A Novel File System Supporting Rich File Classification

Nehad Albadri[1,2] , Stijn Dekeyser[2]
[1]University of Thi-Qar , Iraq
[2]University of Southern Queensland, Australia.

nihadghasab.comp@utq.edu.iq, dekeyser@usq.edu.au
corresponding Author: Nehad Albadri

Abstract: Existing implementations of file systems often seem to be made on an ad hoc and implicit basis. This paper aims to enhance the organization of files and retrieval of files by modifying the traditional hierarchical file system to improve built-in query support and bulk metadata updates supported at the file system level. We introduce tags in a hierarchy of file collections and use links to allow file retrieval from multiple paths as files exist in multiple directories simultaneously. By using a series of modest changes to the hierarchical file system, we propose a novel Linked Tree Tags (LTTs) model. These changes include using multiple tags instead of names, collections instead of directories, exposing a query language at the Application Programming Interface (API) level, and allowing controlled file links. We assess our model's expressive capability and demonstrate that LTTs overcome traditional file systems' limits and provide users with the to manage their files easily.

Keywords: File System, classification, hierarchical file system, metadata capabilities, linked tree tags.

1. Introduction

We interact with our computers every day, storing, organizing, and retrieving files. These routine tasks are growing more difficult as the quantity of stored files increases, thanks in part to increased storage capacity and quick access to data-generating mobile applications [1]. We focus on the specific use case in which users save files in a file system to which they have access and then want to find and recover these files. Someone else created the file [2] [3], or programs are handling a set of files on behalf of a person. The most important things are that the user knows that the file they are looking for exists, that they remember some of the file's properties and that they have direct access to the file system.

Since the 1970s, hierarchical file systems (HFS) have been accountable for personal data management [4], even though the HFS is no longer adequate to successfully support the tasks that users have in managing their saved data [5]. It results from the number of individually [6] created or downloaded files increasing where users commonly cannot recall where those files are stored [7] and how they are titled, so support for successful searching is essential. It has been addressed by various researchers [8] and in our research in the area of the file systems and metadata (e.g. [9] that HFS is a single classification system where files can exist only in one specific directory within the hierarchy [10]. Formally, single classification indicates to if an entity belongs to two distinct classes c1 and c_2 then either $c_1 \subset c_2$ or $c_2 \subset c_1$. It can cause some problems when a user tries to build a hierarchy of files as directories that reveals file properties and supports perceptive search strategies [11]. The problems that arise from single-classification file systems are discussed in detail in this paper by providing multiple-classification solutions [12]. It is noted that to overcome or

alleviate the deficiencies of hierarchical (tree) file system structures, POSIX-compliant operating systems provide both symbolic and hard links [13]. Symbolic links are particular files that contain a reference to another file or directory. Whenever the destination specified by the link is moved [14], renamed, or deleted, the link is not updated; instead, it refers to something that no longer exists [15]. Other problems with links, including hard links, have been described in previous papers [16]. To summarise, the introduction of links does not sufficiently address classic file system single categorization difficulties [17]. Other methods for bypassing HFS restrictions have been described. Some of these recommendations were based on a thorough collection of file metadata rather than a hierarchical directory structure [18]. These recommendations are meant to replace the HFS. Another method for dealing with HFS constraints is to build additional functionality on top of the existing file system. Many comparable strategies have evolved [19], and in previous work [20], we grouped models and applications depending on the primary strategy they employ. However, each approach has limitations [21] that prohibit considering them as a solution to the HFS problems in this paper [22].

We seek to strike a compromise between the HFS replacement and HFS add-on methods in this study. We propose a minor modification to HFS semantics that adds tagging to the core file system structure while retaining the familiar and beneficial hierarchical container structure of traditional file systems [23][27][28]. Because of the similarity in behavior, introducing such a system as a replacement for a purely hierarchical file system would be simple, but more importantly, it would provide a uniform API that could be used to build richer generic user interfaces that could leverage the enhanced metadata structures to better support user file management activities [29][30]. The issues associated with traditional hierarchical file systems in terms of adequately solving file management and search [31][32] inspired this work. Section 2 evaluated the aforementioned challenges, and Section 3 defined the needed services and fundamental physical entities that would be employed in the suggested model. The key contribution of this study is the innovative model known as LTTs, which is discussed in Section 4. Section 5 reviews the suggested model in terms of addressing HFS constraints as well as compared to other solutions; we compare our proposed model to other prior models and demonstrate the differences and benefits of our proposed model.

2. Problems and Contributions

A. Problems

In the traditional hierarchical file system model, files are stored in directories. HFSs make it possible for individuals to create their classification schemes. 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) [24] into subclasses (figure 1). Associated entities inherit more properties as the classification tree descends, and there are fewer members of the subclass [25]. Iteratively, the searcher in a reasonably well-organized file system instance descends the directory (classification) tree [26], choosing one new directory from each child node of the current directory (which should reflect the categorical relationship it has with its parent and siblings). Each step reduces the search space until a relatively small selection of files is presented for selection. One of the reasons for their longevity is the support provided by HFSs to organize files in directories and facilitate iterative, navigational searches. Even though simple hierarchical directories may have been sufficient decades ago, today’s ever-growing amount of data means that HFSs

are unable to meet the organizational and retrieval needs of modern users. We discussed the problems of HFS in this context in our previous paper [2]. These issues are summarised in the following paragraphs.

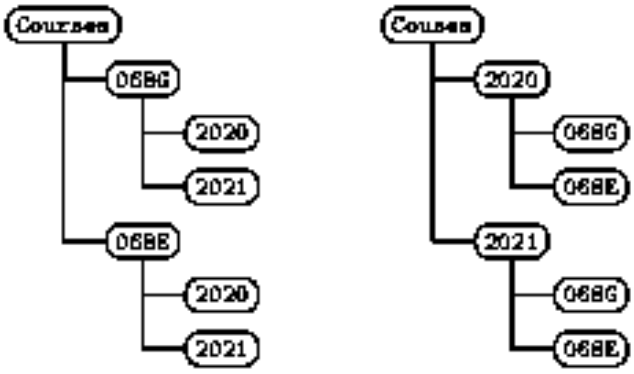

Figure 1. Alternative classification hierarchies

Problem 1: Artificial hierarchies

It is typical to find that file attributes do not naturally establish subclass connections, resulting in the formation of artificial hierarchies. Assume a user possesses files connected with university courses on his or her computer, and these files include at least two characteristics "course code" and "year of offer," although either of the hierarchies presented in Figure 1 can be used to organize course files. The first hierarchy includes the course code, followed by the year, or the second option on the year might be used to categorize course files.

Problem 2: Classification

It is common for items to belong to more than one sub-tree in a hierarchy. Suppose the hypothetical course files presented above are arranged by year, then by course. In that case, how would you place the file that is to be included in both of the courses? There are three options: place the files in one directory, keep duplicate copies (Figure 2), or keep one copy and place a hard or soft link to it in the sibling course directory. All of these solutions are neither practical nor efficient.

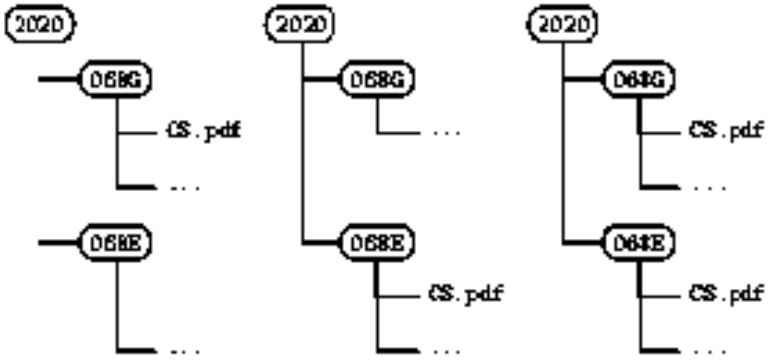

Figure 2. Multiple classification choices

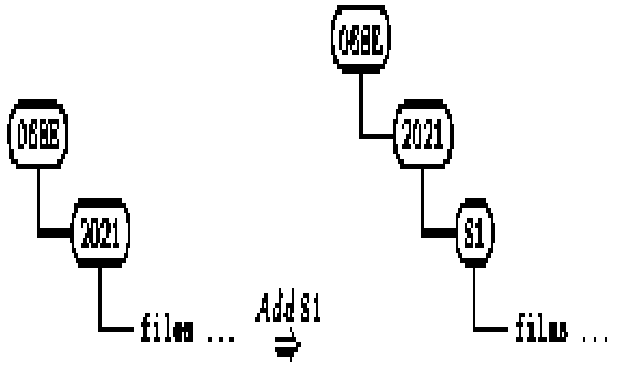

Figure 3. Adding metadata to a hierarchy

Problem 3: Problematic Pruning

Because of the classification problem (above), users are unable to find the files they are looking for when orienting themselves through an imperfect classification hierarchy. Therefore, in the search tree, a seeker will not be able to find the file if they orient themselves the wrong way.

Problem 4: Metadata management

IHFSs have inefficient bulk metadata updates (figure 3). For example, if a user plans to add or remove metadata, usually due to the need to update the classification, it usually involves a sequence of nontrivial directory creation, deletion, and renaming operations in the correct sequence.

Problem 5: Native query support

Conventional file system APIs (e.g., POSIX [15]) have limited query capabilities: you can find a single file by typing its name (e.g., stat, open) or open and read a single directory (opendir, readdir, scandir). Although this limited query capability supports an orienteering style of search, it does not support file system-wide queries such as those provided by the special purpose applications that are layered on top of the file system. Users would be able to identify the structure of their existing file system instance using a general, powerful query tool and therefore be helped spot errors in classification (and correct them).

B. Contributions

We have made the following significant contributions to this paper.

- Proposed LTTs would add to the traditional hierarchical file system by using tags and links as tools to deal with the issues that were mentioned above for the existing file system.
- They would also add more features that can be used by users and include a query language in the File System API level to make it easy to find files.

3. Framework

In the following section, we will describe the set of functions exposed at the programming level API of LTTs. From the description, we exclude user-level file system operations provided by applications that layer above the file system API. In addition, we will only present parts of the API that deal with the file management system's organization. Operations such as file content manipulation and file management system privileges and protections are not handled. We will compare our approach to other research proposals, including those that also use tags in Section 5. Our suggested system will provide three different types of services:

1. The process of creating a file and its related information. In addition to the ability to connect a file in several paths at the same time, without the hassles and constraints of current HFS linkages.
2. Identifying files
   - The ability to look up or find a single file using a metadata standard. It is necessary for a file's 'open' operation. The file system metadata must be regulated to ensure that each file has a unique metadata definition.
   - A query service that provides a collection of files that fulfill the class membership criteria. At the very least, this would imply revealing the contents of a file collection, such as a directory. We expand this concept, however, to a general file system query. These actions are crucial in the creation of file system user interfaces.
3. Modifying files' Metadata
   - Update an item of metadata for a single file, for instance to change a name, a tag or its link.
   - Reorganize a selected group of files by systematically applying possibly complex changes to those files' metadata, thereby potentially reclassifying the files.

Tags, files, linkages, and groups of files are the most fundamental physical elements in the proposed file system. These are specified here: along with the associated metadata-based route and query ideas.

A tag is a metadata item attached to a collection or a file. Conceptually it is a text string that is unlimited in length. To some extent, it is a generalization of the name that is associated with traditional HFS files and directories.

A file is a sequence of bits (or maybe a larger atomic data unit) that is stored in the file system. It has a unique system identifier. The logical organization of the file system is unrelated to file content. Files are simply represented within these structures by the system identifier. So, in the following, the word "file" can usually be interpreted as synonymous with "file identifier". A file may have associated tags.

A link can be defined as special entries that enable multiple references to files and thus, the use of multiple file and path names. The link in our proposal file system will be like the links in the current file systems but without the mentioned limitation (unidirectional) in Section 1. So, the proposed links are bidirectional as will be shown in Section 5.

A collection is a file container. From a logical view, it is an object that has some unique system identifier; each collection is associated with zero or more files (file identifiers).

A path is a sequence of collections such that each member is a child of the preceding collection. It is the route from the tree root to a collection, and so unambiguously identifies a collection. Every collection can be uniquely identified by a path. A file path is the combination of a (collection) path together with the identified file within that collection. File system users navigate [18] paths to find files.

A query specifies a search criterion in terms of collection and file metadata (tags). Evaluating a query returns a set of zero or more files that may reside in many different collections. The file system query is a key divergence from traditional HFS APIs. While the functionality offered by a file system query can be duplicated by a client program of a traditional HFS, it will likely suffer from poor efficiency due to the need for repeated file system calls. A lack of integration with other components of the file system will also make this approach less effective.

4. The LTTs Model

In conventional file systems, such as ext2 or NTFS, directories are organized hierarchically. However, the hierarchy is only determined by the implementation choices made. Directory structures are usually almost indistinguishable from regular files, at least on a low level. We take a different approach to formalizing the data model for our proposed system, where the hierarchy of collections is completely separate from the set of files associated with each collection. The critical aspect of these approaches is, however, their isomorphisms. We add links so the files can at the same time connect to more than one collection without any deficiencies as in the current file systems. We also use tags instead of names for files and directories in our system. During the data model section, we will expose the reason why. Our model differs from traditional file systems in that it includes a query language that is an integral (but separate) part of its API. The following is a formal description (in Z notation) of the initial data model and its associated operations.

A. Data Models

The collection is organized in a hierarchical (tree) manner. Collections can contain sub-collections, which are also known as sub-collections. The terms parent and child naturally describe the relationship. A collection may have associated tags and it may have links to any other collections. Every file in the collection is assumed to inherit this tag, as well as any tags associated with ancestor collections. All files in a collection have been placed there because they share some semantic properties (e.g. all these files are associated with a particular project). The files can be linked to other collections by bi-directional links. A file in a collection can be connected with a set of tags that are directly related to the file. The file inherits from the contained collection, and the file path is its parent. The main semantic distinction is that an atomic action that affects a collection tag broadcasts to all files in that collection, whereas a file tag affects only one file.

1.Hierarchy

$H : cid \nrightarrow cid \cup \{ \tau \}$ *where*

- *cid* is the type of collection identifiers. Note in the hierarchy, we use cid instead of tag. This is to prevent involving the hierarchy in some operations that are not needed such as changing the tags value, which does not need to change the hierarchy. In other words, if we consider *ctag*, the hierarchy will be changed with every operation.
- H describes a tree of collection identifiers with root τ.
- H (*s*) is the parent of *s*.
- Initial value : H=∅.
- Constraint: $\forall s \in dom\, H \bullet (s, \tau) \in H^+$
  All collection identifiers are part of a single rooted tree. This constraint also precludes cycles.

2. Collection tags

$S : cid \cup \{ \tau \} \nrightarrow \mathbb{P} ctag$

where

- ctag is the type of collection tags. By allowing to have more that one attached tag with collection, this might provide many options (flexibility) to locate the collection.
- The collection tag for τ is the distinguished value root.

- Initial value: $S=\{ \tau \mapsto root \}$
- Constraint:
  $\forall (i,p), (j,q) \in H \bullet p=q \wedge i \neq j \Leftrightarrow S(i) \neq S(j)$

  The collection tags of collection identifiers with the same parent must be distinct; collection tags are unique within collections.

3. Files

$F : cid \nrightarrow (id \nrightarrow \mathbb{P}\, ftag)$

- Files are grouped in collections; each collection is identified by a collection identifier.
- Each file is a bidirectional mapping between file tag (type ftag) and a physical identifier (type id).
- Initial value: F=∅
- $\exists (s_1,f_1),(s_2,f_2) \in F \bullet s_1 \neq s_2 \wedge dom\ f_1 \cap dom\ f_2 \neq \emptyset$.

A physical file may exist in more than one collection by file links.

4. Collection path

The following two functions are derived from H and S.

The path function $D : cid \nrightarrow seq\ cid$.

The path function $P : cid \nrightarrow seq\ \mathbb{P}\, ctag$ is now a sequence of tag sets.

$P(s) = \{n:\mathbb{N}; id : cid \mid (n,id) \in D(s) \bullet (n,S(id))\}$

The formal definition for P describes a collection path as a sequence of sets of tags. Figure4 shows abstract syntax for paths where multiple tags exist for both collections and files. Note that, while P defines paths that include all tags associated with collections, in any given instance of a file system it may be possible that a path specification with fewer tags per object will still identify a single file. That is, a single object may be identified by more than one path specification.

Using fully populated path specifications can be useful however: if an object has been created with fully specified path p, and if the tags associated with the objects in p are not subsequently modified, that file can always be located (opened) using path *p*.

$$
\begin{array}{lcl}
CollPath & ::= & ctags/ \mid Path\ ctags/ \\
ctags & ::= & ctag \mid ctags \wedge ctag \\
FilePath & ::= & CollPath/\ ftags \\
ftags & ::= & ftag \mid ftags \wedge ftag
\end{array}
$$

Figure 4. LTTs path syntax

B. LTTs Operations

This model allows the following function calls. Naturally, any interface built upon the API may have different operations that translate to these functions.

1. CrCollection (tags, parentPath) Create a new collection: To complete this operation, the precondition is to provide the input parameters which are new collection (tags) and the parent: the path sequence of collections from the root node to a target collection. First the parent parameter is checked where it must exist and then the tag

parameter is checked where it must not exist within the parent collection. For instance, we want to add a new collection {068C,ref materials} within 2021 collection. CrCollection({068C ,ref materials },2021/)

2. DelCollection(tags,parentPath) Delete a collection: The precondition to completing this operation is that the target collection tag must exist in the parent collection. In addition, the collection must be empty which means that all its sub-collections and files have already been deleted (no sub-collection and files at all). Note that we can locate the collection by providing a subset of the collection tags which must uniquely identify the collection. For instance, we want to delete {068C, ref materials} to do this by providing the {068C, ref materials}, 068C, or ref materials as long as they uniquely identify the collection.

3. UpCollection (operation, path, NewValue) Update a collection: the updating refers to change the tag value or the path of the particular collection. The input parameters are an operation that refers to the type of function call whether it is "move" the collection (changing its location) or, "add", "delete" -changing the set of associated collection tags where in this model rename operation will be expressed by deleting the old one and then adding the new one; old value always refers to the path (whether the operation move, add, or delete as the location of the collection needed in all these operations); and new value means the new path (location) if the operation is "move" while it is the new tag value if the operation is "add" or without value if it is "delete". The old and new values will be checked where the old value must exist and the new one must not exist and it will not affect the local uniqueness of the collection. This function means that all the sub-collections and files underneath this collection will be immediately changed as well.

4. CrFile (ftags, parentPath) create a new file: the input parameters of this operation are a tag/set of tags and collection path where the file will be. The operation preconditions are that the file does not exist and the new tag (set/subset of tags) must be locally unique. For example CrFile({assignment1, answer1},CS /). id returns form this operation.

5. DelFile(ftags,parentPath) refers to deleting a file from a collection with the precondition that the file exists with providing its tags or part of tags which uniquely identifies the file and its collection as well.

6. UpFile(operation,path,value) Update File: changing a tag value or the path of a specific file with a precondition that it must not affect the uniqueness of the files within its collection. This can be done by providing the input parameters that are: operation that means the type of this function call which is either "move", "add", "delete" tags as the files in this model may have set of associated tags, rename operation will be expressed by deleting the old one and then add the new one; old value always refers to path (whether the operation move or rename as the location of the collection needed in both operations); and new value means new path (location) if the operation is "move" while it is the new tag value if the operation is "rename" The old and new values will be checked where the old value has to exist and the new one must not exist and it will not affect the local uniqueness of the file within the collection.

7. ***CrFileLink(filePath,parentPath) Create a new file link***: this means associating existing file to another collection, so the file will exist in more than one collection at the same time. This requires that the file and the parent exist and the file must not exist in this parent.

8. DelFlileLink (filePath) Delete a file link: refers to delete one possible path by deleting one membership of file with a collection. To complete this operation all we need the file which must exist and the file must exist in another collection.
9. UpFileLink(oldFilePath,parentPath) Update File: This operation to change a file link. It refers to two operations deleting the old link and then follow by creating the new link. The preconditions to complete this operation are as the previous operations (CrFileLink (oldFilePath,parentPath)and DelFileLink (oldFilePath)

C. LTTs Queries

LTTs model adds a query language to the file system API as mentioned early. The query language for LTTs is shown in Figure 5. The main LTT features that set it apart from the traditional Hierarchical File System are adding a query language to the file system API, and allowing collections & files to have multi-tags attached. These features add more expressive for LTT file system organizations in that the presence of multiple tags at both collection and file-level support the disjunctive conditions at both levels.

$$
\begin{array}{lll}
Query & ::= & path \mid path\ fileQ \\
path & ::= & collQ/ \mid path\ collQ/ \mid path \wedge path \\
collQ & ::= & ctag \mid collQ \vee ctag \mid collQ \wedge ctag \\
fileQ & ::= & val \mid fileQ \vee fileQ \mid fileQ \wedge fileQ \\
val & ::= & ftag \mid \neg ftag
\end{array}
$$

Figure 5. LTTs model query language

For example, the query CS2∧CS3/ identifies all collections that have tags ‘CS2’ and ‘CS3’ and will return all files recursive located within those collections that have both tags. On the other hand, the query CS2∨CS3/ identifies all collections that have tags ‘CS2’ or ‘CS3’ and will return all files recursively located within those collections that have either CS2 or CS3. In addition, the query language shows that it can re-find files using different paths because of adding links to the data model. Allowing file links facilitates re-find files and collections through links as highlighted.
Furthermore, the query language shows that it can re-find files using different paths because of adding links to the data model. For example the query CS2∧CS3/ identifies all collections that have tags “CS2” and “CS3” and will return all files recursive located within those collections. Complex queries that has both ∨ and ∧ are also possible. Complex queries like the following are possible.
courses/2021/CS2∨CS3/assignment∨exam
courses/2021/CS2∧CS3/reference
courses/2021/CS20∧CS21/reference∧¬Git
Concrete query syntax would require addition of parentheses to resolve operator precedence in cases like courses/2021/CS2/(assignment∨exam)∧results but has been elided here for simplicity.

5. LTTs Evaluation

Firstly, we will expose why is a better solution model. This model supports metadata management better than just tree-based models and rooted graph models (such as VennFS [1]) in terms of the following reason:

- Simplicity: In rooted graph (VennFS), to add or remove a tag, usually because the classification needs to be modified to better reflect reality, it often requires creating a new collection which links to other collections that the files have to belong to. This means that to solve the problem of collections' links, many operations should be done while with LTTs this problem can be solved in just one operation by linking the file with a collection that reflects the new classification desired.

- Another reason is that in the rooted graph model, the new collection created will be unknown; which tag should be given? So, the updating operation will not easy as the LTTs updating operation. Secondly, we will show how LTTs solve the HFS problems. The LTTs file system structure provides a solution to the problems detailed in Section 2. The provision of multiple tags for a collection (file container) and files allows multiple classification schemes. Figures 6 and 7 show how LTTs solve problems 1 (Artificial hierarchies) and 2 (Classification). More visible collection tags can better inform the orienteering style of search that descends a tree to locate a file (Problem 3 (Problematic Pruning)). LTTs model also supports multi-classification by file links. This means that files can exist in more than one collection and multi-tags attached for both collections and files. Based on both file links and the set of tags for collections and files, this model has the ability to provide most of the users requirements by allowing to use ∧,∨,¬ to retrieve files where it can be used as a complex query. The links in the model support metadata management by reducing the number of operations required to update a file. Multi-tags also shows support metadata management. As LTTs has both links and multi-tags, it will be more powerful model in terms of solving the HFS and reducing the required operations that users have to do for updating their files.

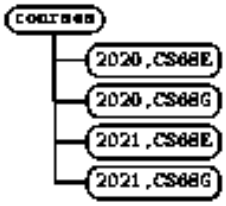

Figure 6. Collections with multiple tags

Figure 7. Supporting multiple classifications

Finally, the LTTs model meets most of the metadata management criteria. The ease of tag manipulation supports associating more relevant metadata with groups of files (Problem 4 (Metadata management)).

6. Conclusions and Future Work

The HFS is ubiquitous and used in operating systems old and new. Over the years, there have been many attempts to define new file and add-on applications that offer much-improved metadata capabilities. The proposed model was motivated by the fact that users of these systems struggle with both properly organizing files and simply relocating them in the future. Researchers have looked at many elements of post-hierarchical file management systems to solve the well-documented drawbacks of conventional systems. In some cases, work has resulted in significantly different file system architectures, while in others new functionality is added via user-space applications on top of traditional architectures. Separately, some proposals prefer tags, whereas others prefer named attribute-value pairs. In this paper, we discussed the details of a file management system structure called LTTs that can reuse tags and links without the disadvantages that exist with traditional HFS links, which refer to linking files simultaneously in different paths. We showed that LTTs resolve the identified HFS problems. Expanding on the current work will take two main directions. We have evaluated the LTT model from a practical standpoint. Developing a proof-of-concept implementation requires key decisions on data structures and algorithms; comparing the software with traditional file systems requires the creation of a metadata-oriented benchmark that could also be used to measure the efficacy of other novel file systems. However, perhaps of greater importance is the possibility of designing user interfaces that can take advantage of the LTTs file system's rich metadata schema and query API. Second, future research should look at alternative models that can solve HFS problems by using attributes instead of tags and examine how they differ in terms of how they solve the problems and whether or not they add complexity for the end-user.

Acknowledgments

The authors wish to thank Dr. Richard Watson in the University of Southern Queensland for his valuable comments on the proposed model. Also, thanks to the sponsor of the first author Ministry of higher education and scientific research (Iraqi Government) and University of Thi-Qar. The first author also thanks the University of Southern Queensland for accepting her in one-year sabbatical leave.

References

1. N. Albadri, S. Dekeyser, and Richard Watson. VennTags : A file management system based on overlapping sets of tags. In Proceedings of Conference 2017 Proceedings. iSchool, 2017.
2. Nehad Albadri, Richard Watson, and Stijn Dekeyser. TreeTags: Bringing tags to the hierarchical file system. In Proceedings of the Australasian Computer Science Week Multiconference, Canberra, Australia, pages 21–31, 2016.
3. Alexander Ames, Nikhil Bobb, Scott A Brandt, Adam Hiatt, Carlos Maltzahn, Ethan L Miller, Alisa Neeman, and Deepa Tuteja. Richer file system metadata using links and attributes. In Proceedings of 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies, pages 49–60. IEEE, 2005.
4. Sasha Ames, Maya Gokhale, and Carlos Maltzahn. QMDS: a file system metadata management service supporting a graph data model-based query language. International Journal of Parallel, Emergent and Distributed Systems., 28(2):159–183, 2013.

5. Sasha Ames, Maya B Gokhale, and Carlos Maltzahn. A metadata-rich file system. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.
6. Deborah Barreau and Bonnie A Nardi. Finding and reminding: file organization from the desktop. ACM SigChi Bulletin, 27(3):39–43, 1995.
7. Ofer Bergman, Noa Gradovitch, Judit Bar-Ilan, and Ruth Beyth-Marom. Folder versus tag preference in personal information management. Journal of the American Society for Information Science and Technology, 64(10):1995–2012, 2013.
8. Ofer Bergman, Tamar Israeli, and Yael Benn. Why do some people search for their files much more than others? a preliminary study. Aslib Journal of Information Management, 2021.
9. Ofer Bergman, Tamar Israeli, and Steve Whittaker. Search is the future? the young search less for files. Proceedings of the Association for Information Science and Technology, 56(1):360–363, 2019
10. Brian Carrier. File system forensic analysis. Addison-Wesley Reading, 2005.
11. Stijn Dekeyser and Richard Watson. Metadata manipulation interface design. In Proceedings of the 14th Australasian User Interface Conference, pages 33–42. Australian Computer Society, Inc., 2013.
12. Stijn Dekeyser, Richard Watson, and Lasse Motrøen. A model, schema, and interface for metadata file systems. In Proceedings of the thirty-first Australasian conference on Computer science, pages 17–26. Australian Computer Society, Inc., 2008.
13. Jesse David Dinneen and Charles-Antoine Julien. The ubiquitous digital file: A review of file management research. Journal of the Association for Information Science and Technology, 71(1):E1–E32, 2020.
14. David K Gifford, Pierre Jouvelot, Mark A Sheldon, et al. Semantic file systems. In ACM SIGOPS Operating Systems Review, pages 16–25. ACM, 1991.
15. IEEE. 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. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base, 2004.
16. Thomas W Jackson and Stephen Smith. Retrieving relevant information: traditional file systems versus tagging. Journal of Enterprise Information Management, 25(1):79–93, 2011.
17. William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. Betrfs: A right-optimized write- optimized file system. In FAST, pages 301–315, 2015.
18. William Jones, Abe Wenning, and Harry Bruce. How do people re-find files, emails and web pages? In Proceedings of iConference 2014. iSchools, 2014.
19. Peter Klemperer, Yuan Liang, Michelle Mazurek, Manya Sleeper, Blase Ur, Lujo Bauer, Lorrie Faith Cranor, Nitin Gupta, and Michael Reiter. Tag, you can see it!: using tags for access control in photo sharing. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 377–386. ACM, 2012.
20. Anghel Leonard. Manage symbolic and hard links. In Pro Java 7 NIO. 2, pages 35–42. Springer, 2011.
21. Han Lin, Huang Hao, Xie Changsheng, and Wang Wei. Clustering files with extended file attributes in metadata. Journal of Multimedia, pages 278–285, (2021).
22. Syed Rahman Mashwani, Azhar Rauf, Shah Khusro, and Saeed Mahfooz. Linked file system:Towards exploiting linked data technology in file systems. In 2016 International Conference on Open Source Systems & Technologies (ICOSST), pages 135–141. IEEE, 2016.
23. Mahajan, H.B., Rashid, A.S., Junnarkar, A.A. et al. Integration of Healthcare 4.0 and blockchain into secure cloud-based electronic health records systems. Appl Nanosci (2022). https://doi.org/10.1007/s13204-021-02164-0
24. Ofer Bergman, Tamar Israeli, and Yael Benn. Why do some people search for their files much more than others? a preliminary study. Aslib Journal of Information Management, 2021.
25. Brackenbury, W., Harrison, G., Chard, K., Elmore, A., & Ur, B. Files of a feather flock together? Measuring and modeling how users perceive file similarity in cloud storage. In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021.

26. Brackenbury, W., McNutt, A., Chard, K., Elmore, A., & Ur, B. KondoCloud: Improving Information Management in Cloud Storage via Recommendations Based on File Similarity. In The 34th Annual ACM Symposium on User Interface Software and Technology, 2021.
27. Conrad, L. Y. Managing academic information: A grounded theory of the student-researcher information experience (Doctoral dissertation, Queensland University of Technology), 2022.
28. Al-ali, A. H., Qalaja, L. K., & Abu-Rumman, A. (2019). Justice in organizations and its impact on Organizational Citizenship Behaviors: A multidimensional approach. *Cogent Business & Management*, *6*(1).
29. Alon, L., & Nachmias, R. Gaps between actual and ideal personal information management behavior. Computers in Human Behavior, 2020.
30. Alon, L., & Nachmias, R.The role of feelings in personal information management behavior: Deleting and organizing information. Journal of Librarianship and Information Science, 2022.
31. Mashwani, S. R., & Khusro, S. 360 semantic file system: Augmented directory navigation for nonhierarchical retrieval of files. IEEE Access, 2019.
32. Tian, H., Ju, F., Nie, H., Wu, Y., Yang, Q. and Li, S., A new technology for real-time file system of high-speed storage system in airborne sensors. IEICE Electronics Express, 2021.

Biography

Nehad Albadri is now a lecturer with the College of Education for Pure Sciences, University of Thi-Qar, Iraq. She has a PhD in computer science from USQ in Australia. She is interested in Digital Signal Processing , operating systems, database management systems, and text mining.

Stijn Dekeyser is currently an associate Professor of Computer Science with the School of Sciences, University of Southern Queensland, Australia. He earned his PhD from Antwerp University in Belgium in 2003. His research interests include data management, metadata file systems, web technology, and mobile systems.

Conflicts of Interest Statement

Title: **A Novel File System Supporting Rich File Classification**, We certify that they have no any financial and personal relationships with other people or organizations that could inappropriately influence (bias) their work. Examples of potential conflicts of interest include Employment, consultancies, stock ownership, honoraria, paid expert testimony, patent applications, we are non-financial interest, and there are no conflicts of interest.

This study was self- funded, All authors declare, have no conflict of interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or Materials discussed in this manuscript.

This statement is signed by all the authors to indicate agreement that the above information is true and correct

Authors Name

Nehad Albadri and Stijn Dekeyser