such that it can be queried and managed uniformly is clearly desirable, but its importance relative to the complexity of the task is unclear.

Backward compatibility, the ability of existing code to execute unchanged when using a new file system, is clearly an important criterion, and most designs should be able to achieve this by serialisation of the equivalent "file path" concept. However advanced file system APIs will include new functionality such that applications written to use this capability will no longer be compatible with (say) existing POSIX compliant file systems. This should not be seen as a limitation.

Advanced file system metadata functionality is most likely to be accessed through user applications (Figure 1) like file browsers and file managers as users organise and search for files. The absence of effective and sophisticated (especially graphical) applications would severely hamper the use of file system (API) functionality. Perhaps it behooves us as file system implementors to also create portable prototype versions of common Open/Save File dialog boxes that bring enhanced user interface functionality to all user applications. These could be implemented in the major OS and GUI toolkit combinations.

8. CONCLUSION AND FUTURE WORK

This paper examines design options for the implementation of metadata structures in post-hierarchical file systems. To place these

in context, and to aid in their evaluation, we have provided an overview of implementation methods and described key limitations of hierarchical file systems.

The main work in this paper is to detail the dimensions of the design space and determine a set of core metadata file management system services. In addition, we have mapped existing research into the design space. Two overall themes have emerged in this work. First and foremost is the vital importance of classification schemes in every aspect of file system metadata definition and management. The other relates to the disconnect between file systems UIs and APIs; enhanced functionality available from a beautifully designed new file system is useless unless it can be exploited by users. This is an implementation problem: how can advanced file system functionality, exposed via its API, be exploited by users who interact with higher level and possibly generic user interface programs? Another theme additionally emerges, concerning the extent to which a file system either restricts or aids user actions to improve classification; managing attribute name spaces and event handling fall into this class of functionality.

There are different directions that can complete and extend the current work. For example, investigation of unexplored areas of the design space could discover potentially useful designs. The proposed flat attributed model with views, introduced in section 5.6, is one such possibility.

Another direction for future work involves developing formal criteria by which the utility of different metadata file system designs can be evaluated. This could be used either directly to evaluate a design, or practically to inform the creation of a "metadata benchmark" test suite that could be used to measure the performance of file system prototypes on key metadata operations.

Finally the two conjectures mentioned in the section 7 could be investigated; this would yield insight into the relative merits of structured and unstructured metadata models.

9. REFERENCES

[1] IEEE standard for information technology - Portable Operating System Interface (POSIX) base definitions. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6.*, 2004.

[2] N. Albadri, R. Watson, and S. Dekeyser. TreeTags: bringing tags to the hierarchical file system. In *Proceedings of the Australasian Computer Science Week Multiconference, Canberra, Australia, February 2-5*, page 21, 2016.

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

[4] 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*, 2006.

[5] S. Ames, M. Gokhale, and C. Maltzahn. QMDS: a file system metadata management service supporting a graph data model-based query language. *International Journal of Parallel, Emergent and Distributed Systems*, pages 159–183, 2013.

[6] J. Amoson and T. Lundqvist. A light-weight non-hierarchical file system navigation extension. In *Seventh International Workshop on Plan 9, IWP9. November 14th–16th, Bell Labs Ireland*, pages 11–13, 2012.

[7] D. Barreau and B. A. Nardi. Finding and reminding: file organization from the desktop. *ACM SigChi Bulletin*, 27(3):39–43, 1995.

[8] H. Bui, P. Bui, P. Flynn, and D. Thain. Roars: A scalable repository for data intensive scientific computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 766–775, New York, NY, USA, 2010. ACM.

[9] J. Chou. FindFS: adding tag-based views to a hierarchical filesystem. Master's thesis, University of British Columbia, 2015.

[10] S. Dekeyser, R. Watson, and L. Motrøn. A model, schema, and interface for metadata file systems. In *Proceedings of the thirty-first Australasian conference on Computer science-Volume 74*, pages 17–26. Australian Computer Society, Inc., 2008.

[11] fuse. http://fuse.sourceforge.net/, 2014. [Online; accessed April-2014].

[12] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 16–25, New York, NY, USA, 1991. ACM.

[13] R. J. Glushko, editor. *The Discipline of Organising*. MIT Press, Cambridge, MA, 2013.

[14] P. Lyman. How much information? http://www. sims. berkeley. edu/research/projects/how-much-info-2003/, 2003.

[15] S. A. Moffatt. Access control in semantic information systems. Master's thesis, University of Southern Queensland, 2010.

[16] P. Mohan, V. S. Raghuraman, and A. Siromoney. Semantic file retrieval in file systems using virtual directories. In *the Poster Session of the 13th Annual IEEE International Conference on High Performance Computing (HiPC), Bangalore, India*, 2006.

[17] B.-H. Ngo, C. Bac, and F. Silber-Chaussumier. Integrating ontology into semantic file systems. In *Huitièmes Journées Doctorales en Informatique et Réseaux (JDIR'07)*, pages 139–142, 2007.

[18] L. Octalina. Interoperability of metadata in file systems. Master's thesis, University of Southern Queensland, 2008.

[19] Y. Padioleau, B. Sigonneau, and O. Ridoux. LISFS: A logical information system as a file system. In *Proceedings of the 28th international conference on Software Engineering*, pages 803–806. ACM, 2006.

[20] A. Parker-Wood, D. D. E. Long, E. Miller, P. Rigaux, and A. Isaacson. A file by any other name: Managing file names with metadata. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 3:1–3:11, New York, NY, USA, 2014. ACM.

[21] Z. P. Perišić. Too much information. *Review of the National Center for Digitization*, pages 68–70, 2007.

[22] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[23] S. R. Ranganathan. *Colon Classification*. Ess Ess Publications, 6th. edition, 2006.

[24] T. Rizzo. WinFS 101: Introducing the new windows file system. http://archive.today/ZfniG, 2004. [Online; accessed February-2014].

[25] S. Schenk, O. Görlitz, and S. Staab. TagFS: Bringing

semantic metadata to the filesystem. In *Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06)*, 2006.

[26] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*. USENIX Association, 2009.

[27] A. G. Taylor and D. N. Joudrey. *The organisation of information*. Libraries Unlimited, Westport, CT, 3rd. edition, 2009.

[28] J. Teevan, C. Alvarado, M. Ackerman, and D. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004.

[29] C. Trattner, C. Körner, and D. Helic. Enhancing the navigability of social tagging systems with tag taxonomies. In *Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*, i-KNOW '11, pages 18:1–18:8, New York, NY, USA, 2011. ACM.

[30] K. Voit, K. Andrews, and W. Slany. TagTree: Storing and re-finding files using tags. In *Information Quality in e-Health*. Springer, 2011.