VennTags: A File Management System based on Overlapping Sets of Tags

N. Albadri[1], Stijn Dekeyser[1], Richard Watson[1]
[1]University of Southern Queensland, Australia

Abstract
File systems (FS) are an essential part of operating systems in that they are responsible for storing and organising files and then retrieving those when needed. Because of the high capacity of modern storage devices and the growing number of files stored, the traditional FS model is no longer able to meet modern users' needs in terms of storing and retrieving files. So using metadata emerges as an efficacy solution for the limitations of file systems.
In this paper we propose a new model dubbed *VennTags* to solve the FS problems. We do this by utilising the idea of overlapping the sets as in Venn diagram, and adopting DAG structure (instead of tree) to achieve that we have used tagging capability and exposed a query language at the level of the API. We evaluate the expressive power of VennTags model that shows its ability to resolve the FS limitations compared to other solutions.

Citation: Albadri, N. Dekeyser, S., & Watson, R. (2017). VennTags: A File Management System based on Overlapping Sets of Tags. In iConference 2017 Proceedings, Vol. 2 (pp. 1-14). https://doi.org/10.9776/17002
Keywords: Operating Systems, File systems, metadata, tagging, Directed Acyclic Graph
Contact: nehad.albadri@usq.edu.au, richard.watson@usq.edu.au, stijn.dekeyser@usq.edu.au

1 Introduction

Personal computer systems, mobile and cloud-based as well as desktop oriented, are permanent companions in users daily lives, for both private and professional activities. Stored collections of files grow steadily as users obtain more files than ever before; whether those files store scientific and experimental data generated by the increasingly sophisticated instruments and computer models, work-related documents, or more personal collections of media and artefacts of our digital life (Lyman, 2003; Alvarado, Teevan, Ackerman, & Karger, 2003; Perišić, 2007). As a result of the increase of the size of the stored collections, the responsibilities that a user faces in storing, organising, and later retrieving files are becoming more complex and problematic. We emphasise our attention on the particular use case where a user stores a file in a file system to which they have write access, and subsequently (perhaps at a much later time) needs to locate and retrieve that file. Variations of the problem exist, such as the file being created by another user, but the key characteristics are that the user knows that the file they are seeking does indeed exist and they have direct access to the file system (unlike the web search case).

Hierarchical file systems (HFSs) have been the standard for personal data management since 1970s (Carrier, 2005). However, HFS is not able to effectively support the tasks that users employ to manage their file collections (Seltzer & Murphy, 2009; Albadri, Watson, & Dekeyser, 2016). This is because (as mentioned above) that the number of personally created or curated files is so enormous that users typically cannot remember where their files are stored (Barreau & Nardi, 1995; Jackson & Smith, 2011) and how they are named, so support for effective searching is vital.
HFS is a single classification systems which means 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. So, in HFS, files normally reside only in one particular directory in the hierarchy (Bergman, Gradovitch, Bar-Ilan, & Beyth-Marom, 2013; Lin, Hao, Changsheng, & Wei, 2014) which leads to problems when a user attempts to build a hierarchy of directories that reflects file properties and supports intuitive search strategies. On most Unix-like operating systems (Shacklette, 2004), symbolic and hard links are available to circumvent the shortcomings of hierarchical (tree) file system structures. The links are special files that contain a reference to another file or directory. They are cumbersome to use for personal file management as the link is not updated whenever the target pointed to by a link is moved,

renamed, or deleted; instead, it points to something that no longer exists. In addition, some applications do not handle links as if they were the objects they point to; but rather resolve the referenced path, using that instead of the link's path. Another related issue is that if the name of the link is changed usually the link is renamed, not its target. The problems that arise from single classification file systems are explored in detail in this paper together with a solution that offers multi-classification.

Numerous attempts have been proposed in order for solving the limitations of HFSs. Some of these proposals rely on a rich collection of file metadata rather than the hierarchical directory structure (A. Ames et al., 2005; S. Ames, Gokhale, & Maltzahn, 2013; Dekeyser, Watson, & Motrøn, 2008; Gifford, Jouvelot, Sheldon, et al., 1991; Rizzo, 2004; Seltzer & Murphy, 2009). These systems are designed to fully replace the HFS. Another approach to dealing with HFS limitations is to introduce extra functionality layered on top of the underlying existing file system. Many such approaches have surfaced, and we can group models and applications in terms of the main technique they employ (see Section 6). However, these attempts have some limitations that prohibit considering them as a solution to the HFS problems.

It is clear that for efficiency and practical reasons (e.g. binary files) a search must be conducted in terms of metadata associated with files rather than their contents (though metadata may be obtained automatically from file content). The structure of file management system metadata, and the services provided to manipulate that metadata, becomes a key contributing factor to the efficacy of any search 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.

In this work, we propose a new model (named *VennTags*) that allows overlapping sets (containers) of files which is isomorphic the idea of *Venn diagram*. In order to achieve that we use the *rooted Directed Acyclic Graph*(*DAG*) instead of the hierarchy (tree) structure. In VennTags model, the containers of files (named collections in this model) might have a plurality of membership (having more than one parent except the root at the same) while in the HFSs the containers (folders/directories) are allowed to have just one parent. In addition, in this novel model, we add tagging capability to the fundamental file management system structure (both collections and files) and a query ability as well to avoid the limitations of HFS. This 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.

Organisation In this paper we will first expose the motivation of this paper by showing the limitations of the HFSs in terms of file management and search which are summarized in Section 2. In Section 3, we identify the main service requirements with some important definitions. The main contribution of this paper is the novel model called *VennTags* (Section 4). Section 5 evaluates the proposed model in terms of the solving limitations of HFS as well as comparison to the other solutions. In Section 6, we explore the work related to our proposal and show the differences and the benefits of our proposed.

Contribution The contributions made in this paper include: named and described problems of the hierarchical file system, proposal of *VennTags* as a file management model to solve the traditional hierarchical file system, allowed overlapping the containers (collections) of files, Directed Cyclic Graph, and tags as a solution for the highlighted limitations, and introduction of a query language as part of the File management System API level to support easy retrieval of files.

2 Motivation Scenarios

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

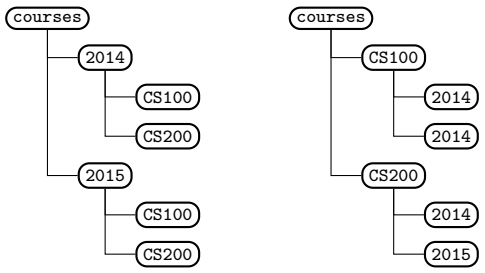

Figure 1: Alternative classification hierarchies

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. Problems of traditional hierarchical file systems have been noted repeatedly in the literature (Seltzer & Murphy, 2009; A. Ames et al., 2005). In our previous research project (Albadri et al., 2016), we detailed HFS problems in this context. The following is a summary of these issues.

1. *Problem 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: these items have properties 'course code' and 'year of offer', but either of the hierarchies as shown in Figure 1 course-code and then year of offer or the second on then course code could justifiably be used to group course files.

2. *Problem 2: Classification* In a hierarchy, items can often belong to more than one sub- tree. 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 2 a and b, keep duplicated copies (Figure 2 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 about that in (Albadri et al., 2016)).

3. *Problem 3: Problematic Pruning*

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

4. *Problem 4: Metadata management*

   In HFSs, bulk updates of metadata are inefficient. For example, if a user wishes to add or remove such meta- data, 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. *Problem 5: Native query support*

   The traditional file system API (e.g. POSIX ("IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Definitions", 2004)) 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.

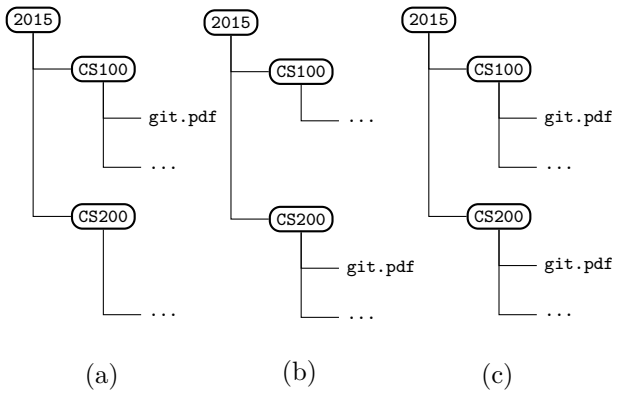

Figure 2: Multiple classification choices

3 Framework

In the following section, our proposed file management system (*VennTags*) design is described by the set of functions exposed at the programming level–the application programming interface (API). We exclude from the description the kind of user level file system related operations provided by applications that layer above the file system API

We further limit our presentation to just the parts of the API that deal with the file management system *organisation*. For example, file content manipulation operations and file management system privilege and protection operations are not handled. In Section 6, we will compare our approach to other research proposals, including those that also use tags.

Our proposed file management system offers three classes of services that are:

1. *Create* a file together with associated metadata.
2. *Identifying files*
   (a) The ability to look up or locate a single file given a metadata based specification. This is needed for the file 'open' operation. The file system metadata must be managed to ensure that every file has a unique metadata specification
   (b) A *query* service that returns a group of files that meet class membership conditions. At the very least, this would include exposing the contents of a collection of files such as a directory. However we extend this notion to a generic file system query. These operations are critical in the development of user interfaces to file systems.
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.

The fundamental physical entities stored in the proposed file system are *tags*, *files*, and *collections* of files. These are defined as follows, together with the related metadata based *path* and *query* concepts.

A **tag** is an item of metadata associated with a collection or a file. It could be represented as an unlimited length text string. This is a generalisation 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 organisation 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 for 'file identifier'.

A file may have associated tags.

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). To offer the idea of the Venn diagram, the collections are organised as a *Directed Acyclic Graph (DAG)*.

A *DAG* is a directed graph that has no cycles so it is a special kind of directed graph while a rooted tree is a special kind of the DAG. The reason of preferring and selecting this kind of structure for this model is that DAG represents the solution of some of the HFS limitation as it provides a multi classification. This is because a collection might have a plurality of membership so the collections can be placed in several different categories at the same time if they are need to. In addition, as this structure has a single root and there is no cycles, the proposed model will provide the benefits of the semi-hierarchy in organising the collections' of files

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).

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 (see path discussion below).

A file in a collection can have a set of associated tags which are directly linked to the file. The file inherits from the containing collection and its parent to be the file path. The key semantic difference is that an atomic operation that affects a collection tag broadcasts to involve many files within that collection while a file tag affects just one file.

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(Jones, Wenning, & Bruce, 2014) paths to find files.

A **query** specifies a search criterion in terms of collection and file metadata (tags). Performing a search based on such a query returns a set of zero or more files which may reside in many different collections.

The file system query is a key divergence from traditional HFS APIs. The functionality offered by a file system query can be duplicated by a client program of a traditional HFS but will likely suffer from poor efficiency due to the need for repeated file system calls

4 VennTags Model

As mentioned in Section 2, HFS is a single classification system which causes a problem in terms of organising the files and re-finding them when needed. So we propose a file management system based on the Venn diagram idea in order to provide a multi- classification model. This means allowing a collection to belong to plurality of other collection with some constraints as will be shown in the following.

VennTags considers a rooted directed acyclic graph where there are a single root as tree but a collection is allowed to have a plurality of membership instead of just having a single membership and using tags instead of names. In addition, we introduce a basic query language at API level as it has benefits in terms of retrieving files and metadata management as well. So the proposed model represents the solution to the problems listed in Section 1 as we will present every component in full detail.

What follows is a detailed formal description (using **Z** notation) of the data model (low level); its associated operations to update the data model; and then the model queries that is the high level of the model.

4.1 Data Model

Collections are organised in a manner that represents a rooted graph. A collection may contain other collections which are called sub-collections. The terms *parent* and *child* naturally describe the relationship

between sub-collections and collections; more generally a unique path exist between any two collections. Paths can be expressed as an ordered list of 0 or more items. There is a special collection which does not have a parent, called *root*. some collections -as mentioned early- might have a plurality of membership, so in this case they do have number of paths.

Collections are district from files. Files are simply associated with a collection in a *belongs-to* relationship. The term 'collection' was chosen to explicitly distinguish the concept from the traditional 'directory' (or 'folder').

1. Graph

   $G \subset cid \times cid \cup \{\tau\}$

   - *cid* is the type of collection identifier.
   - G describes a graph of collection identifiers with root τ.
   - $G(s)$ is the parent of s.
   - Initial value: $G = \emptyset$
   - Constraint: $\forall\, s \in \operatorname{dom} G \bullet (s, \tau) \in G^+$
     All collection identifiers are part of a single rooted graph. This constraint also precludes cycles.

2. Collection tags

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

   - *ctag* is the type of collection tags.
   - 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

   $F : cid \nrightarrow (id \rightarrowtail ftag)$

   - 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: $F = \emptyset$
   - Constraint: $\forall (s_1, f_1), (s_2, f_2) \in F \bullet s_1 \neq s_2 \Leftrightarrow \operatorname{dom} f_1 \cap \operatorname{dom} f_2 = \emptyset$
     A physical file may only be referenced within a single collection.

4. Collection path

   The following two functions are derived from G and S.

   $P : cid \nrightarrow seq\ ctag$
   $D : cid \nrightarrow seq\ cid$

   - A collection path describes the node traversal sequence from the root node to a target collection. $D(s)$ is the collection *identifier* path to s while $P(s)$ is the collection *name* path.
   - A path is defined in terms the function *up*, which is the sequence from a node to the tree root.
     $up\, \tau = \langle \tau \rangle$
     $up\, s = \langle s \rangle \frown up(H(s))$
     $D(s) = rev(up(s))$
     $P(s) = \{(n, S(i)) \mid (n, i) \in D(s)\}$

   The syntax of paths is shown in Figure 3. *CollPath* is the collection path defined by the P function, while a path to a file includes an appended file tag. Except for the mandated trailing slash, and the absence of an unnamed root collection, this is identical to the POSIX style path syntax.

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

Figure 3: *VennTags* Path syntax

4.2 Operations of the *VennTags* Model

The possible operations for this model include the function calls below. Rather than precisely formal definitions, we describe their essential properties. Naturally, any user interface built on top of the API may have different operations that translate to these functions.

1. *CrCollection*(*newcollection*, *parent*) : To add a new collection, the precondition is that the collection does not exist. It has to provide the path where the new collection will be with the set of collection tags which must be unique.

   For instance, considering the scenario associated with a university course: these items have properties 'course code' and 'year of offer', we want to add a new collection {reference,Git} within /{2014,CS200} . To do so, all we need function call

   `CrCollection({reference,Git},/{2014,CS200})` to get the target path. This operation returns *cid* if preconditions are met, or false if not.

2. *CrCollectionLink* (*collection*, *parent*)- create a link between a collection and another collection which refers to create a new membership for a collection. So the collection will have another parent- a new path. The precondition to complete this operation is that the collection and parent have to exist, and there is not link between the collection and the parent before. The semantic of this operation means adding a new path to the collection path to that collection- by allowing to have multiple paths. This operation returns true if preconditions are met, or false if not.

   For instance, suppose that the new collection {reference,Git} that we added in within /{2014,CS200} collection has to exist in /{2015,CS200} collection as well, so all we need

   `CrCollectionLink({reference,Git},/{2015,CS200})` function call.

3. *DelCollection*(*collection*, *parent*) : refers to delete a collection. This can be done if the collection is empty which means that all it sub-collections and files have already been deleted (no sub-collection and files at all) and it has not to be linked to another collection as well. This operation returns true if preconditions are met, or false if not.

   For example, to delete {2015,CS100} collection within Courses collection, we need
   `DelCollection (/{2015,CS100},/courses)` call function which will be false as 2015 collection has subcollections.

4. *DelCollectionLink* ({*collection*}, *parent*): delete a collection link: refers to delete one possible path by deleting one membership of collection with other collection. To complete this operation all we need is the collection and the parent which have to exist and the collection has to have another link with another parent- it means the collection has to have more than one link. The semantic of deleting a link is that it refers to change the collection paths by deleting the collection membership from one path. This operation returns true if preconditions are met, or false if not.

   For Example, if we want to delete existing of {reference,Git},/{2015,CS200} collection in /{2015,CS200}, we just need to `DelCollectionLink({reference,Git},/{2015,CS200})`, so {reference,Git} collection will remove just from /{2015,CS200} so in this case one of {reference,Git} path will be cancelled.

5. *UpCollection*(*operation*, *new value*, *old value*) -Update a collection: the updating refers to change the tag value or the path of the particular collection. The input parameters are: operation that means to type of function call whether it is "move" the collection (changing its location) or, "add", "delete"

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

Figure 4: *VennTags* query language

-changing the set of associated collection tags; old value always refers to path (whether the operation move , add, or delete as the location of the collection needed in all these operations); and new value means to new path (location) if the operation is "move" while it is the new tag value or without value if the operation is "add or delete respectively" The old and new values will be checked where the old value has to exist and the new one has to not exist and it will not affect the locally uniqueness of the collection. This function means that all the sub-collections and files underneath this collection will be immediately changed as well. This function call returns true if preconditions are met, or false if not.

6. *CreFile*($\{new\,file\}$, *parent*) -create a new file: The input parameters of this operation are a tag or 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) has to be locally unique.

7. *DelFile*($\{file\}$, *parent*) refers to delete a file from a collection with precondition that the file exists with providing its tags or part of tags which is uniquely identify the file and its collection as well. This operation returns true if preconditions are met, or false if not.

8. *UpFile*(*operation*, *old value*, *new value*) -Update File: changing a tag value or the path of a specific file with a precondition it has to not affect the uniqueness of the files within its collection. This is done by providing the input parameters that are: operation that means the type of this function call which is either "move" , "add", "delete" tags; rename operation will be expressed by delete 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 to 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 has to not exist and it will not affect the locally uniqueness of the file within the collection. This operation returns true if preconditions are met, or false if not.

4.3 Queries

VennTags model adds a query language to the file system API as mentioned early. Figure 4 shows the abstract syntax of the query language. It extends the path language, which is designed to identify a single file system object, by replacing the collection tag by a disjunctive list of tags and the file tag by a disjunctive list of file tags or their inverse. So for files, either the presence or absence of a tag can identify a file to include in the query result.

A query returns a set of files. All files immediately associated with, as well as recursively contained in, the collections identified by the query's path, are returned in the set. The result set is homogeneous: at first glance it is not possible to determine which file has come from which collection. However, subsequent calls to lower-level operations can retrieve that information.

By adding the query language in the API level, problem 3 of the hierarchy -addressed in §1- is avoided if a user adopts an orienteering-style search using a simple query while descending the hierarchy. The result at each level shows all files in the remaining subtree.

For example, the query `/{2014,CS200}` $\wedge$ `/{2015,CS200}` identifies collection that exists in these both collections while `/{2014,CS200}` $\vee$ `/{2015,CS200}` identifies all collections that in /{2014,CS200} or /{2015,CS200} and in both queries all files located within those collections will return recursively. An other possible quires:

In addition, complex queries are possible, for example:

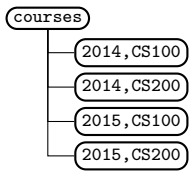

Figure 5: University courses

`courses/2014/CS400` $\wedge$ `CS100/`
`courses/2015/CS400` $\wedge$ `CS100/ reference` $\vee$ `Git`

With concrete query syntax, to perform those complex queries, it would need addition of parentheses to resolve this case, but it has been omitted here for simplicity.

5 Evaluation

The evaluation will be in two parts as shown in the below:

5.1 *How does VennTags solve HFS problems?*

The *VennTags* file management system structure provides a solution to the problems detailed in section 1.

The provision of multiple tags for a collection (file container) offers a solution for problems 1& 2. In the case of the problem 1 (Artificial hierarchies), VennTags solves this problem as shown in Figure 5. About problem 2 (Classification), our model allows multiple classification schemes (problem 2) by two ways: one by adopting *DAG* where a collection can belong to more than one collection at the same time (having more than one parent), and using multi-tags for collection and files. So to solve this problem, the user can create a collection with set of tags that reflects the classification and then link the collection to the desired collections that meet the classification as shown in Figure 6. In addition, more visible collection tags can better inform the orienteering style of search that descends a rooted DAG to locate a file (problem 3). Finally, the ease of tag manipulation supports associating more relevant metadata with groups of files (problem 4).

Multiple file tags can also assist users in the latter two cases: file search and metadata management. Both these are potential benefits dependant to a significant extent on the development of appropriate user interfaces that can exploit the opportunities offered by multiple tags.

It should be noted that, while multiple tags give users better tools to organise consistent file hierarchies, the success any user has in doing so depends on their own ability to suitable tag and categorise the files that they create

The insertion of a generic and powerful tag-based query in the API is a novel feature and one which, like some of the aspects of the tagging structure, depend on appropriate user interfaces to deliver real utility to users In particularly the correspondence between a query result and the concept of a virtual directory or folder (Gifford et al., 1991) can lead to some real advances in GUI-based metadata management and query(Dekeyser et al., 2008). In particular, if a virtual directory is updatable (this corresponds to an updatable base view (Siberschatz, Korth, & Sudarshan, 2011)) a file can automatically acquire the metadata associated with (files in the) virtual directory. So from all these points, VennTags is powerful model that is a solution to the HFS problems

5.2 *Why rooted graph structure?*

To prove that choosing DAG structure provided a more powerful model than other structures, we choose hierarchical (tree) model to compare with as it has been used for long time.So to prove that graph is more expressive than hierarchical (tree) model, consider a graph model where files exist in just one collection and both collections and files have a single tag.

$G \subset sid \times sid \cup \{\tau\}$

Where *sid* is collection identifier.

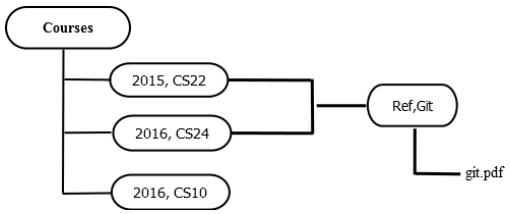

Figure 6: Supporting multiple classifications

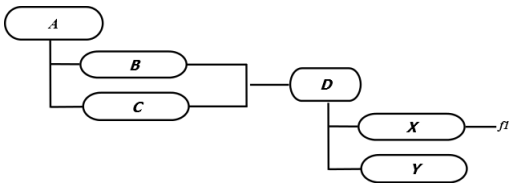

Figure 7: Graph collections

Consider a hierarchical model. Files just exist in one collection and both collections and files have a single tag.

$H : sid \nrightarrow sid \cup \{\tau\}$

THEOREM: Graph is more expressive than hierarchical (tree) model.

PROOF: We show by example that the graph model can exhibit some functionality that tree cannot. Observe that a graph model exhibits the following properties :

1. Preserve metadata.
2. A collection can have more than one parent, so to retrieve a file, there could be more than one path (as defined in section).
3. Metadata updated

For example: if we have a collection that contains other collections -subcollections. The subcollections also contain other subcollections where the files exist, as shown in the in the Figure 7. This figure shows that the collections 'B' and 'C' shared the same collection which 'D'.
So the f_1 file has two paths:

$\langle A, B, D, X\rangle$ $\langle A\ B\ C\ X\rangle$

The abstract view is

$[f_1, f, \{\langle A, B, D, X\rangle\}, \{\langle A, C, D, X\rangle\}]$

If we try to transfer this example to the isomorphic of the tree model, there are alternative structures as shown in the Figure 8. We would arbitrarily choose one collection (B or C) in which to place the shared subcollection (D)(Figure 8 a and b), keep two duplicate copies (Figure 8 c). Both of these structures lead to less metadata and having just one path which mean that both do not meet all the three properties addressed earlier. Other structure could be met one of the key properties which is preserved metadata but non of the others (Figure 8 d). Some of those structures provide one correct path with less metadata while others preserve metadata but uncorrect path. However, non structures could respect the multi-path property which considers the key property.

6 Related Work

As the amount of data stored on personal computers has grown with the limitations of the existing file systems in terms of organizing and retrieving data which are addressed in (§1), number of attempts has focused on finding solution for those limitations. We can categorise the observed attempts to solve the HFS limitations into three groups. These groups are: proposing a new class of file systems by replacing directory

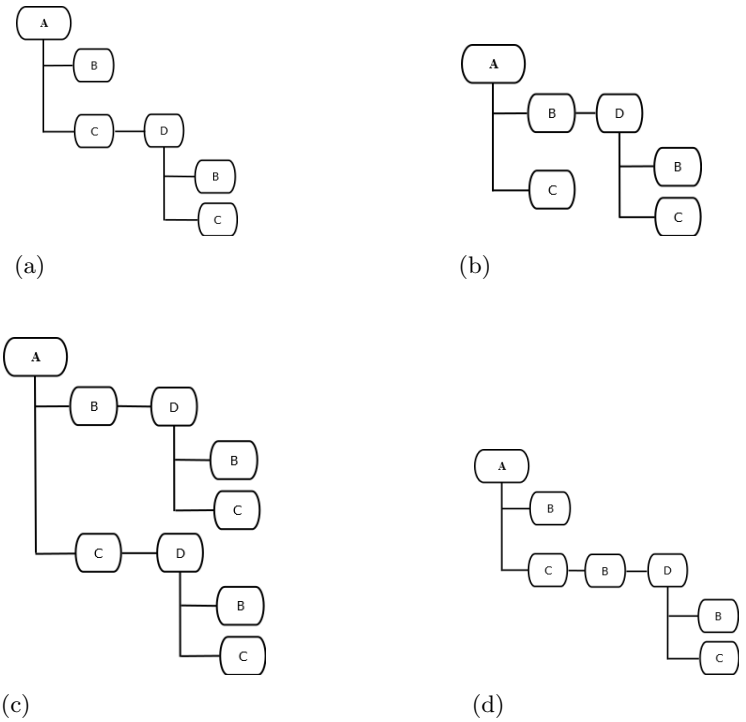

Figure 8: Alternative hierarchy

hierarchies in favour of a more metadata-centric approach, so these systems often do not rely on a physically existing hierarchy; adding on the traditional file system that can be found as part common operating systems have been extended; or enhancing the HFS by doing some changes that can help improving HFS. The details of each group in the subsections below.

6.1 Replacing Hierarchal File Systems

Alternative post-hierarchical file system architectures have been proposed (Gifford et al., 1991; Dekeyser et al., 2008; A. Ames et al., 2005; S. Ames et al., 2013; Seltzer & Murphy, 2009; Rizzo, 2004; Padioleau, Sigonneau, & Ridoux, 2006) to avoid the problems posed by traditional file systems. The organisation and retrieval of files in the cited systems relies on a rich collection of file metadata rather than the hierarchical directory structure. These systems are designed to fully replace the HFS, though as yet none have succeeded in doing so. This is because it might represent a problem for end- users. This is because using a tree is an easy and simple way to classify objects. Users are familiar with tree structures and they can easily understand them. This is an important issue in acceptance and usage of the HFSs by users.

6.2 Extension- adding on HFS

Another approach to dealing with HFS limitations is to introduce extra functionality layered on top of the underlying existing file system. Many approaches involve the use of *tags*. Tags have been used effectively in social websites such as *Flickr* and *YouTube*(Furnas et al., 2006) which have introduced ways of organising multimedia and allowed users to associate tags with media items and then retrieve those items based on metadata (Livia & Ross, 2010).

TagTree (Voit, Andrews, & Slany, 2011) is an example; it takes user-supplied tags and automatically generates and maintains a navigation tree (folder) structure of tags. The system builds an extensive hierarchy such that multiple paths, each one an ordered permutation of the files's set of tags, are generated for each file. This novel system is problematic when files have many tags, leading to exponential growth of the tag tree.

Others (Civan, Jones, Klasnja, & Bruce, 2008; Ma & Wiedenbeck, 2009; Schenk, Görlitz, & Staab, 2006; Bloehdorn & Völke, 2006; Lin et al., 2014; Sajedi, Afzali, & Zabardast, 2012) propose models to help

users to organise their files based on supplied tags. However, the cited systems have not offered a query language so the users cannot easily refind their files.
Our novel model VennTags does also utilise tags as the cited models - as mentioned earlier; however, VennTags differs from other proposed solutions, which are cited above, in many points. One of these points is that the users are allowed to attach the tags to their collections and files as well not just to files as other models. This promotes the utility of multiple tags in classifying, retrieving files, and manipulating files. The another point is that our model allows overlapping collections which solve some problems of HFS as shown in Section 5. The third point is that VennTags provides a query language built in API, so the users can easily re-find their files.

On the other hand there are proposals that also use DAG and tags such as *CoFS* (B.-H. Ngo, Silber-Chaussumier, & Bac, 2008; H. B. Ngo, Silber-Chaussumier, & Bac, 2008). However, in these models supplied tags are automatically generated and maintained in a *DAG* folder structure of tags. The major drawback in this system is that the difficulty of searching to retrieve files which required time for that as the system do not provide a query language for that. In addition, these system lack the metadata management operations to facilitate easy update group/subgroup files because the tags are automatically provided.

6.3 Enhancing HFS

Other proposals to solve HFS problems attempt to find a balance between the HFS replacement (Section 6.1) and the HFS add-on approaches (Section 6.2). *TreeTags* (Albadri et al., 2016) is an example; it is a modest variation to HFS semantics that adds tagging capability to the fundamental file system structure but retains the familiar and clearly useful hierarchical container structure of traditional file systems. However, VennTags is more powerful than Treetags as proved in Section 5.

FindFS (Chou, 2015) and *TrueNames* (Parker-Wood, Long, Miller, Rigaux, & Isaacson, 2014) are other examples that offer an enhancing for HFS problems. However, none of them have succeeded to solve all the HFS limitation, they just focus on just one problem and ignore the others.

Hence from the above subsections, it can be seen that different proposed solutions have been utilized tags and/or DAG but none have succeeded in doing so. These approaches can be grouped into four groups based on their drawbacks. The first group includes proposals that support multi-classification, but they are lack inbuilt query system that facilitates advanced use of such classifications. In the second group of attempts, metadata is managed to some extent by the file system, compared to systems where metadata manipulation is under complete user control. The third 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).

The members of the fourth group use novel non-traditional approaches; some of these may be more correctly seen as data or information management systems.

The above weakness of the cited attempts are avoided in our proposal *VennTags* as shown in section 4.

7 Conclusion

The main work in this paper is to introduce with formal description of a file management system structure that utilize the idea of overlapping sets as in Venn diagram and reuse tags but integrates it into the tried-and-trusted rooted graph paradigm. *VennTags* has been shown to resolve the identified HFS problems.

There are two broad directions that extend the current work. The first one is to continue investigating alternative models that also solve these problems, such as TreeTags (our previous work (Albadri et al., 2016)) but have symblic/hard links without any limitations that exist in HFS links. We are currently evaluating a model that allows files to exist in multiple collections – links, in essence, but without the problems associated with managing them, and then comparing that with VennTags and show which one more powerful model.

The second direction for future work involves evaluating the *VennTags* model in a practical sense. A proof-of-concept implementation must make key decisions on data structures and algorithms; comparing the software with traditional file systems then requires the creation of a metadata-oriented benchmark that

could also be used to measure the efficacy of other novel file systems. Perhaps more importantly the user interface design space afforded by the richer metadata and query API of the *VennTags* file system needs to be explored, prototyped and experimentally evaluated.

References

Albadri, N., Watson, R., & Dekeyser, S. (2016). TreeTags: bringing tags to the hierarchical file system. In *Proceedings of the australasian computer science week multiconference, canberra, australia, february 2-5* (p. 21).

Alvarado, C., Teevan, J., Ackerman, M. S., & Karger, D. (2003). Surviving the information explosion: How people find their electronic information.

Ames, A., Bobb, N., Brandt, S. A., Hiatt, A., Maltzahn, C., Miller, E. L., ... Tuteja, D. (2005). Richer file system metadata using links and attributes. In *Proceedings of the 22nd ieee/13th nasa goddard conference on mass storage systems and technologies* (pp. 49–60).

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*, 159–183.

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

Bergman, O., Gradovitch, N., Bar-Ilan, J., & Beyth-Marom, R. (2013). Folder versus tag preference in personal information management. *Journal of the American Society for Information Science and Technology*, *64*(10), 1995–2012.

Bloehdorn, S., & Völke, M. (2006). TagFS-tag semantics for hierarchical file systems. In *Proceedings of the 6th international conference on knowledge management (i-know 06)*.

Carrier, B. (2005). *File system forensic analysis* (Vol. 3). Addison-Wesley Reading.

Chou, J. (2015). *FindFS: adding tag-based views to a hierarchical filesystem* (Unpublished master's thesis). University of British Columbia.

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. *Proceedings of the American Society for Information Science and Technology*, *45*(1), 1–13.

Dekeyser, S., Watson, R., & Motrøn, L. (2008). A model, schema, and interface for metadata file systems. In *Proceedings of the thirty-first australasian conference on computer science-volume 74* (pp. 17–26).

Furnas, G. W., Fake, C., von Ahn, L., Schachter, J., Golder, S., Fox, K., ... Naaman, M. (2006). Why do tagging systems work? In *Chi '06 extended abstracts on human factors in computing systems* (pp. 36–39). New York, NY, USA: ACM.

Gifford, D. K., Jouvelot, P., Sheldon, M. A., et al. (1991). Semantic file systems. In *Acm sigops operating systems review* (pp. 16–25).

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

Jackson, T. W., & Smith, S. (2011). Retrieving relevant information: traditional file systems versus tagging. *Journal of Enterprise Information Management*, *25*(1), 79–93.

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

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*, 21–48.

Lyman, P. (2003). *How much information?* http://www. sims. berkeley. edu/research/projects/how-much-info-2003/.

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).

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 furture in computing & communication technologies, hochiminh, vietnam* (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).

Parker-Wood, A., Long, D. D. E., Miller, E., Rigaux, P., & Isaacson, A. (2014). A file by any other name: Managing file names with metadata. In *Proceedings of international conference on systems and storage*

(pp. 3:1–3:11). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2611354.2611367 doi: 10.1145/2611354.2611367

Perišić, Z. P. (2007). Too much information. *Review of the National Center for Digitization*, 68–70.

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

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.*

Schenk, S., Görlitz, O., & Staab, S. (2006). TagFS: Bringing semantic metadata to the filesystem. In *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.*

Shacklette, M. (2004). Unix operating system. *The Internet Encyclopedia.*

Siberschatz, A., Korth, H., & Sudarshan, S. (2011). *Database system concepts.* McGraw Hill Companies. (6th edition)

Voit, K., Andrews, K., & Slany, W. (2011). TagTree: Storing and re-finding files using tags. In *Information quality in e-health.* Springer.