TagFS: A simple tag-based filesystem

Scott Bezek
sbezek@mit.edu
Raza (R07)
6.033 Design Project 1
March 17, 2011

1 Introduction

TagFS is a simple yet effective tag-based filesystem. Instead of organizing files and documents in a strict hierarchy (like traditional filesystems), TagFS allows users to assign descriptive attributes (called *tags)* to files and subsequently locate those files by searching for tags of interest.

This design has several motivations. First, the ubiquity of internet search engines demonstrates that users prefer to search, rather than browse, for information. TagFS avoids cumbersome directory traversal by allowing direct searches for files. Second, a tag-based system avoids the problem of categorization: how should a user determine how to organize a set of files? For example, photos might be organized by year, by event, by location, or perhaps even by the people that appear in them. TagFS allows the user to conveniently find files using any criterion (or combination thereof).

2 Design

The TagFS design has two major components: *tags* and *scopes*. A *tag* is simply a free-form string that describes some file A *scope* is a way of finding and grouping files by searching for specific tags. The following sections describe the behavior and implementation of these components.

2.1 Tags and Files

In order for a tag-based filesystem to be useful, tags that have been assigned to a file must persist across system reboots and crashes. For this reason, TagFS stores tag associations alongside file data in non-volatile memory. TagFS adopts the 64-bit file and device numbering scheme proposed in [1].

2.1.1 Tag and File Storage

Each storage device contains four sections: the Tag List contains tags and their associated list of File IDs (FIDs); the Tag Tree allows for quick tag lookup by name and links to elements of the Tag List; the File Node section maps FIDs to data blocks; and the Data Block section contains the contents of files.

The Tag List is a block of memory split into numbered tag nodes. Each tag node represents a tag on the device, and holds tag metadata. The tag nodes also provide a compact numbering scheme (Tag IDs) for referencing a particular tag. A tag node contains the tag name (fixed length string), a count of files containing that tag, a fixed-length array of FIDs, and a pointer to a data block that may contain a longer array of FIDs if necessary. The data block may also link to another data block (forming a linked-list) if even more space is needed to hold the entire file list. Since each tag node has a fixed size, a tag node's location on disk can be determined by the Tag ID. The Tag List section contains a bitmap of Tag ID availability so that new tags can reuse deprecated Tag IDs. Figure 1 shows a diagram of the Tag List.

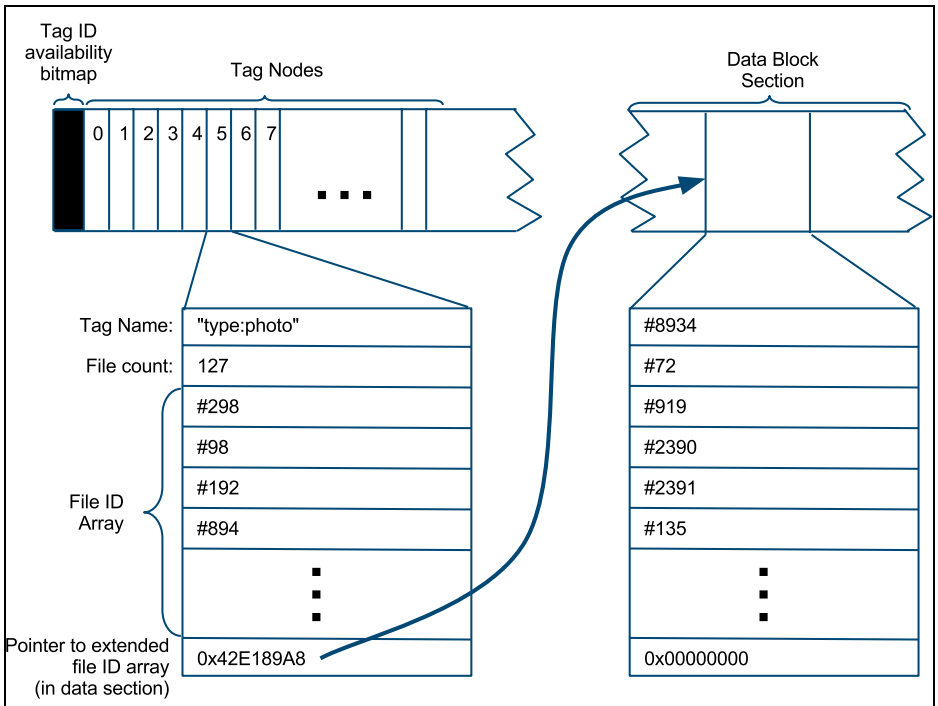

Figure 1: The layout of the Tag List on disk.

The Tag Tree is a B+ Tree that allows for a quick tag search across all files on the device. The keys of the tree are the string tags of all files on the device, and the leaf-nodes point to the tag node in the Tag List. Thus, given a search tag, all FIDs that contain that tag can be found quickly (in $O(\log n)$ time, where n is the number of files on the device) by searching the Tag Tree and then reading the tag node.

For file storage, TagFS uses the inode design as described in [2] with a few modifications. TagFS does not need the link-count or file code fields of the inode (which relate to directory structure), so these are replaced by a pointer to a data block containing the file's associated tag IDs. That block may link to another data block with more tag IDs if necessary (forms a linked list). Files on different devices may have the same FID/inode-number, so a file is uniquely identified by the tuple (DeviceID, FileID).

2.1.2 Tag and File API

TagFS implements the following functions to manage files and tags:

- **`create(deviceID)`** – allocates a free inode on the device and returns the FID
- **`copy(deviceID, fileID, destinationDeviceID)`** – creates a new file and copies the contents of the existing file
- **`delete(deviceID, fileID)`** – deletes a file (removes appropriate entry from Tag Tree)
- **`read(deviceID, fileID, offset, length)`** – reads *length* bytes from file at the specified byte offset
- **`write(deviceID, fileID, offset, data)`** – writes *data* to the file at the

specified offset, overwriting existing data and extending the file as necessary

- **`tag_add(deviceID, fileID, tagName)`** – adds a tag to a file. Finds the tag ID of tagName by searching the Tag Tree, or else uses a free Tag Node and adds the tag to the Tag Tree. Adds fileID to the Tag Node's file array. Adds the tag ID to the inode's tag list. Also updates scope caches as necessary (see 2.2.3).
- **`tag_remove(deviceID, fileID, tagName)`** – removes a tag from a file. Finds the tag ID of tagName by searching the Tag Tree, and removes the element from the tree. Removes fileID from the Tag Node's file array, and releases the Tag Node if its file array becomes empty. Removes the tag ID from the inode's tag list. Also updates scope caches as necessary (see 2.2.3).
- **`tag_list(deviceID, fileID)`** – returns the list of string tags assigned to fileID by looking up the tag ID list in the appropriate inode and cross-referencing string names from the appropriate Tag Nodes.
- **`device_list()`** - returns the list of device IDs of all connected devices. Device IDs are stored in a device map (mapping device IDs to hardware IDs) which is maintained by the OS.

2.2 Scopes

A *scope* is a dynamic group of files found by searching for a given set of tags. Scopes can be thought of as operators on lists of files: a scope accepts one or more lists of files as input, merges those lists, filters the merged list using the specified tags, and outputs the resulting list. This structure can be seen in Figure 2. For example, photos on HDD0 can be found by using a scope with input=HDD0 and filters="type:photo".

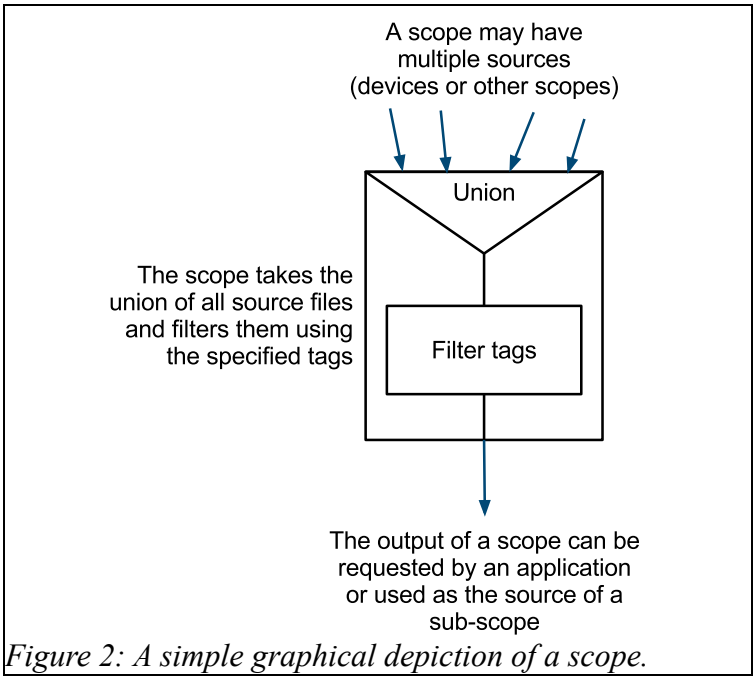

Figure 2: A simple graphical depiction of a scope.

Scopes can be chained together to select complex groups of files – the output of one scope may feed into another. For example, suppose iTunes wants to find all of Scott's photos and videos. First find all *photos* owned by Scott using one scope, find all *videos* using another scope, then merge the two lists using another scope. This chained scope construction is shown in Figure 3.

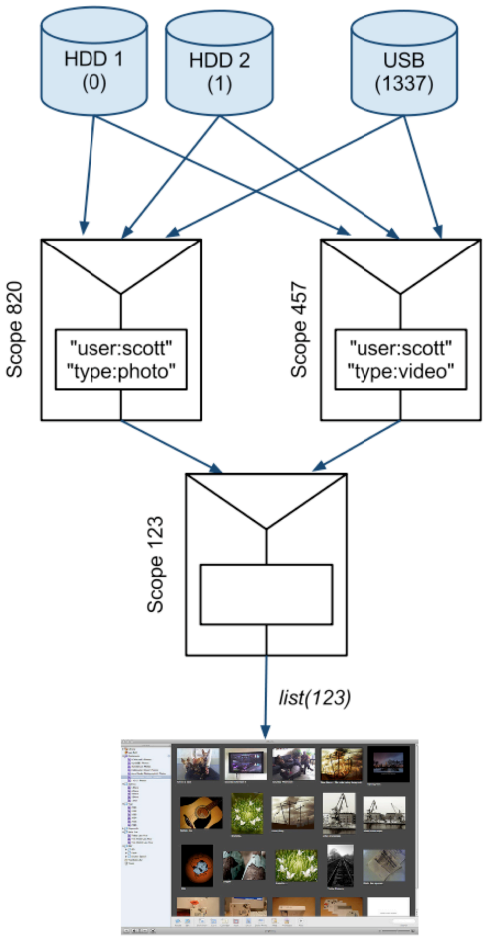

Figure 3: An example scope configuration to find photos and videos owned by Scott on HDD 1, 2 or a USB device.

2.2.1 Scope Structures

TagFS stores scopes in system memory which allows scopes to easily refer to multiple devices and provides fast creation/search speed. Scopes need not persist across system restarts since they can easily be recreated. Furthermore, scopes are "lazy" - a scope does not compute its output file list until explicitly requested. Thus, scopes will always return the most up-to-date file list.

The most basic scope structure contains a list of source (scope/device) IDs and a list of tag filters. Both of these lists may be stored as HashSets to avoid duplicates. Consider the example from Figure 3: if an application requests the files in scope 123, the filesystem would look up the list of sources (scopes 820 and 457), recursively request the files, and merge and filter those results.

In order to find a particular scope by ID, the filesystem maintains a mapping from 64-bit scope IDs to scope objects in memory. TagFS uses a hash table called *ScopeLookup* to provide this mapping with fast $O(1)$ amortized lookup time. Figure 4 shows an example of this basic scope representation, using the same scope setup as Figure 3.

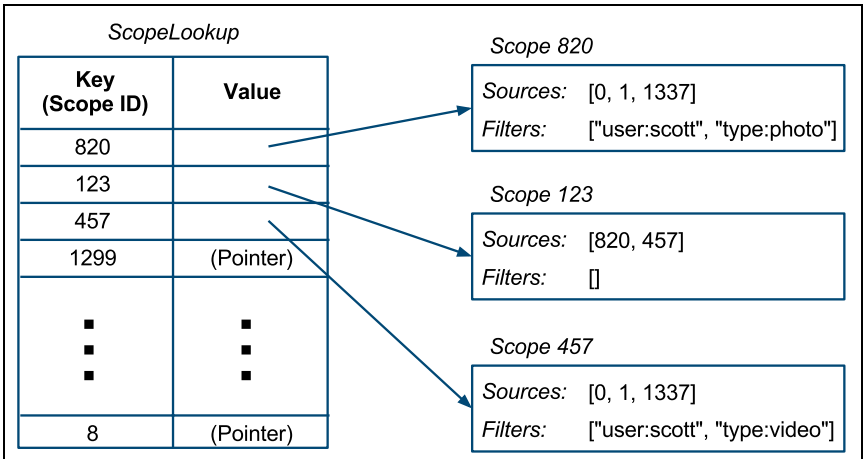

Figure 4: An example of a basic scope data structure.

2.2.2 Scope Behavior

The process by which TagFS merges and filters file lists requires particular care in order to implement TagFS efficiently. Depending on the type of source, the merging and filtering algorithms behave differently.

If the source is a device, the device's Tag Tree should be used to find sets of files corresponding to each tag. The intersection of these sets (which can be computed quickly in memory) represents the files that match *all* specified filter tags. By using the set-intersection technique, it is not necessary to seek to each inode on the disk to verify the file's tags.

If the source is another scope, TagFS will recursively request its file list. If multiple sources are scopes, TagFS will compute the set union of their returned file lists (using a HashSet) to eliminate duplicates. TagFS then looks up the tags associated with each file in the union (by looking at the inode) and checks if all filter tags are present. This operation requires $O(n)$ device seeks, where n is the number of files being searched. In practice, these sub-scopes generally search a small number of files (since their parent scopes reduce the search space) so this is not an issue. However, if scope chains are poorly constructed, as in Figure 5, performance can suffer.

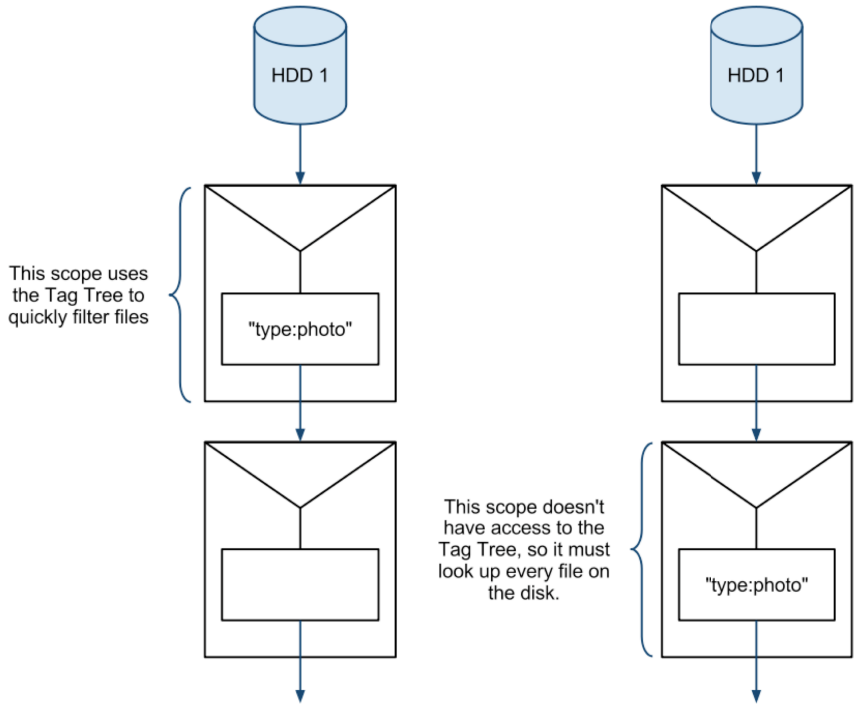

Figure 5: Performance comparison of scope chains. Both scope chains return the same list of files, but the left chain only takes $O(\log n)$ time whereas the right chain takes $O(n)$.

2.2.3 Scope Caching

Although the basic scope construction (sections 2.2.1/2.2.2) functions correctly, it would perform poorly in practice, as each time a scope is requested, the filesystem must rebuild the file list by searching the disk. To avoid this, TagFS maintains scope caches of (DeviceID, FileID) tuples along with a *cacheValid* flag.

However, there are several important changes that must be made for a cache to function properly. First, if a scope's cache is invalidated (by changing filters or sources), the cache invalidation must propagate to all sub-scopes. In order to do this, each scope links to sub-scopes, enabling scope traversal. Second, tagging a file may affect scopes watching that tag. However, since only one file changed, TagFS can just update the cache instead of invalidating it. To find all scopes affected by a tag change, TagFS keeps a global hash table, *TagScopeRefTable*, mapping tag names to directly affected scopes, and the sub-scope pointers can be traversed to update the sub-scope caches as well (Figure 6).

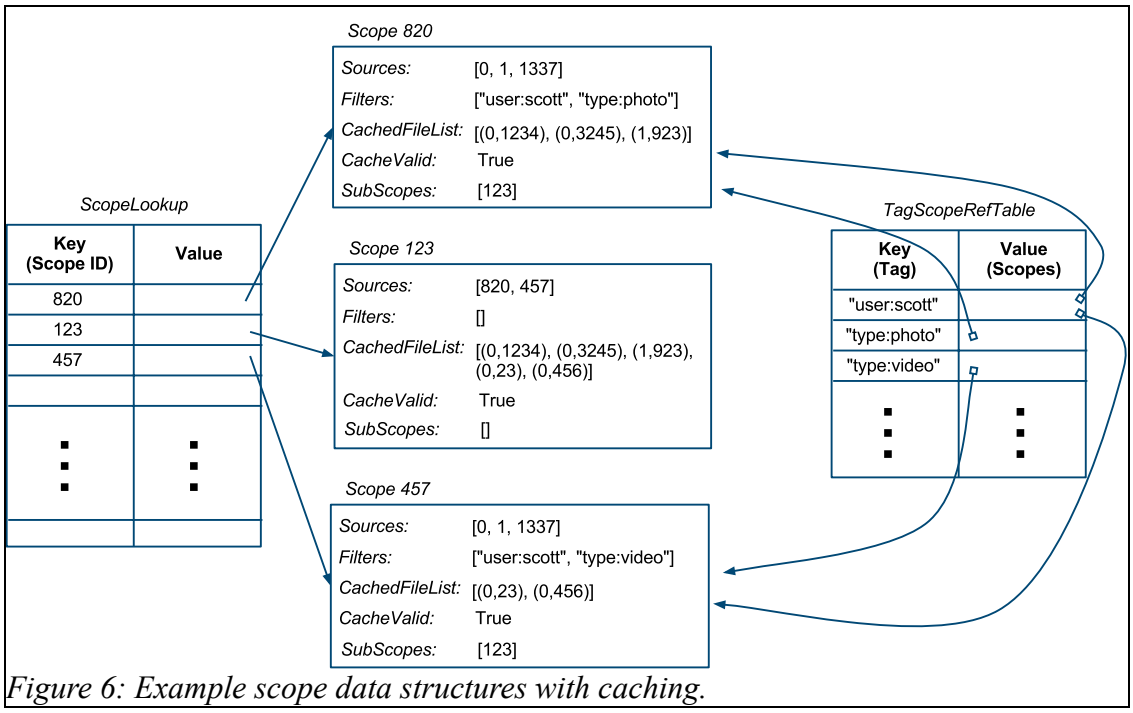

Figure 6: Example scope data structures with caching.

2.2.4 Scope API

TagFS implements the following functions for applications to manage scopes:

- **`mkscope()`** - creates a new scope in memory, adds it to *ScopeLookup* and returns the scope ID
- **`scope_add_source(scopeID, sourceID)` / `scope_rm_source(scopeID, sourceID)`** – add/remove source to/from scope's source list. Invalidates caches (see 2.2.3). Checks to prevent scope cycles.
- **`scope_add_filter(scopeID, tagNameList)` / `scope_rm_filter(scopeID, tagNameList)`** – add/remove one or more tag filter. Invalidates caches (see 2.2.3)
- **`list(scopeID)`** – get the file list of the scope (refresh cache if necessary) – see 2.2.2.

2.3 Conventions

There are many standard tags to improve application interoperability, including:

- "type:" - filetype
- "user:" - file owner
- "application:" - shared by all application files
- "version:" - identifies the version of file/application
- "year:" - date
- "location:" - place

By using standard tags, multiple programs can easily interact with the same set of files, and files can be organized more easily. In addition, applications and their dependencies are packaged

together by applying an “application:” tag. For example, Photoshop and all of its dependencies would share the tags “application:photoshop” and “version:4.9” to make them easy to locate.

3 Analysis

The following sections describe possible scenarios that demonstrate how TagFS can be used.

3.1 Photo Viewer

Suppose a photo viewer is running and displaying the user's own photos, and the user plugs in a USB drive with pre-tagged photos and wants to display those along with the current set.

The photos being displayed have already been tagged “type:photo” and “user:scott” and are found using a scope setup like Figure 3. The photo viewer periodically checks `device_list()` and notices the new USB device. After asking the user if he wants to display photos from the device, the application creates a new scope, setting the old scope as one source and the USB device as another source, with a filter for “type:photo.” The application calls `list()` to request which files to display.

3.2 Photo Viewer, Part II

Suppose the user now wants to show all photos (on any device) taken in Colorado in 2007.

The application creates a new scope for the search, adding “type:photo” as a default filter. The user selects which devices to search (all of them) and the application adds those device IDs as sources. As the user selects search criteria, the application adds the appropriate filters to the scope - “location:colorado” and “year:2007”. These changes are instantaneous because scopes are lazy. When the user clicks “Search,” the applications calls `list()` on the scope, which follows the procedure of 2.2.2 to find the photos.

3.3 Photo Viewer, Part III

Suppose the user inserts a read-only CD, containing photos without assigned tags.

The photo viewer notices the CD (by polling `device_list()`), and asks the user about displaying photos from it. If the user clicks “yes,” the viewer creates a scope with the CD as the source and the filter “type:photo.” However, `list()` returns no photos, so the viewer asks “Search for untagged photos?” If the user clicks “yes,” the viewer removes the “type:photo” filter, and analyzes each file to determine if it is a photo. If so, the application keeps track of its FID and displays it.

3.4 Copying Files

Suppose the user wants to copy the photos from the CD to a new USB drive.

The user opens a file browser window, which displays the available devices using `device_list()`. The user selects the USB drive. The browser calls `list()` and displays the files. The user switches back to the photo viewer and drags the photos onto the file browser window, invoking a drag-and-drop handler in the browser. The browser is passed a list of

(DeviceID, FileID) tuples, and the browser uses `copy()` for each one to be copied to the USB device.

3.5 Shell
Suppose the user types a command like "python" at a shell prompt.

The shell relies on the SYSTEM_DEVICES environment variable to determine which device IDs to search for executables. When the user types "python" the shell creates a scope with sources from SYSTEM_DEVICES and the filters "type:executable" "version:current" and "application:python." Since the shell uses SYSTEM_DEVICES, it will not be confused if Python exists on a USB drive. If `list()` returns multiple results, the shell can ask the user which version to run.

3.6 Libraries
Suppose the user runs a Python interpreter, which must load associated Python modules.

Application dependencies are packaged by sharing the "application:" tag. The shell creates and passes the "application scope" to Python which filters SYSTEM_DEVICES for "application:python2.6". Python creates a sub-scope of the application scope and filters for "type:python_module", "version:current", and the module name: "numpy".

3.7 Multiple Application Versions
Suppose the user has two versions of Python installed with distinct modules.

As discussed in 3.6, the applications will be tagged "application:python2.6" and "application:python3.0" along with all associated dependencies. The shell passes the application scope to Python, which can create a sub-scope of the application scope using the filter "type:pyc" to find the appropriate compiled Python modules.

4 Conclusion
TagFS is a groundbreaking tag-based filesystem that was designed with usability and efficiency in mind. The system centers around a tagging system that can be searched quickly and flexibly. By introducing the concept of scopes, TagFS allows applications to quickly find files by narrowing the search space with a series of filters. Furthermore, the simplicity of scopes allows them to be chained together into specific search queries without sacrificing speed. Finally, by implementing a basic caching system, TagFS can scale well by avoiding unnecessary disk accesses. These features combine to make TagFS a unique but practical modern filesystem.

References

[1] B. Bitdiddle, "6.033 2011 Design Project 1," [Online document], 2011 Feb 23, [cited 2011 Mar 16], Available HTTP: http://web.mit.edu/6.033/www/assignments/dp1.html

[2] D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System," Communications of the ACM, vol. 17, no. 7, 1974.

Word Count: 2681