Mars: Jurnal Teknik Mesin, Industri, Elektro dan Ilmu Komputer
Volume 3 Nomor 1, Tahun 2025
e-ISSN : 3031-8742, dan p-ISSN : 3031-8750, Hal. 120-138
DOI: https://doi.org/10.61132/mars.v3i1.607
Available online at: https://journal.arteii.or.id/index.php/Mars

Attributes or Tags for Files? AttFS: Bringing Attributes to the Hierarchical File System

Nehad Albadri

Department of Computer Sciences, College of Education for Pure Sciences, University of Thi-Qar, Iraq

Email: nihadghasab.comp@utq.edu.iq

Abstract: *File systems are key operating system components because they store and retrieve files as needed. Traditional hierarchical file systems no longer suit current users' expectations in organizing extensive collections of files for easy retrieval based on intrinsic and user-defined properties. The number of files in users' collections is growing substantially, partly because of the ease with which consumer devices capture information. With the enormous capacity of contemporary storage devices and the rising trend of users storing their data in the cloud, they only fuel the number of files needing to be managed. We suggest that current file systems require an enhanced technique of data organization and access so that users can properly handle these ever-increasing data collections. File systems are key operating system components because they store and retrieve files as needed. This study offers improvements to the conventional Hierarchical File System to improve file organization and retrieval through built-in querying capabilities and support filesystem-level operations that execute significant amounts of metadata updates. This is accomplished using attributes (name-value pairs) in a file collection hierarchy. A series of improvements to the HFS introduce the "AttFS" file system. These improvements include using attributes rather than names, logical collections rather than directories, and introducing a query language to the API. We assess the expressive capacity of the resulting model, demonstrate that it solves the relevant shortcomings of traditional file systems in this space, and compare our new approach to those provided by others and our earlier work. We conclude that attributes are better suited than tags to overcome traditional HFS shortcomings.*

Keywords: *Attributes, Tags, Hierarchical File System.*

1. INTRODUCTION

According to file system researchers [1, 2, 3, 4], the number of files and the average file lifespan have increased significantly over the last 30 years. This is because the number of files owned by a given user has increased in lockstep with the expansion in storage capacity. Indeed, storage capacity has risen to the point where removing old content is no longer necessary; in fact, the work users spend doing would be a waste of time [5]. The challenge has shifted from deciding what to save to locating specific data when required. To remedy this issue, users must improve their file organization and search capabilities [6, 7, 8].

The central feature traditional file systems provide end-users to organize their files is the folder or directory, arranged in a hierarchy. While this has worked well for smaller files, users have found over time that the traditional hierarchical model has significant limitations, making it hard to manage vast collections of files that often cannot simply be organized in a tree. In many cases, an alternative information system that was not dependent on hierarchical path names was overlayed on top of the filesystem to overcome several problems of the HFS [9, 10, 11].

Received Desember 02, 2024; Revised Desember 16, 2024; Accepted Desember 31, 2024;
Published: Januari 06, 2025

A. Motivation

The following example shows the motivation behind this work. Assume a user has thousands of images, movies, papers, and other material on their computer or device. The user wants to organize these files using HFSs for convenient access. To manage data, the user knows that each set of files must be saved in a folder with a name that matches its contents. The user can create a folder for music, another for photographs, and so on, each with its subfolders. In the photo folder, the user must organize files. The following example shows user challenges: Users can choose one folder per group year—figure 1 shows option (1). If the user wants to find a photo she knows was taken in a particular place but not the year, she must search all files. The user may decide to add file category features. Photo folders are organized by year and place (see Figure 1(2)). After placing these files in their folders, some contain only a few photos and do not need a separate folder. After putting these files in their folders, some contain few photos and do not need a separate folder, while others have too many.

The user realizes the files must be recategorized using different criteria. Figure 1(3) suggests organizing photos by crucial events. After organizing files into folders, it becomes evident that some photos must be in both the Holiday and Birthday folders. Due to the previous difficulties, as shown in Figure 1(4), the user reclassifies photos by year and occasion. After putting files in folders, the user sees that some folders have more photos than others, making it hard to search for a specific photo. Instead, they would sort photos by year, event, and place (Figure 1(5)). If another user (such as the original user's partner) wants to find a Roma Park photo and browse the collections, they may not know if it was labelled a Holiday or a Birthday, so they look in both files. As seen in this situation, a hierarchical (conventional) file system is wasteful and inefficient for file organization. The impracticality applies to any file type and its data, not only photo file organization. The key to motivating our work is plenty of special-purpose software to manage various file formats. Numerous consumers use photo-management tools. These methods are problematic because (a) they are not generic and (b) they cannot seamlessly integrate photo information with rich event or actual item information. The user may want to assign several attributes to items like music,

documents, and programs to categorize them. Traditional hierarchical file systems classify files using only one of these qualities: folders. Because one attribute is allowed, there is one classification scheme [12]. The scenario analysis shows that a multi-classification system is needed.

Figure 1: Files organization choices

B. HFS Problems

The issues mentioned in the previous section occur in the daily lives of IT users. In our broad research in this area [13, 14, 15, 16, 17], we have detailed the limitations of the HFS and provided examples for each. For ease of understanding, we provide a summary of these issues:

1) File attributes do not create natural subclass relationships, resulting in artificial hierarchies.
2) Items within the type hierarchy can frequently belong in more than one subtree. However, a traditional file system is a single arrangement system. Therefore, the

user has to either duplicate the item or utilize the facility of the hard/soft links, which has some deficiencies that we detailed in [14].

3) As a consequence of the single classification, HFS fails to support orienteering-style search. Indeed, if searchers navigate the directory tree incorrectly, they will never discover the file.
4) Metadata updates in HFS are inefficient in bulk. So, a user may want to add or remove such metadata to improve categorization accuracy. In that case, creating, deleting, and renaming actions must be performed in the precise order.
5) The lack of a powerful query mechanism built into the HFS APIs leads to inconsistencies in special-purpose search solutions and an increase in erroneous categorization instances.
6) A reliable categorization method is necessary for efficient file navigation in a file system directory tree. An orienteering process will never find inconsistently categorized files. HFSs use hierarchical folders to classify files, but all files in each directory should have a shared subset of attributes different from those in sister directories to ensure consistency. The correct directory name usually implies these traits.

Some problems can be solved using tags[1], as we demonstrated in our previously proposed models, TreeTags and VennTags; however, those tag models cannot solve several remaining problems. Instead, as we will explore in the following sections, attributes can address those.

Organization In this paper, we summarize the restrictions of traditional file systems in file management retrieval, as elaborated in Sections 1 and 2. In addition, we define the primary service necessities with some significant definitions. The paper's primary contribution is proposing a unique model (AttFS) (Section 3). In Section 4, the AttFS model is evaluated regarding resolving HFS limitations and compared to other solutions offered by us to demonstrate the distinctions between using attributes and tags and which one is superior in terms of resolving file systems' limitations. The paper also compares solutions offered by others to determine the distinctions and benefits of our proposed model.

Contributions The paper's contributions include:

1) Naming and describing difficulties in the hierarchical file system.
2) Proposing AttFS as a formal file management model to solve common problems experienced with traditional hierarchical file systems.
3) Demonstrating that using attributes for files and collections is a better solution than tags for the highlighted limitations.
4) Introducing a query language within the File Management System API to facilitate efficient file retrieval.

2. FRAMEWORK

Our suggested file management system provides three types of services:

1) Create a file along with its corresponding metadata.
2) file identification
   a. For an 'open' file, it is required to locate a file using metadata-based specifications. The file system metadata must be handled to give each file a unique metadata specification.
   b. A query service that retrieves files conforming to specified class membership criteria would reveal the contents of a directory. We apply this to generic file system queries. These procedures are essential for file system user interfaces.

3. EDITING FILE METADATA

1) Modify metadata for a single file, such as tag, name, or attribute.
2) Reorganize a set of files by applying complicated metadata modifications, potentially reclassifying them.

The main physical objects in the AttFS are:

a. Attributes are metadata for collections and files. It could be name/value pairs. Name/value pairs offer files and collections meaning, unlike HFS name files and directories.
b. Files store bits (atomic data units). System ID is unique. The system identification represents the files' logical organization, not the content. Thus, 'file' usually denotes 'file identifier'. Files have attributes.
c. A collection holds files. It has a unique system identity and zero or more files (file identifiers). The collections are tree-like. All files in a collection are there because they share semantic properties, such as those associated with a project. A collection

can have a name/value attribute. Every file in the collection inherits this name/value pair and any ancestor collection attributes (see path discussion below).

d. Paths are sequences of collection members that are children of the previous collection. The path from tree root to collection explicitly identifies the collection. Paths identify collections. A file path combines a collection path and the identified file. Routes help file system users find files [18].

e. Queries are search conditions based on collections and files' metadata (name/values). A search using a query yields zero or more files from many collections. File system enquiry is unlike most HFS APIs. A client application of a typical HFS can imitate file system query capability, although performance may suffer due to repetitive file system calls.

f. AttFS File System

Tradition file system is a monolithic categorization method that presents organizational and retrieval issues. However, both methods are isomorphic from a critical standpoint. So, we provide a file system based on hierarchies of collections that are entirely independent of the set of files it maintains.

Additionally, our approach relies not on directory or file names but on a multi-classification model based on name/value pairs. This change allows files to be placed in the appropriate collection, including the collection in multiple additional collections, as illustrated in the following.

By digging extensively into each component of our proposed model (dubbed AttFS), we address the issues raised in Section 1.1. Following a formal definition of the data model's low-level operations, a thorough description of the model's high-level queries is provided (in Z notation).

4. DATA MODEL

The collections are arranged in a tree-like pattern. Sub-collections may exist inside a collection. The words parent and child logically represent the connection between subcollections and collections; more broadly, any two collections have a unique path. Paths can be represented in terms of an ordered sequence comprising zero or more elements. A specific type of collection, termed root that lacks a parent. As noted previously, certain collections may contain a plurality of members, implying that they do have many paths.

District-based files are used to create collections. Files are simple belongs-to-relationships with collections. The word ‘collection’ was used to differentiate the traditional from the more conventional ‘directory expressly’ (or ‘folder’).

1. Hierarchy

   $H : cid \rightarrow cid \cup \{\tau\}$ where

   • cid is the type of collection identifiers.

   The hierarchy uses cid instead of attributes. This prevents the hierarchy from being involved in unnecessary actions like updating attribute values. When considering names − value, the hierarchy changes with each operation.

   • H represents a collection identifier tree that has a root τ.

   • H (s) is the parent of s.

   • Initial value: $H = \emptyset$

   • Constraint: $\forall s \in dom\ H \bullet (s, \tau) \in H^+$

   A single rooted tree contains all collection identifiers. This limitation prevents cycles.

2. Attributes of Collection

   $S : cid \cup \{\tau\} \rightarrow (A \rightarrow \top)$

   • cid is the collection identifier

   • Initial value: $S = \{\tau \rightarrow root\}$

   • A is set of attribute names.

   • $\top$ is a universal top type -set of values is the universal type

   If the tuple $(id, \{(a_1 \rightarrow v_1), (a_2 \rightarrow v_2), \ldots (a_1 \rightarrow v_n)\})$ is an member of S , then the value of attribute $a_i \in A$ is $v_i \in t_i$

   where $t_i \subset \top$.

   Permitting multiple attached attributes with a collection will provide flexibility to locate collections.

3. Files

   $F : cid \rightarrow (id \rightarrowtail (A \rightarrow \top))$

   • Files are organized into collections, and each collection has a unique identifier..

   • Each file's name(/value) mapping is bidirectional.

   $F : id \rightarrow (A \rightarrow \top) \wedge F^{\sim} : (A \rightarrow \top) \rightarrow$ id and an identifier (id).

   • Initial value: $F = \emptyset$

• $\exists(s_1, (a_1 \rightarrow v_1)), (s_2, (a_2 \rightarrow v_2)) \in F \bullet s_1 \neq s_2 \wedge \text{dom } A_1 \cap \text{dom } A_2 \neq \emptyset$. A file may be referenced only once within a collection.

4. The path from H, S and the following functions are derived.

   The path function D : cid → seq cid .

   The path function P : cid → seq A → ⊤ is now a sequence of (name/values) groups.

   P(s) = {n : N; id : cid | (n, id) ∈ D(s) • (n, S (id))}

   A collection path (P) is formally defined as a succession of attributes (name/value) sets. The abstract syntax shown in Figure 2 is for paths that contain multiple name/value pairs for both collections and files.

   It is important to recognize that although P generates paths encompassing all attributes (names/values) linked to collections, a path specification containing fewer characteristics per object can still effectively identify a single file within any specific instance of a file system. A single object may possess multiple path specifications.

Collpath	::=	val/	|	path collQ/
collQ	::=	val	|	collQ ∨ val
fileQ	::=	val	|	fileQ ∨ val
val	::=	A = val | A ≠ val		

Figure 2: AttFS path syntax

However, utilizing entirely path specifications is helpful. If an object is instantiated with a fully specified path p where the attributes linked to the object in p remain unchanged, that file can consistently be accessed via path p.

Note that in this proposed model, we have not introduced the concept of view at all. This is because views in our proposed model add more complexity for users; it cannot recognize the views and collections. The users can attach metadata to that collection and files, while in the view world, there is no such thing as present metadata being attached to views. Views are conceptual groups that are likely related to the flat structure.

5. ATTFS OPERATIONS

The following function calls are among the available operations for our proposed model (AttFS). Any interface developed on the API may exhibit distinct operations corresponding to the functionalities. There are collection operations and file operations. Some collection and file operations require collection and file lookup sub-operations.

The collection lookup operation has the following semantics.

lookup : (A →⊤) → cid

lookup(atts) = S∼(atts) if atts ∈ ran S

The file lookup operation has the following semantics.

lookup : (A → ⊤) → id

lookup(atts) = F∼(atts) if atts ∈ ran F

Informally, either the supplied attributes exactly match a collection's/ file's attributes or a subset of a collection's/ file's attributes, and only one collection/file has a matching subset.

Note: Before we explore the AttFS operations, we will consider files associated with university courses as an explanation example. These courses have properties such as 'course code' and 'year of offer', organized by course and year.

1. CrCollection(att,parentPath) Creating a collection: The input parameters are the new attributes, and the parent is the path sequence of collections to a target collection. The parent parameter must exist in this operation, and the lookup collection function fails. The characteristics parameter is then verified for parent collection type conformance. Upon meeting these conditions, the operation will be finished.
   Assuming an undergraduate program, we want to add CS100 to the 2020 collection. The desired path can be found using CrCollection ((Course, CS100),/2020) call function. This operation returns cid.
2. DelCollection(att,parentPath) Att must be in the parent collection for the lookup collection function to work for the delete action. All sub-collections and files must be deleted from the collection. Call DelCollection ((year, 2020),/courses) to remove the 2020 collection from Courses. If the 2020 collection has sub-collections, this returns false.
3. UpCollection(oldpath,newpath) Updating a collection path is called updating. The input parameters are as follows: A new path signifies "move." The old value must exist but not the new. Local collection uniqueness and matching with the new path (location) attributes will not be affected. This function changes all sub-collections and files in this collection instantaneously.

4. CrColAtt(collection,atts) Create attributes: This operation refers to adding new attributes to a collection identified by the attributes collection. To complete this operation, lookup(collection) succeeds and (S (lookup(collection)) ∪ atts)∉ran S where adding atts does not create the attribute set of an existing collection.
5. DelAtt(collection,atts) Delete Attributes: This operation refers to deleting a specific set of attributes from a collection identified by attributes collection. In this operation, lookup(collection) succeeds and (S (lookup(collection)) \ atts)∉ran S . That is, deleting these attributes does not affect the uniqueness of existing collections.
   For instance, to delete the Year=2016 attribute from the file{(Course, CS100), (Year, 2016) }, first, it will be checked whether there is a collection that is just (Course, CS100) or not. If yes, the operation will fail. Otherwise, the operation will be complete, and the collection has (Course, CS100).
6. UpAtt(collection,oldAtts,newAtts) Update Attributes: This operation refers to changing the name of an attribute in the collection. The operation precondition is lookup (collection) succeeds and ((S (lookup(collection)) \ oldAtts) ∪ newAtts)∉ran S where replacing these attributes does not impact the uniqueness of the existing collection.
7. CrFile (fatt, parentPath) Creating a new file: adding a new file requires the provision of a file att (fatt) which must be locally unique and where lookup (file) fails. The new file will be in a collection, so it must provide its collection path. This operation returns id for the new file.
8. DelFile(fatt,parentPath) refers to deleting collection files. To complete this action, the target file att must exist where lookup (file) succeeds in the collection parent Path.
9. UpFile(operation,path,newAtt): Updates change attribute values or file paths. The input parameters are as follows: operation refers to a function call that changes file location or attribute; old value is the path (whichever the operation is); the path is either a new location (path), if the operation is changing the file location, or the new attribute value if the operation is renaming the file new and old values, will be examined; and newAtt is the new value which must not change the file's local uniqueness in the collection. For example, to move f_1 from collection

(Course,CS01) to (Course,CS200), we need

UpFile call function(move, /CS 100/f_1, CS 200).

10. CrAtt(file,atts,path) Create attributes: This operation refers to adding set of new attributes atts to a file identified by attributes file and exists in a specific collection. To complete this operation, lookup(file) succeeds and (F (lookup(file)) ∪ atts)/ ran F. So, adding atts does not create the attribute set of an existing file in that path.
11. DelAtt(file,atts,parentPath)Delete Attributes: This operation refers to deleting a specific set of attributes from a file identified by the attributes file. To complete this operation, lookup (file) succeeds and (F (lookup(file)) \ atts)/ ran F. That is, deleting these attributes does not effect on the uniqueness of existing files. For instance, to delete Year=2016 attribute from the file {(Course, CS100), (Year, 2016)}, first it will be checked whether there are a file that is just (Course, CS100) in this path or not. If yes, the operation will fail; otherwise, the operation will be complete and the file has just (Course, CS100).
12. UpAtt(file,oldAtts,newAtts,parentPath)Update Attributes: This operation refers to changing the name of an attribute in the file. To complete this operation, lookup(file) succeeds and ((F (lookup(file)) \ oldAtts) ∪ newAtts)/ ran F. That is, replacing these attributes have no effects on the uniqueness of existing files in its parent path.

6. ATTFS QUERIES

One of the main differences between AttFS and other Hierarchical File Systems is the file system API query language; Figure 3 exhibits the inquiry language. The path language, which identifies a single object, is expanded by exchanging the collection attribute with a disjunctive list of attributes and the file attributes with their inverse. Thus, an attribute's presence or absence may identify a file for the query result. Mathematical operator symbols represent disjunction and negation in this abstract grammar. An implementation would use ASCII/Latin-1 textual symbols for these operations.

A query returns a list of files. For example, all files in a collection should be retrieved with the attribute ‘Year, 2014’. Because the files inherit the collection (parent) attribute, all files under the /Year, 2014 collection will be returned. This means that all

files in / Year, 2014 and their sub-collections will be retrieved. The query language allows using ∧ and ∨ as shown in Figure 3.

For instance, (Course, CS400)∧(Course,CS500) query finds every collections with Course='CS400 and Course=CS500 attributes and will retrieve all-recursive files located in those collections with both attributes. On the other hand, (Course, CS400)∨(Course,CS500)/ query find all collections that have attributes Course='CS400' or Course='CS500' and will retrieve all files recursively located in those collections that have either CS400 or CS500.

$$
\begin{array}{lcl}
\mathit{Query} & ::= & \mathit{path} \mid \mathit{path}\ \mathit{fileQ} \\
\mathit{path} & ::= & \mathit{collQ}/ \mid \mathit{path}\ \mathit{collQ}/ \\
\mathit{collQ} & ::= & \mathit{val} \mid \mathit{collQ} \vee \mathit{val} \mid \mathit{collQ} \wedge \mathit{val} \\
\mathit{fileQ} & ::= & \mathit{val} \mid \mathit{fileQ} \vee \mathit{fileQ} \mid \mathit{fileQ} \wedge \mathit{fileQ} \\
\mathit{val} & ::= & A = \mathit{value} \mid A \neq \mathit{value}
\end{array}
$$

Figure 3: AttFS query language

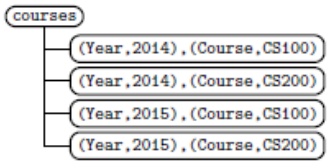

Figure 4: Collections with multiple attributes

Furthermore, intricate queries such as the following are feasible.

courses/2014/CS600∨CS700/assig∨exam

courses/2015/CS600∧CS600/source

courses/2015/CS700∧CS600/source ¬Git

Concrete query syntax requires parentheses to resolve operator precedence in scenarios like

courses/2015/CS600/(assig∨exam)∧results

but it has been elided here for brevity

7. EVALUATION

Our evaluation will be in two parts as shown in the subsections below

A. How does AttFS solve HFS problems?

The AttFS file system solves the problems detailed in 1.2. The availability of multiple attributes for a collection allows multiple classifications (problem 2) as shown in Figure 4, and so the user can avoid the constructing of artificial hierarchies (problem 1) as explained in Figure 5 . More visible collection properties can help orienteering searchers descend a tree to find files (problem 3). Finally, attribute manipulation makes it easier to link useful metadata to file groups (issue 4). Additional file properties can help users in the latter two cases: Finding files and managing metadata.

It should be noted that while multiple attributes give users better tools to organize consistent file hierarchies, any user's success depends on their ability to find suitable attributes to categorise the files they create.

A general and powerful attribute-based query is a new API feature. Like some aspects of the attribute structure, it relies on suitable user interfaces to deliver real utility to users. In particular, the correspondence between a query result and the

rectory or some real advances in GUI-

ement and

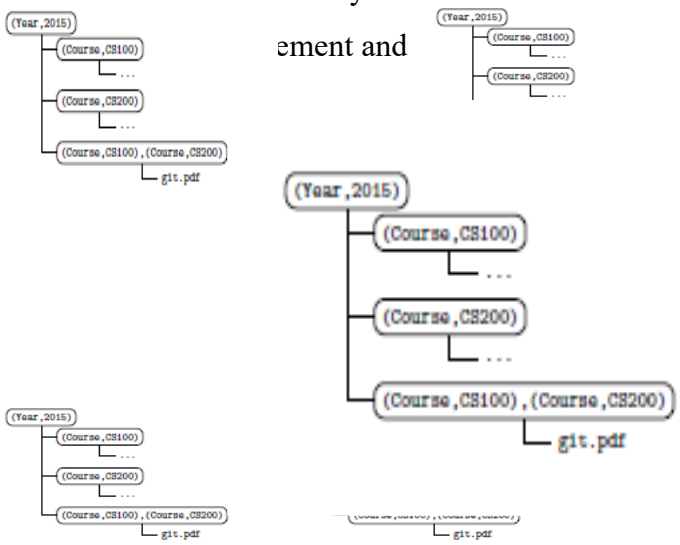

Figure 5: Supporting multiple classifications

The important part that the proposed model introduces is adding semantic meaning; operations can fail if there is no match in terms of the meaning. For

example, 'CS100' indicates that all files in that tree are associated with CS100, so operations to add something else should fail. As mentioned earlier, the HFS and other tags models lack a mechanism for formally defining directory semantics properties that all resident files should have. This is represented in problem 6, where tags-based models cannot solve it.

B. How AttFS is distinct from other work

In this section, we will first compare AttFS, an attributes file system, with our other proposed file systems to determine which is better at solving file system problems. We will also compare AttFS with previous work.

LTTs [13], TreeTags [15] and VennFS [16] are examples of tags file systems that are our previously proposed models. We demonstrated that these models solved the hierarchical file system problems with differences in simplicity and the number of operations required to add or delete tags (for more details, see [13,16]). However, these models fail to solve the point of semantics because tags do not have meaning, so when a user makes a mistake dropping a file to a collection that does not match its classification, it will be completed while in the attributed world, it is not as it will break the classification. This point has been solved by AttFS as well as the other problems, as shown in §4.1.

AttFS was compared to the solutions of prior academics [10, 19, 20, 22, 23, 24, 25, 26, 27, 28], who employed post-hierarchical file system architectures to address the challenges of conventional file systems. The referenced systems utilized an extensive file information set to organise and retrieve files instead of employing a hierarchical directory structure. These systems aimed to entirely exchange the conventional file system, although none have succeeded thus far. Users may encounter a novel issue because classifying items via a tree is straightforward, facilitates comprehension of tree topologies, and fosters familiarity.

Other researchers [11, 29, 30, 31, 32, 33] present models to assist users in organizing their files based on specified tags. However, as we explained earlier in this section, the differences between tags and attributes where the users may make a mistake in tagging systems and place a collection/file in the wrong path (not matching the classifications). In addition, because none of the cited methods includes a query language, users cannot simply re-find their files.

Other approaches, however, employ a different paradigm, such as directed graphs (DAG) and tags [34, 35, 36]. In the provided suggestions, tags are generated and stored automatically within a tag DAG folder. The primary disadvantage of these proposed systems is the difficulty of searching for and retrieving files, which takes time due to the absence of a query language. Furthermore, because the tags are provided automatically, these systems lack the metadata management to update group/subgroup files easily. As a result, individuals who are accustomed to a tree structure will find it difficult to use.

Another technique to cope with HFS restrictions is adding new functionality to the existing file system. Tags are used in a variety of methods. Tags have been utilized in social networks which established means of organizing multimedia and permitted users to attach tags with media objects and subsequently re-find those items using metadata [37, 38]. TagTree [39] is one example; it automatically utilized user-supplied tags to build and maintain a navigation tree structure of tags. TagTree created a hierarchy for each file in which many routes are constructed. This new technique is troublesome when files have numerous tags, causing the tag tree to grow exponentially.

As can be observed, AttFS differs from the other proposed solutions described above in some ways. One of these aspects is that users can attach attributes to collections and files, unlike other proposals. This encourages using many attributes to classify, retrieve, and alter files. The other issue is that employing attributes will add meaning to the model, which means that all files and collections will be classified correctly. There will be no room for error in organizing the collections and files because the operation will fail if no matching classifications are found (as explained in the operation section). The third feature is that AttFS includes an API with a query language to re-find files.

5 CONCLUSION

The fundamental contribution of this work is to present a formal definition of a file system structure that uses attributes and incorporates them into the tree paradigm. Compared to our previous tagging models [15, 16], AttFS has been proven to resolve the

observed HFS issues. Utilizing attributes is better than tags to solve all HFS problems in this paper.

There are two primary ways that the current work will be expanded. The first is to continue researching alternate models, such as those using a rooted graph (instead of a tree) paradigm. We will compare new models to AttFS to determine which is the most expressive and which model can be most effective for end-users. Practical testing of the AttFS model is the second area of focus for future research. Data structures and algorithms must be selected for a proof of concept implementation, and a metadata-oriented benchmark is needed to compare the software to conventional file systems.

REFERENCES

Albadri, N., & Dekeyser, S. (2022). A novel file system supporting rich file classification. Computers and Electrical Engineering, 103, 108081.

Albadri, N., Dekeyser, S., & Watson, R. (2017). VennTags: A file management system based on overlapping sets of tags. In Proceedings of Conference 2017 Proceedings, iSchool.

Albadri, N., Watson, R., & Dekeyser, S. (2016). TreeTags: Bringing tags to the hierarchical file system. In Proceedings of the Australasian Computer Science Week Multiconference (pp. 21–31). Canberra, Australia.

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

Ames, S., Gokhale, M., & Maltzahn, C. (2013). 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.

Amoson, J., & Lundqvist, T. (2012). A lightweight non-hierarchical file system navigation extension. In Proceedings of the Seventh International Workshop on Plan 9, IWP9 (pp. 11–13).

Bergman, O., Gradovitch, N., Bar-Ilan, J., Beyth-Marom, R. (2013). Tagging personal information: A contrast between attitudes and behavior. Proceedings of the American Society for Information Science and Technology, 50(1), 1–8.

Bergman, O., Israeli, T., & Benn, Y. (2021). Why do some people search for their files much more than others? A preliminary study. Aslib Journal of Information Management.

Bergman, O., Israeli, T., & Whittaker, S. (2019). Search is the future? The young search less for files. Proceedings of the Association for Information Science and Technology, 56(1), 360–363.

Bloehdorn, S., & Völker, M. (2006). TagFS-tag semantics for hierarchical file systems. In Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06).

Civan, A., Jones, W., Klasnja, P., & Bruce, H. (2008). Better to organize personal information by folders or by tags?: The devil is in the details. The American Society for Information Science and Technology, 45(1), 1–13.

Dekeyser, S., Watson, R., & Motrøen, L. (2008). A model, schema, and interface for metadata file systems. In Proceedings of the thirty-first Australasian conference on Computer science (pp. 17–26). Australian Computer Society, Inc.

Di Sarli, D., & Geraci, F. (2017). GFS: A graph-based file system enhanced with semantic features. In Proceedings of the 2017 International Conference on Information System and Data Mining (pp. 51–55).

Dinneen, J. D. (2018). Analysing file management behaviour (McGill University).

Dinneen, J. D., & Julien, C.-A. (2020). The ubiquitous digital file: A review of file management research. Journal of the Association for Information Science and Technology, 71(1), E1–E32.

Douceur, J. R., & Bolosky, W. J. (1999). A large-scale study of file-system contents. ACM SIGMETRICS Performance Evaluation Review, 27(1), 59–70.

Furnas, G. W., Fake, C., von Ahn, L., Schachter, J., Golder, S., Fox, K., Davis, M., Marlow, C., Naaman, M. (2006). Why do tagging systems work? In CHI ’06 Extended Abstracts on Human Factors in Computing Systems (pp. 36–39). ACM.

Gifford, D. K., Jouvelot, P., Sheldon, M. A., et al. (1991). Semantic file systems. ACM SIGOPS Operating Systems Review, 16–25.

Jones, W., Wenning, A., & Bruce, H. (2014). How do people re-find files, emails and web pages? In Proceedings of iConference 2014.

Lin, H., Hao, H., Changsheng, X., & Wei, W. (2014). Clustering files with extended file attributes in metadata. Journal of Multimedia, 278–285.

Livia, H., & Ross, P. (2010). Exploring place through user-generated content: Using flickr tags to describe city cores. J. Spatial Information Science, 2010(1), 21–48.

Ma, S., & Wiedenbeck, S. (2009). File management with hierarchical folders and tags. In CHI'09 Extended Abstracts on Human Factors in Computing Systems (pp. 3745–3750). ACM.

Mashwani, S. R., Rauf, A., Khusro, S., & Mahfooz, S. (2016). Linked file system: Towards exploiting linked data technology in file systems. In 2016 International Conference on Open Source Systems Technologies (ICOSST) (pp. 135–141).

Motrrøen, L. (2008). Metadata file systems: UI concepts (Master's thesis, University of Southern Queensland).

Nehad, A. (2018). Metadata in file systems: Exploring problems and solutions (Ph.D. thesis, University of Southern Queensland, Australia).

Ngo, B.-H., Silber-Chaussumier, F., & Bac, C. (2008). Enhancing personal file retrieval in semantic file systems with tag-based context. In EGC (pp. 73–78).

Ngo, H. B., Silber-Chaussumier, F., & Bac, C. (2008). A context-based system for personal file retrieval. In Addendum Contributions to the 2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing & Communication Technologies (pp. 167–171).

Padioleau, Y., Sigonneau, B., & Ridoux, O. (2006). LisFS: A logical information system as a file system. In Proceedings of the 28th international conference on Software engineering (pp. 803–806). ACM.

Rizzo, T. (2004). WinFS 101: Introducing the new windows file system. http://archive.today/ZfniG [Online; accessed February-2023].

Sajedi, A., Afzali, S. H., & Zabardast, Z. (2012). Can you retrieve a file on the computer in your first attempt? Think to a new file manager for multiple categorization of your personal information. In 6th international workshop on personal information management (pp. 10–21).

Schenk, S., Görlitz, O., & Staab, S. (2006). TagFS: Bringing semantic metadata to the filesystem. Poster at the 3rd European Semantic Web Conference (ESWC).

Seltzer, M., & Murphy, N. (2009). Hierarchical file systems are dead. In Proceedings of the 12th conference on Hot topics in operating systems. USENIX Association.

Siberschatz, A., Korth, H., & Sudarshan, S. (2011). Database System Concepts (6th ed.). McGraw-Hill Companies.

Soules, C. A., & Ganger, G. R. (2004). Toward automatic context-based attribute assignment for semantic file systems. Parallel Data Laboratory, Carnegie Mellon University.

Voit, K., Andrews, K., & Slany, W. (2011). TagTree: Storing and re-finding files using tags. In Information Quality in e-Health (pp. 1–12). Springer.

Voit, K., Andrews, K., & Slany, W. (2012). Tagging might not be slower than filing in folders. In CHI'12 Extended Abstracts on Human Factors in Computing Systems (pp. 2063–2068). ACM.

Watson, R., Dekeyser, S., & Albadri, N. (2017). Exploring the design space of metadata-focused file management systems. In Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2017 (pp. 20:1–20:10). Geelong, Australia.

Whittaker, S., & Massey, C. (2020). Mood and personal information management: How we feel influences how we organize our information. Personal and Ubiquitous Computing, 24(5), 695–707.

Whittaker, S., Matthews, T., Cerruti, J., Badenes, H., & Tang, J. (2011). Am I wasting my time organizing email?: A study of email refinding. In Proceedings of the 2011 annual conference on Human factors in computing systems (pp. 3449–3458). ACM.