Tag-based File System

6.033 Design Project 1
Usman Masood
Strauss 1pm
March 22, 2010

1 Overview

This paper outlines a design for a file system which uses string-based tags to organize and retrieve files. User can run search queries using tags to group files into *scopes*. Scopes contain a dynamically updating view of files matching the search criteria. This allows users to add and remove tags freely.

The proposed system uses B+ trees to organize tags, special directory files to maintain tagged file sets, and reuses the inode layer of the UNIX file system. The primary goals of the design are: high performance under different workloads, and simplicity to facilitate implementation and testing.

2 Design Description

The system uses on-disk B+ trees to index tags. Scopes are represented as linked lists and are lazily evaluated by the *scope evaluator*. To enhance performance, the scope evaluator caches evaluated scopes and uses set cardinalities to determine how filter operations on file sets are carried out. A *daemon* listens for new devices and exposes a pub/sub interface to user-level programs.

The next sections explain the system design. Section 2.1 outlines the data structures used. Section 2.2 describes helper modules. Section 2.3 discusses performance optimizations. Section 2.4 describes the API implementation.

2.1 Data Structures

2.1.1 Representation of Tags

The system allows users to label files with free form tags of up to 140 characters. Tags are not case sensitive. Each device contains an on-disk B+ tree that indexes tags. Each node in the B+ tree represents a unique tag and points to a directory file which lists inode numbers of files labeled by that tag. Figure 1 illustrates this organization of files.

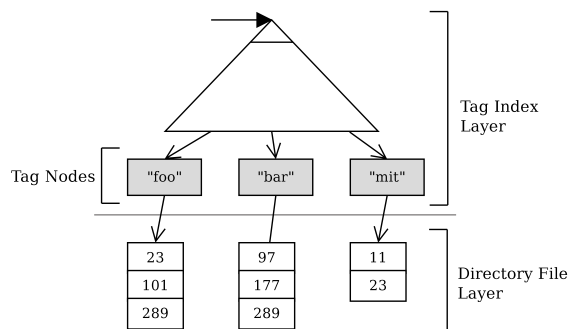


Figure 1: On-disk B+ tree to index tags

2.1.2 Representation of Scopes

There are two types of scopes: basic scopes and views. Basic scopes represent physical storage devices; views represent ephemeral scopes which users can populate using existing scopes. The `search(srcId, tags, destId)` system call finds the set of files in the `srcId` scope labeled with all tags in list `tags` and unions it with the set of files already in the `destId` scope. We represent scopes as a linked list of `FilterNodes`.

```
structure FilterNode
{
        long sourceScopeId;
        List<String> tags;
        FilterNode *next;
}
```

Figure 2: Pseudocode declaration for the `FilterNode` structure.

Each node in the linked list represents a filter operation on the `sourceScopeId` scope using the tags in `tags`. The results of all nodes are joined to populate the scope. A hash-table (called *view table*) is maintained in memory which maps views to their linked list representations. Figure 3 describes how search calls can be used to create such linked list representation of scopes.

The following sequence of system calls shows how scopes can be modeled using lists:

1. `mkscope()` // returns an empty scope S10

   $S10$:[ ]

2. `search (S1, a, b, S10)`

   $S10$: $[(S1 :< a, b >)]$

3. `search (S3, p, q, S10)`

   $S10$: $[(S1 :< a, b >), (S3 :< p, q >)]$

Meta-language

- $< a, b >$ represents a list of tags a and b.
- $(S1, < a, b >)$ is a node that represents a search call. It encapsulates a filter operation on scope $S1$ using tags a & b.
- $[x, z]$ represents a linked list of nodes x and y.

Figure 3: A linked list representation of scopes.

Only views can be mutated by a `search` call, i.e. basic scopes cannot be used as destination scopes in `search` calls. Users can group files, even from different devices, by creating scopes that derive from other scopes recursively.

Scopes are lazily evaluated; we only evaluate a scope to generate the set of files it contains when the `list()` is invoked on it. Furthermore, scopes are dynamically updating: if a scope is modified, all other scopes that derive from it are modified as well. For instance, if *S1* in Figure 3 is modified, the change will be propagated to *S10* as well.

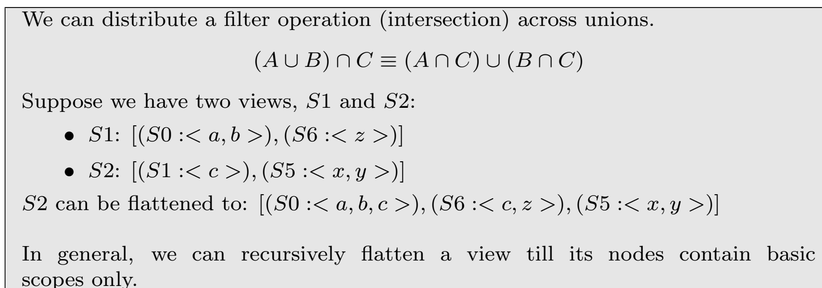
2.1.3 Inode Structure Augmentation

Files are represented using UNIX inodes. The inode structure is augmented to store a pointer to a linked list which stores all tags associated with the file. Directory files now store only inode numbers for files they contain.

2.2 Helper Modules

2.2.1 Scope Evaluator

Since scopes are represented as unions of filter operations, any arbitrarily complex scope can be flattened into filters and unions on basic scopes only (see Figure 4).



We can distribute a filter operation (intersection) across unions.

$$(A \cup B) \cap C \equiv (A \cap C) \cup (B \cap C)$$

Suppose we have two views, $S1$ and $S2$:

- $S1$: $[(S0 :< a, b >), (S6 :< z >)]$
- $S2$: $[(S1 :< c >), (S5 :< x, y >)]$

$S2$ can be flattened to: $[(S0 :< a, b, c >), (S6 :< c, z >), (S5 :< x, y >)]$

In general, we can recursively flatten a view till its nodes contain basic scopes only.

Figure 4: Nested views can be flattened into filters and unions on basic scopes.

The scope evaluator generates the set of files in a scope by calling the `evaluate()` function (see below).

```
evaluate(long scopeId)
{
        // Check scope cache for scopeId (see Section 2.3.2)
        1.  If scope cache has valid entry for scopeId, return it.

        // Otherwise, evaluate scopeId
        2.  Get the linked list representation of scopeId from the view
            table.
        2.  Flatten the representation retrieved in (2).
        3.  For each node in the flattened representation:
            a.  Fetch the directory file for all tags in the node.
            b.  Generate the intersection of file ids between the
                directories obtained in (3a).
        4.  Join results for all filter operations carried out in (3).
        5.  Store the result in the scope cache and return it.
}
```

Figure 5: Pseudocode of `evaluate()`.

The scope evaluator tries to optimize the filter operation for high performance (see Section 2.3.3).

2.2.2 Device Daemon

A daemon service runs in the background and listens for new devices. It exposes a publish/subscribe interface to user-level applications. Applications can use this interface to subscribe to tags they are interested in. When a new device hosting our native file system is plugged in, the daemon scans its B+ tree and notifies all applications whose subscribed tags appear in the B+ tree.

When a device containing a foreign file system (e.g. CD), the daemon constructs a new in-memory B+ tree for the foreign file system and indexes all files by generating tags using its naming layer. For instance a file located at "foo/bar/abc.xyz" can be tagged with "foo", "bar", "file_name:abc" and "file_extension:xyx". This ensures that duplicate file names can be distinguished by their path's tags. Paths can also contain some semantic meaning, e.g. files in a directory "photos" are likely to be photos. This tagging scheme preserves such semantics. The scope for such devices points to this in-memory B+ tree.

2.3 Performance Optimizations

2.3.1 Directory File Format

Directory files should store inode numbers in sorted order so that intersection can be done in linear instead of quadratic time. However this also increases insertion overhead from constant time to $O(\log n)$. An alternative is to store inode numbers in a hash-map to achieve $O(n)$ intersection and $O(1)$ insertion time (expected), but the complexity introduced by such a data structure might be unfavorable.

2.3.2 Caching Scopes

We will introduce three new data structures to implement caching for scopes:

1. *scope cache*: maps scope ids to the set of files they contain.
2. *scope dependency graph*: a directed graph which models dependencies between scopes: an edge from A to B represents the relationship that scope B derives from scope A.
3. *tag reference table*: a hash-table which maps `(deviceId, tag)` keys to a list of scopes which reference `tag` on `deviceId`.

The cache coherence protocol works as follows:

- Whenever `evaluate(scopeId)` is called, it caches its result in the scope cache if the entry for `scopeId` is absent or invalid.
- Whenever `search(sourceId, tags, destinationId)` is called:
    1. Cache entry for `destinationId` and all scopes reachable from it in the scope dependency graph are invalidated.

2. If `sourceId` is a view then an edge is added from `sourceId` to `destinationId` in the scope dependency graph. Otherwise, for every tag in `tags`, `destinationId` is added to the entry for `(sourceId,` *tag*`)` in the tag reference table.

- Whenever `tag_add(deviceId, fileId, tag)` or `tag_remove(deviceId, fileId, tag)` is called, it invalidates all scopes in the entry for `(deviceId, tag)` in the tag reference table. To invalidate scopes which derive from these scopes, all scopes reachable from them in the scope dependency graph are invalidated as well.

2.3.3 Optimized Filter Operation

Rather than evaluating intersections between directories in a linear left-deep manner, our system uses bushy evaluation trees like many RDBMS do (see Figure 6). This helps reduce the size of sets to intersect and provides opportunity for parallelization, e.g. parallel reads from different devices. Directories with fewer files are pushed down (intersected earlier) to reduce the size of the working set rapidly.

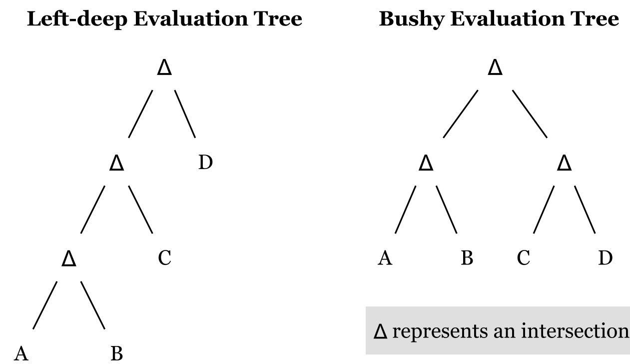


Figure 6: Using bushy evaluation trees for intersections.

If the cardinality of the two file sets being intersected is close, we evaluate the intersection by doing a linear scan. On the other hand, manually checking the tags of files in the smaller set will be faster if one of the sets is much smaller than the other.

When the size of the working set becomes small enough, the system just manually checks the tags of all the files in it to filter for remaining tags.

2.3.4 Handling Large Number of Tags

If the number of tags in a B+ tree becomes too high, the system creates hash buckets for tags where each bucked stores its tags in a separate B+ tree. This

ensures that the size of a single B+ tree does not become too large.

2.4 API Implementation

2.4.1 search(sourceId, tags, destinationId)

It creates a new `FilterNode` from `sourceId` and `tags` and prepends it to the linked list entry for `destinationId` in the view table. To adhere to the scope cache coherence protocol, `search()` also modifies some cache-related data structures as described in Section 2.3.2.

2.4.2 list(scopeId)

It calls `evaluate(scopeId)` on the scope evaluator and returns the result as a list of `(deviceId, fileId)`.

2.4.3 create(deviceId)

A new file inode is created in the inode table of `deviceId` and its inode number is returned.

2.4.4 delete(deviceId, fileId)

It first looks up the tags associated with `fileId`, and calls `tag_remove()` for each one of them. This ensures all scopes which might contain `fileId` are invalidated. The inode and data segments of `fileId` are then freed.

2.4.5 read(deviceId, fileId, buffer, count, offset)

`read()` is implemented using existing UNIX system calls (see Figure 7).

```
read(deviceId, fileId, buffer, count, offset)
{
    1.  Open file with inumber = fileId on device with id = deviceId.
    2.  Use lseek() system call to move cursor to required offset.
    3.  Use UNIX read() system call to read into buffer.
}
```

Figure 7: Pseudocode for `read()`.

2.4.6 write(deviceId, fileId, buffer, count, offset)

The implementation of `write()` is analogous to `read()`.

2.4.7 mkscope()

It generates a new unique scope id, creates a new entry for it in the view table and maps it to an empty linked list.

2.4.8 tag_add(deviceId, fileId, tag)

It finds the node for `tag` in the B+ tree in `deviceId`, and adds `fileId` to the directory file it points to. If `tag` has not been previously used, a new node is created for it and inserted into the B+ tree. The `tag` is also added to `fileId`'s inode. It invalidates modified cached scopes as explained in Section 2.3.2.

2.4.9 tag_remove(deviceId, fileId, tag)

It does the converse of `tag_add()`.

2.4.10 tag_get(deviceId, fileId)

It looks up the inode of `fileId` in `deviceId` and returns the list of tags it points to (see Section 2.1.3).

2.4.11 device_list()

It queries the device daemon for the set of storage devices currently plugged-in.

2.4.12 Interface for Device Daemon

User-level processes communicate with the device daemon via IPC. The following pseudocode describes its high-level interface.

```
class DeviceDaemon
{
      // Maps tags to the process which have subscribed to them
      Map<Tag, List<ProcessId>> subscriptionsMap;

      // Returns whether the process successfully subscribed to the
         provided tags
      boolean subscribe(List<Tag> tags, int processId);

      // Notifies the process that the device plugged-in contains
         relevant tags
      void notify(int processId, long deviceId);

      // Returns the scope ids of all plugged-in devices (including)
         foreign file systems)
      List<long> get_device_list();
}
```

Figure 8: Interface of device daemon.

3 Putting It All Together

There are some tagging conventions in our file system which all applications must adhere to. File attributes are tagged to files with "file_*attribute*:*value*".

All files must have a *name* and an *extension* attribute tag. All system files are tagged with "system:*attribute*". Applications tag their files with "*application*:*attribute*". If applications allow multiple versions to exist on the same system, then to prevent conflicting tags they should append their version ids to *application*.

When a user logs on, a default scope is created to represent the user's own files. It is populated by running a search for files tagged with "file_owner:username". Whenever the user creates a new file, it is automatically tagged with his user id.

Applications search the user scope to obtains user files such as photos and documents. For instance, a photo viewer application would search the user scope at launch time for files tagged with "file_type:photo" or "file_extention:jpg". If the user adds a new photo, the cache coherence protocol ensures that the change is reflected in the photo viewer the next time it calls `list()`.

Applications can also subscribe to the device daemon for appropriate tags. The photo viewer could subscribe for tags like "file_type:photo" and "file_extention: jpg". When a new device with our native file system (e.g. a USB drive) is plugged in, the daemon scans its B+ tree and if it finds nodes for "file_type:photo" or "file_extention:jpg", it notifies the photo viewer. The photo viewer can then run a search on the USB drive with appropriate tags and add photos from the USB drive to its working set of files. In case the device plugged in has a foreign file system (e.g. a CD), the daemon first constructs a temporary in-memory B+ tree and then uses it to notify applications. The photo viewer application will be notified in this case because files are tagged with their extension (see Section 2.2.2) in the temporary B+ tree.

To copy files from a foreign file system to a native file system device, the user first uses the file layer to copy every file. This can be done by running a batch script which looks like:

```
copy_file()
{
    1.  Use create() to make a new file in the destination device.
    2.  read() the contents of the file from the source device.
    2.  write() the contents read in (2) to the file created in (1).
}
```

Figure 9: Pseudocode for a sample script to copy files.

The naming layer can then be used to generate tags for every file like the device daemon does (see Section 2.2.2). For copying files from a native file system, the procedure is similar, except that tags are simply copied from the source file system.

Executable binaries are tagged with "file_type:executable". Commands that can be run from the shell are tagged with "system:environment". When a user types "python" at the shell, the system runs a search for files with "file_type:executable", "file_name:python" and "system:environment" tags. The file in the resulting set is executed. If there are multiple files in the resulting set, users can tag one of them with "system:default" to set it to the default executable to run.

During installation, application runtime should be configured with the tag address space they use. For instance "Python 2.7" could choose to use "python_27: *attribute*" as its tag space. Later when we run the Python 2.7 interpreter, it already knows the tag space it needs to search in order to find its modules. To create pre-compiled versions of its modules, the Python runtime can create a new pre-compiled file (pyc file) on the same device and copy over all tags (except file extension) from the corresponding source code (py file) file. Since the tags of the source code file will already have the version embedded in them, there will be no conflict in pre-compiled files of different Python versions.

4 Analysis

The following table summarizes the metrics used for the performance analysis of our system:

Table 1: Performance Metrics Description.

Metric	**Value**	**Comments**
Tag Size	140 bytes	140 characters allowed
Inode Size	128 bytes	Size of inode structure
Directory Size	$4n$ bytes	n files in directory
Tag Density	20	Number of tags per file
Disk Latency	0.1 ms	Time to read/write a disk block
Disk Block	512 bytes	Size of a disk block

4.1 Single Tag Search Workload

B+ trees have a high branching factor, hence the tag leaf nodes will not be more than a few levels deep. Furthermore, we except most of the high levels of the B+ tree to be in cache most of the time. Therefore, we will assume finding the node for any tag takes a small constant number of disk reads.

A single tag search workload will be required by many application to locate their modules. Such a search requires the look up of a single node from the B+ tree and then reading the directory it points to, hence it should be sufficiently fast. We do not expect such scopes to be frequently modified, and therefore they are expected to be resident in the scope cache.

4.2 Multiple Tag Filter Workload

We will concentrate of a work load which requires filtering by two tags only, because the result of its analysis can be extrapolated to an arbitrary number of tags.

Our scenario will be a user wanting to search for files tagged with "location: Colorado" and "year:2007". Let X be the directory file for the "location: Colorado" tag, and Y for the "year:2007" tag.

4.2.1 Cardinality of X and Y is close

In this case we linearly scan both directories to generate their intersection, so time taken to evaluation the intersection is $O(|X| + |Y|)$.

In all the system has to read $4X + 4Y$ bytes to generate the intersection. The amount of time this will take is given by equation 1.

$$\frac{4(|X| + |Y|)}{512} \times 0.1 \quad ms \tag{1}$$

4.2.2 Cardinality of X is much larger than cardinality of Y

As described in Section 2.3.3, in this case we manually check the tags of files in the smaller directory. Back-of-the-envelope calculation done below shows that if X is approximately 556 (or more) times larger than Y, this approach will be faster.

$$\begin{aligned} 4X + 4Y &> 128Y + (15 \times 140)Y \\ 4X &> 2224Y \\ X &> 556Y \end{aligned}$$

The greater the difference in the size of X and Y, the faster our optimized approach will be. Its impact should be significant when one of the tags in the search query is very popular. Figure 10 compares the difference in performance of the linear scan approach with this optimized approach.

4.3 Join Workload

When joining results of filter operations on multiple scopes, we don't need to create a new list of `(deviceId, fileId)`. Instead, the results can be pipelined to the requesting process; no additional disk I/O is required.

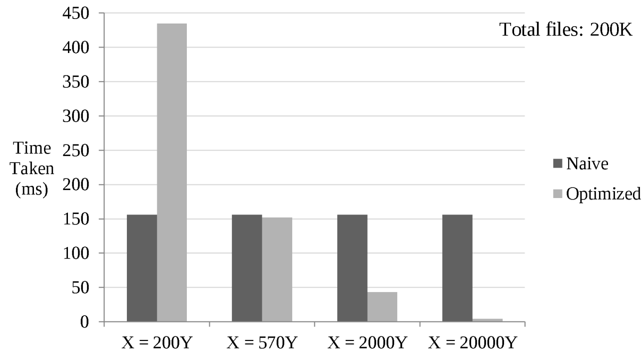


Figure 10: Comparative performance of the optimized and naive filter operations.

4.4 Read/Write Workload

Since we use the same file layer as the standard UNIX file system, we expect read and write operations on files to perform at the same level.

4.5 Impact of Scope Cache

The system will define multiple scopes to speed up common user operations. For example, the system can define a scope with tags "system:environment" and "file_type: executable". This scope represents the set of binary executables which can be run from the shell. When a user types a command, the shell can just search this scope rather than creating a new scope from three different tags as described in Section 3. The scope described earlier is likely to be resident in the scope cache most of the time because it will be frequently used and rarely invalidated. Therefore, we expect such common operations to run effeciently by making effective use of the scope cache.

5 Conclusion

The described system effectively uses a B+ tree to organize files using tags. It requires some tagging conventions to simplify implementation and encourage consistency amongst applications. A number of optimizations have been described to enhance average and worst case performance. These include: caching scopes to avoid the overhead of re-evaluating scopes on every call to `list()`, bucketing tags to contain the size of on-disk B+ trees, storing files in sorted order in directories to speed up evaluating intersections, and optimizing the filter operation in cases when one of the tags is very popular.

2592 words (excluding pseudocode, internal diagram text and section titles). Sorry!