# FindFS - Adding Tag-Based Views to a Hierarchical Filesystem

by

Jason Chou

B.Sc., The University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2015

# Abstract

Organising files by grouping them using tags is an alternative approach to filesystem design that has benefits over the traditional hierarchical model. However, the majority of filesystems in use remain hierarchical. In this paper we describe FindFS, an extension middleware which provides dynamic, tag-based views of an existing hierarchical filesystem. FindFS adopts the functionality of the `find` utility and adds support for extended attribute queries, which can be used for tagging files and filtering the filesystem. Search results are represented as directories containing symbolic links which are kept up-to-date in response to filesystem operations, allowing them to persist as views that are accessible from existing unmodified applications. Control of the system is accomplished via filesystem operations, enhancing its ease of integration and portability. We have developed a prototype of this system, and characterised the performance overhead that it adds to file operations.

# Preface

The contents of this thesis were submitted, as a candidate paper of the same name, to the *Middleware 2015* conference. The primary author was Jason Chou, with supervisor Mike Feeley as a secondary author.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgements

Many thanks to supervisor Mike Feeley for his suggestions that resulted in the initial idea, and providing assistance throughout the progress of this project until its completion.

# Chapter 1

# Introduction

Hierarchical filesystems have become well-established in mass storage systems since they provide an easily understood and intuitive model of organising data in the form of a tree structure. Because of this, the convention of pathnames, directories, and filenames is nearly universal across most filesystems, and it has resulted in the existence of a large amount of software which makes use of such names in identifying the files they work with. However, the hierarchical model of organisation has its limitations, one of which is the fact that it is not always possible to find a unique or convenient single hierarchy in which files are easily categorised.

Semantic filesystems provide an alternative design. First described in [1], this concept involves the use of various types of metadata to produce a set of attributes in key-value format, which are assigned to the files in the system. By constructing queries on these attributes, a subset of the files can be identified and grouped together in a non-hierarchical manner, serving to provide a "flattened" namespace view.

A common way to configure such a system is to "label" files with their metadata-derived attributes; the term "tag-based" has become commonplace in the description of these systems. These tags allow any application, and by extension the user, to manipulate files through any of the groupings to which they belong, in a more

flexible way than any fixed hierarchy.[2][3]

Although they may still contain elements of hierarchical access, tag-based filesystems are distinguished by their ability to allow for files to be selected in a non-hierarchical manner. Because the files stored in modern systems can have various types of content-derived metadata attributes associated with them, it thus seems natural that a file should not be contained in any single group, but instead be able to be viewed as part of a variety of groups, based on the values of attributes associated with it. Furthermore, it should be possible to take advantage of these attributes when searching. For example, a photo application will place image files in directories, perhaps based on when they are imported into the application; but these files might be naturally grouped by such attributes as when or where they were taken, or rankings or tags placed on them by users.

There are many more dimensions of organisation one can use for photos (orientation, camera model, resolution, filesize); and the same is true for the other types of files one may wish to store, such as music (date, album, artist, genre, length), making the task of finding a good hierarchical organisation more difficult. Furthermore, the huge growth in production of data by users puts pressure on means to organise this data with more effective techniques. It becomes clear that, while a hierarchical configuration of directories is one useful way to organise files, confining a file to a single taxonomy is too limiting for modern systems. Filesystems now store vastly more files and a much wider array of file types than they did when the hierarchical namespace was born.

In contrast, a hierarchical filesystem rigidly confines files to a single directory. Other dimensions of grouping are left to the application and as such are not available

through the filesystem's API. This limitation is unfortunate because it is simple and universal. Every application that accesses persistent storage typically uses this API.

It has been many years now since the hierarchical filesystem was declared dead[2], and yet it lives. The argument for replacing the file hierarchy and switching to one of these new designs is powerful; why then is it that every mainstream commercial filesystem stores files hierarchically? The filesystem plays a particularly important role for applications. It sits between every application and persistent storage. It must be robust, performant, and scalable. Persistent data must survive a variety of software failures — and some hardware failures. The filesystem implementation must be as free of bugs as possible; and so, filesystems are carefully engineered and system vendors are extraordinarily resistant to making wholesale changes to their implementation.

However, it is not necessarily true that the hierarchical filesystem must be replaced in order to build a system which can provide a more flexible form of file organisation and access. Users have learned to organise files effectively and develop ad-hoc methods of accessing them non-hierarchically, given the constraints of hierarchical filesystems. We can build upon those organisational techniques to enhance their functionality and effectiveness, resulting in a system which combines all the advantages of traditional hierarchical filesystems with the flexibility of tag-based ones.

The exploitation of this idea has resulted in FindFS, a system which allows non-hierarchical, tag-based access to an existing hierarchical filesystem by using the features that they provide. The key feature of this approach is that it is layered entirely on top of a traditional filesystem, and thus inherits the large engineering

efforts that made them robust, nearly bug-free, and scalable. As described herein, exposing the hybridisation of performant, ubiquitous hierarchical filesystems and emerging flexible tag-based file selection strategies through familiar interfaces creates a powerful, practical, and low-overhead solution for the efficient management of the large, growing semantically-rich file collections common in systems today.

# Chapter 2

# Related Work

Previous work in attempts to move away from the pure hierarchical filesystem and offer tag-based access have ranged from reimplementing most of the system starting at the layer of block storage [2], augmenting a filesystem with a tag metadata database [1][3], or as an application layer shell feature [4] or library [7]. All of these approaches have certain disadvantages. A filesystem reimplemented with non-hierarchical access as its primary goal may perform poorly with the large base of existing applications that were written to take advantage of unique characteristics of hierarchical filesystems, such as access locality within directories. Systems that use databases incur the overhead of the database system, one whose full capabilities may rarely be needed. Finally, an application-level feature is exclusive to a given application or library, and is unavailable to use for other existing applications.

The original semantic filesystem [1] used pathnames as queries into a database to present virtual, created-on-demand directories containing the desired files that match the query. This method is suited to ad-hoc exploration, but for often-used queries or those with more complex expressions, it is advantageous to persist the expression as well as its results in a more convenient form, such as in an actual filesystem structure, so that it can be easily referenced in the future and avoid recomputation. TagTree[5], TagFS[6], HAC[7], Tags for Plan 9[8], and the Logic

File System[9] are also based upon a similar strategy, only differing in how they represent attributes and their mapping to pathnames.

These aforementioned systems use an underlying hierarchical filesystem to store the data of the files, augmenting it with a database and indexing; in contrast, hFAD[2] eliminates the hierarchical filesystem completely, replacing it with a database that directly utilises a block storage allocator and using that for both metadata and file storage. It is able to emulate a hierarchical filesystem by storing paths in the database, and while there are no benchmarks presented, there are other examples of using a database as a filesystem, such as Inversion[10], which show their higher overhead. In these systems, the need for a database, and the specific APIs and applications required to manipulate one, restrict their generality compared to a fully-filesystem-based approach.

Approaches in providing non-hierarchical access at the application level have also been studied, such as by extending the `chdir` command of the shell to perform a search of the hierarchy[4]. This is a relatively simple and non-disruptive change to the system, but also the least general, since such a feature becomes exclusive to an application and would have to be replicated or explicitly linked as a library for it to be usable by other applications. HAC[7] is an example of a more fully-featured application-level approach, providing a virtual, tag-based hierarchical namespace filesystem within a library. In consideration of the fact that file access is a very general capability, and thus one that should be consistent across applications, this limitation can be extremely disadvantageous.

Other examples of application-level approaches include OS X's Spotlight[11] and Windows Desktop Search[12], which utilise an indexer to collect metadata about the

files in the system and store it in a database. This allows them to persist a search query[13] and present to the user a "virtual folder" which is dynamically updated with matching files[14], but since they are not accessible via the filesystem API, other applications cannot access their contents in the same way they do with files in a regular filesystem. In practice, since the application in question is a GUI and the primary means for the user to access the filesystem, this is not a strong disadvantage — the "real" path is passed as a parameter when another application is executed on a selected file — but the ability to navigate and inspect search results in the same way as other structures in the filesystem is a useful feature. The fact that these virtual folders are specific to the shell application is noticeable when accessing files from within another application, since they do not have a pathname and are not visible in the filesystem.

# Chapter 3

# FindFS Overview

The general approach we take in providing an alternative non-hierarchical view is to leverage the capabilities of an existing hierarchical filesystem as much as possible. This obviates any need to reimplement much of the underlying functionality with regards to file storage. It allows reuse of the standard filesystem API, meaning that existing applications do not need to be specifically modified in order to take advantage of the new features and can continue to access files as they did before.

From the user perspective, the filesystem remains compatible with both existing applications and the well-ingrained idea of hierarchical storage; there is no need to relearn or be forced into using a new paradigm of file organisation. However, when the need arises to use a tag-based view of the filesystem, these capabilities become available. This approach is motivated by the observation that with only the existing features available in a standard Unix system, users are already able to identify and manipulate files in ways that are not strictly hierarchical. However, doing this requires extra effort and has its limitations; FindFS builds on the principles behind these ad-hoc manipulations and extends them into the form of a filesystem, making search results which persist and are updated according to filesystem changes.

## 3.1 Enabling Multiple Hierarchies

Consider server access logs for a series of $N$ websites; they may be organised in a hierarchy of the form `logs/siteN/YYYY/MM/DD`. A consequence of this choice is that it is easy to obtain all the logs of one site for all dates, a particular date, or set of dates; they can all be found under e.g. the `logs/site3` directory. On the other hand, to obtain the logs of all sites on a particular date requires "cutting across" this hierarchy, since the path component that varies in this case is not the last one. Alternatively we could choose to use the hierarchy `logs/YYYY/MM/DD/siteN`, which makes the latter use case easier but the former harder.

Links — symbolic[15] or otherwise[16] — provide a standard solution to this problem in the context of a hierarchical filesystem; they allow a file to be accessed via multiple pathnames, meaning that it can appear to be in different hierarchies. We choose to exclusively use symbolic links in FindFS, due to their added flexibility in allowing links to directories and across different filesystems.

With this flexibility, the user can create hierarchies as needed, and add links to the appropriate files in it to enable different views of the filesystem; to use the log example, both the `logs/siteN/YYYY/MM/DD` and `logs/YYYY/MM/DD/siteN` hierarchies may coexist, with one of them containing links to files stored in the other, or both of them links to same set of files stored in an entirely different hierarchy (e.g. a "flatter" structure, with the names containing this information: `logs/N-YYYY-MM-DD`.) The original semantic filesystem [1] embodies this idea, creating symbolic links from a "virtual", on-demand tag-based hierarchy to files stored elsewhere. In FindFS, search results are also directories of symbolic links to matching files, stored in the

underlying filesystem, but we depart from the traditional semantic filesystem in the handling of tags and file selection.

## 3.2 Tag-Based Filtering and Searching

On standard Unix systems, the `find` utility standardised by POSIX [17] is the usual means by which users can search the filesystem; in its most basic usage, it takes as parameters a series of paths and an expression, and recursively traverses these paths while evaluating the expression for each file/directory encountered. The expression allows matching on various metadata such as names, sizes, dates, and ownership, which combined with boolean operators allows for complex queries to be easily constructed. The effect of this filtering process is not unlike that of tag-based filesystems, where files are selected based on the metadata of their tags.

However, the standard `find` queries only allow selection based on the typical metadata found in a traditional filesystem. Index-based search systems have evolved similar tools such as `mdfind`[18], which query the metadata database, but its expression syntax[20] is not compatible with `find` and they are specific to one index-based system. A more general solution is available in the filesystem, in the form of extended attributes. Although the standard POSIX filesystem API has no support for them, many filesystems such as ext*, XFS, etc. do provide this feature[19]. This allows a set of key-value pairs to be associated with each file, and thus be used to store the information on the tag namespace. An application like `find` could then be used to query these attributes, providing tag-based filtering.

## 3.3 Persisting and Updating Results

The output of `find` is not a continuous view of the files that it selects, but an approximately instantaneous list created at the time it is run; in other words, the results are not inherently persistent. Using the flexibility of I/O redirection, one could save these results to a file, or use them to create an alternate hierarchy of links. For example, with a suitably extended `find`, the command

```
$ find logs -eav user.date_year 2015 -a -eav
 user.date_month 05
 | xargs -I linkdest ln -s linkdest
```

could create a set of links (a *resultset*) in the current directory to all the logs tagged with a date of May 2015. This solves the problem of persistence, but if the files in this resultset are modified such that they no longer exist or satisfy the query, or additional files are added that do, the resultset will become inconsistent with the filesystem. Thus, a method of ensuring that these links are up-to-date is required.

A set of these commands could be put in a script which refreshes the resultsets when periodically executed, but such a "polling" method is an inefficient use of resources. Increasing the refresh frequency allows the results to be kept in closer correspondence to filesystem changes but consumes more processing time, while reducing it leaves more resources for other processes but decreases the consistency of results. However, no periodic update frequency guarantees that the results are consistent whenever they are inspected; the solution is to not poll but to use event-based notifications of filesystem activity, causing the results to be updated synchronously

with their associated filesystem operation. In this way, no inconsistency can occur, and no unnecessary work is performed when there are no associated filesystem operations. FindFS adopts such a solution, monitoring filesystem operations and updating resultsets accordingly.

## 3.4 Control Interface

In addition to monitoring filesystem operations for the purposes of modifying resultsets, FindFS also monitors operations performed on special files and directories which allow control over query creation, modification, execution, and deletion; queries are themselves also special directories. As with resultset contents, these special files are persisted by the underlying filesystem like any other file, relieving FindFS of the need to explicitly manage their storage. For example, the selected paths and query expression are written to a `query` file, while the `control` file can be read to determine the state of a query as well as written to command it to a different state, e.g. to initiate a `find` operation. The `output` file provides textual informational, warning, and error messages, while the `results` directory contains the query's resultset.

Presenting a control interface via the filesystem API is a common practice in existing Unix and Unix-like operating systems[21][22], and enables applications to build upon it to provide more advanced functionality. It also allows for an easy extension to distributed systems[23], since the filesystem API is uniform and a well-understood paradigm for interaction.

# Chapter 4

# FindFS Design

FindFS is designed to present itself to the OS as a filesystem, while also requiring an underlying filesystem on which it can search and also store the persistent data associated with its search queries. This means that FindFS can be mounted anywhere in the OS's virtual filesystem tree, allowing applications to issue requests to it; once mounted, it acts as a thin layer between applications and an underlying filesystem, transparently passing through requests from applications and acting on those which have a special meaning to it such as query control operations.

Because the underlying filesystem is specified as a path relative to the VFS root, FindFS' filtering capabilities can be selectively applied to subtrees of the VFS. This flexibility is advantageous since access via FindFS adds overhead (examined in more detail in Chapter 6), and it may not be beneficial to apply it to certain directories in the same way that indexing systems may not attempt to index all files. For example, temporary and system directories could be excluded, while those containing metadata-rich multimedia are included.

## 4.1 Queries

FindFS' model of operation is based on the concept of the query, through which filtered views of the filesystem are observed. Once mounted onto an underlying filesystem, a `queries` directory is visible in the root of the mount, and operations on the files and directories within this subtree are interpreted as query control operations. A query is an object represented by a directory in the `queries` directory, and itself contains files and directories which reflect its state. The manipulation of queries is accomplished entirely via filesystem operations; for example, the creation of a subdirectory of the `queries` directory creates a query of the same name. Whenever a query is created, FindFS creates a few special files inside the query's directory and monitors the activity on them, allowing the user to control the query. Similarly, when deletion is performed on a query's directory, FindFS also frees the other resources associated with a query.

Queries are always in one of four states: `INIT`, `FIND`, `WAIT`, and `STOP`. Changes in the state of a query can be effected either by the system itself or by the commands issued to it via the `control` file. Figure 4.1 shows the states and transitions which are commanded or system-initiated.

### 4.1.1 Query State `INIT`

The `INIT` state is the state in which a query is initially in after it has been created (by the use of `mkdir()` in the `queries` directory). In this state, its `query` file can be freely edited to construct the desired query. The results directory is present, but empty. The query can be removed with a `rmdir()` call on the query's directory,

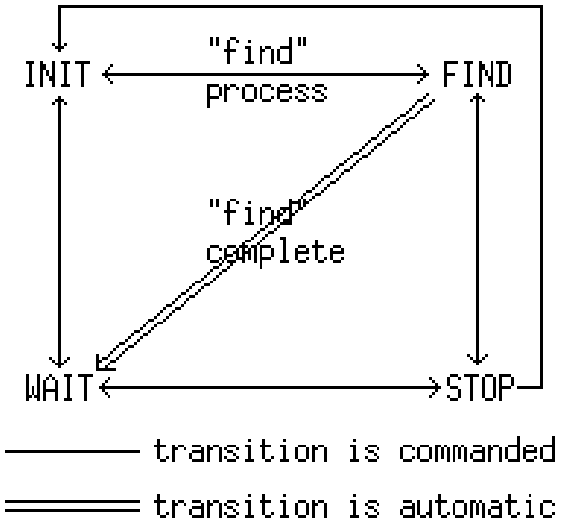


Figure 4.1: Query States and Transitions

which automatically removes all the files as well as directories that were created during the creation of the query. Writing `find` or `wait` to the controlfile causes an attempt to transition to the indicated state. When this occurs, the system parses the contents of the `query` file, and if the query is syntactically valid, the transition is allowed. From `INIT` to `WAIT`, the query becomes *active* and its results directory will be updated with the files and/or subdirectories which match the query as a result of the filesystem operations executed by applications. From `INIT` to `FIND`, in addition to new operations, a process similar to the `find` utility is used to collect existing files that match the query. Otherwise, error messages are written to the `output` file and the transition does not occur, allowing the user to edit and correct the query.

### 4.1.2 Query State `FIND`

The `FIND` state is anticipated to be the usual state a query will transition to from `INIT`. In this state, the system searches the filesystem for files which match the query, a process similar to the `find` utility — hence the name of this state — and adds them to the set of results. In addition, files that are accessed, modified, and created during this state will also be subjected to an evaluation by the query's expression. When the `find` process is complete, the query automatically transitions to either the `WAIT` or `STOP` state, depending on the options used for the query. While a query is in this state, it is possible to command it to `INIT`, `WAIT`, or `STOP`, interrupting the process.

### 4.1.3 Query State `WAIT`

A query in the `WAIT` state has finished its process of traversing the existing filesystem and gathering results, and is waiting for filesystem operations in order to update the resultset. (Alternatively, if it was commanded into this state without entering `FIND`, the resultset will be initially empty but updated as new filesystem operations occur.) In the terminology of the system, queries in a `FIND` or `INIT` state are known as an *active* query. In this state, a query may be commanded to the `FIND`, `INIT`, or `STOP` states to respectively reinitiate a `find` process, be reinitialised after altering it, or made inactive.

### 4.1.4 Query State `STOP`

`STOP` is the other state, besides `INIT`, in which a query is inactive; in this state its results remain accessible but, because its processing in response to filesystem operations is disabled, do not change. It is a state that is useful when managing many queries; for example, a `WAIT`ing query whose results are not anticipated to change during an upcoming period of intense filesystem activity can be commanded to `STOP` for that period, avoiding the overhead that it adds, and be "resumed" to `WAIT` afterwards.

This state can also be used to preserve certain historical information about the filesystem, since a query which goes through the `INIT`→`FIND`→`WAIT` sequence contains in its resultset all the files existing which match it, and those that currently do — until made inactive. Thereafter, files that now match and files that no longer match have no effect on it, so it has effectively captured a snapshot of matching files; deletions and renames notwithstanding, since results are symbolic links. Combined with the `INIT`→`WAIT` sequence, this allows persisting information on files that were accessed and matched a query in a certain timeframe, a task not possible with the `find` utility.

## 4.2 Expressions

The query expression language of FindFS is an extension of that supported by the `find` utility. It includes all of the primary-expressions (predicates) defined by the POSIX[17] standard, and adds those shown in Table 4.1 to allow matching on extended attributes.

| Predicate | Description |
|---|---|
| `ea` *name* | EA *name* exists |
| `eav` *name value* | EA exists with *value* |
| `eavl` *name value* | EA exists with a value $<$ *value* |
| `eavle` *name value* | EA exists with a value $\leq$ *value* |
| `eavg` *name value* | EA exists with a value $>$ *value* |
| `eavge` *name value* | EA exists with a value $\geq$ *value* |
| `eavne` *name value* | EA exists with a value $\neq$ *value* |

Table 4.1: Predicates for Extended Attributes (EAs)

The predicates that require a value to compare against have two forms; the one shown in the table, intended for attribute values which are relatively short printable strings, and an alternate version intended for either longer attribute values or those containing nonprintable characters. The latter form takes a filename for the second argument, and uses the contents of that file as the value; they are named with an *f* replacing the *v*: `eaf`, `eafge`, etc. Because attribute values are stored in the filesystem as a sequence of bytes, not unlike file contents, they are not necessarily required to be strings, and as such, no C-style null-termination is applied to the specified values.

Since the standard predicates and operators are also available, they can be used together with the extended attribute predicates, further increasing the flexibility of query expressions; unlike the previous efforts discussed above, whose query expressiveness is limited to matches on the tag namespace, FindFS allows filtering by standard metadata, and is usable even without any extended attributes. For example, a FindFS query could be created to contain all files that were modified in the last hour and owned by a particular user.

It also does not impose any structure on the tag namespace, since the match-

ing predicates it supports are quite general; by using only the name of extended attributes (the `ea` predicate) in selection, tags can be simple labels. Using both name and value, and comparing values with the equality or inequality predicates combined with the standard conjunction and disjunction operators (`-o` and `-a`), an arbitrarily complex tagging system can be utilised. Furthermore, there is no restriction on the values, allowing binary and text formats to be accommodated. The particular design of tagging systems is beyond the scope of FindFS — it supports any semantic model that can be expressed as key-value pairs or standard filesystem metadata, such as those discussed in [24] and [25]. Likewise, this metadata could be derived from file contents and updated when they change, or produced in some other manner; as long as metadata changes are performed through the existing interface that FindFS monitors, it will interoperate.

For example, considering the use-case of website access logs presented in Chapter 3, FindFS may be employed in various ways to achieve the filtering desired. With filenames that contain the desired information, e.g. using the `N-YYYY-MM-DD` format, a query of the form `logs -name 5-*` will match all the logs for site 5, `logs -name *-2015-01-*` all the logs for January 2015, etc., presenting them in the query results as a "flattened" view in a directory of links. Observe that, even without any use of extended attributes, there is already substantial filtering flexibility available. However, its use with files that have extended attributes brings more flexibility; suppose that the process which generates the logs also adds an extended attribute with the name `user.high_rate` to the log files containing an unusually large number of recorded accesses. Then the query `logs -ea user.high_rate` can be used to provide a continuously updated view of the sites receiving the most traffic, and

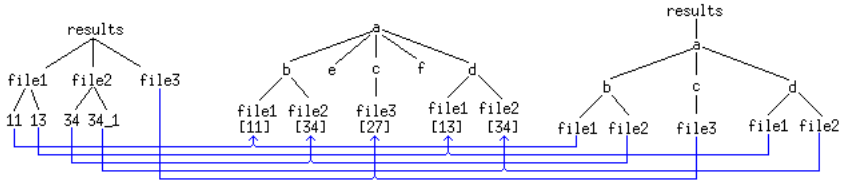


Figure 4.2: Flat (left) and Hierarchical (right) Resultset Structure

when, for further analysis or even additional filtering via another FindFS query that operates on this query's results.

## 4.3 Resultsets and Dynamic Update

In the `results` subdirectory of the query, references to matching files and directories are present in the form of symbolic links, in a similar manner to the original semantic file system. A query can be configured to use one of two different formats for the results: a flattened list, or a hierarchical view. These two formats are provided to allow flexibility in choosing the one which is most suitable to the specific use-case, since they both have advantages and disadvantages. Figure 4.2 graphically illustrates the difference between these formats.

The design of the resultsets follows from the requirement that operations on them, particularly incremental ones, should be both efficient and make use of underlying filesystem features as much as possible. Furthermore, by monitoring filesystem activity, FindFS continuously keeps the contents of this directory updated to reflect the latest modifications to the filesystem. Thus, by navigating inside this subtree, one can obtain a dynamic realtime non-hierarchical selection of files from

the original filesystem.

### 4.3.1 Flat Resultset

A flat resultset, whenever possible, represents its contents using a single level of hierarchy — a symbolic link in the `results` directory of the query, named after and targeting the matching file; this is not possible in all cases since using the name of the file in the results is most sensible to the user, and traditional hierarchical filesystems do not impose any requirement that filenames be globally unique. To handle instances of duplicate names, FindFS adds a second level of hierarchy, using a subdirectory named after the file and putting the links in it. These links are named after the inode numbers of the files they reference; a decision whose benefit over a naming that appears to be simpler at first glance, like sequentially increasing numbering, is that it avoids counting and the issues that arise when files are added and removed from the resultset.

A second level of duplication, which can occur with files that have multiple hard links of the same name, is handled by appending an "`-`" to the inode number and a sequentially increasing, first-fit identifier; here, we have resorted to counting since this case should be rare in practice — the majority of regular files in a filesystem usually have a hard link count of 1.

For example, a resultset containing the following paths with the corresponding inode numbers

```
/a/b/file1  [11]
/a/b/file2  [34]
/a/c/file3  [27]
/a/d/file1  [13]
/a/d/file2  [34]
```

would be represented by a results directory of the following structure:

```
results/file1/11 -> /a/b/file1
results/file1/13 -> /a/d/file1
results/file2/34 -> /a/b/file2
results/file2/34_1 -> /a/d/file2
results/file3 -> /a/c/file3
```

### 4.3.2 Hierarchical Resultset

On the other hand, the hierarchical resultset presents a "virtual hierarchy" tree rooted in the result directory of the query, which positions the symbolic link for a file in the resultset nested in a set of directories with the same name as found in its path in the underlying filesystem, i.e. relative to the root of FindFS' mount point. Using the same example as above, the resultset structure becomes:

```
results/a/b/file1 -> /a/b/file1
results/a/b/file2 -> /a/b/file2
results/a/c/file3 -> /a/c/file3
results/a/d/file1 -> /a/d/file1
results/a/d/file2 -> /a/d/file2
```

With this format, duplicate names do not pose any problem, meaning that the resultset modification algorithm can be more efficient for some operations. The trade-off is that queries which match files in deeply-nested directories tend to create a tree structure with low fan-out, i.e. they are sparse and mainly consist of directories whose only content is a single subdirectory, and this frustrates access to the files at the ends of these long branches. One capability which this format lacks compared to the flat mode is for resultset entries to be directories; since it requires the directories to store links, it is impossible to have a link to a directory as a result and also as a directory to store results matched in its subtree. With the flat format, those directories which match are accommodated with the same mechanism for regular files: by creating links to them in the result directory, or in a directory one level deeper (if there is a duplicate name).

### 4.3.3 Resultset Format Comparison

The two resultset formats also differ in performance characteristics, which make them suited to different use-cases. A flat resultset, although slower in response to some filesystem operations, is ideal for queries which do not have many results or duplicate names expected, and for which the user desires to see all the files in one directory without needing to recurse into a series of subdirectories. It represents a mostly non-hierarchical view that emphasises the selection process of the query, which is similar to those of the other tag-based filesystems. The hierarchical resultset presents a view not unlike a traditional a hierarchical filesystem, but one that is filtered to include only the files (and their containing directories) which match.

### 4.3.4 Resultset Updating

The resultsets of active queries are kept updated by monitoring operations on the filesystem, and adding or removing files from them according to the result of evaluating the query's matching expression on the changed file(s). Because the values of expressions depend only on filesystem metadata, they may change, and the results be updated, only when such metadata changes. Thus, only those operations on a file which can affect its presence in a resultset need to be monitored by FindFS. This includes creation, deletion, renaming, and modifying of extended attributes, but not reading or writing; although the latter two operations may update access times, this is handled upon closure of the file.

The semantics of expression evaluation differ slightly from that of the `find` utility, to better reflect how FindFS operates; this is because the utility does not actually perform a filtering operation — instead, its operation can be considered as one that executes an expression on each file in a directory tree, and the usual output it produces is only a side-effect of this recursive filesystem walk. In fact, the `print` primary predicate is used to produce this output; given the possibility of multiple occurrences of `print` in the expression, or none at all, the standard[17] specifies that in absence of a `print`, `exec`, or `ok` predicate, the actual expression which is evaluated is conjuncted with `print` (i.e. *expr* `-a -print`). This results in the usual output if no `print` is explicitly specified. In FindFS, the value of the expression is used to determine resultset membership and `print` evaluates to true with no side-effects, which achieves good compatibility with existing `find` expressions.

## 4.4 Example Interaction

Since FindFS is entirely controlled by filesystem operations, its interface is available to all applications; this makes viable the possibility of developing specialised applications to assist users in creating and managing queries, but also allows its operation via existing command-line utilities, as shown in the following interactions. Assuming the current directory is the root directory of the FindFS mountpoint, we start by creating a query named `q1`:

```
$ mkdir queries/q1
```

The query now exists but cannot be executed (nor is it active), since it is in the `INIT` state and contains no matching expression. The following command may be used to give it one:

```
$ echo '/ -name file1.txt -o -eav user.attr1 value1
 ' > queries/q1/query
```

This matches files with a name of `file1.txt`, or those with an extended attribute named `user.attr1` and value `value1`. The syntax used here is equivalent to the parameters to the `find` utility; options are specified first if present, followed by one or more paths (relative to the mountpoint, where `/` refers to the mountpoint directory), and finally the query expression. Strings may be quoted in the usual shell manner via single and double quotes and escaped with the backslash, to facilitate use with directories containing spaces or other special characters; neither command nor variable substitution are supported, with those characters being parsed as regular

ones. If the need arises, they can be performed using the shell in the customary manner, and the results, correctly quoted, written to the query file.

In the `INIT` state, the query file may be rewritten multiple times, as writing to the file does not trigger any FindFS operations; this allows composition of arbitrarily complex queries with saves of partial edits, and applications such as text editors to manipulate the query file.

The state of this query is examinable by reading the control file:

```
$ cat queries/q1/control
init
```

However, a write to the control file, as effected by e.g. this command,

```
$ echo 'find' > queries/q1/control
```

will command FindFS to request a state change of the query object as described above. This may also be performed via another application, but care must be taken when doing so to ensure that it writes only the command and nothing else to this special file, and that reading will obtain the current status instead of the command last issued. After a query has been put into the `FIND` state, its progress can be monitored using the control file:

```
$ cat queries/q1/control
find
$ cat queries/q1/control
find
$ cat queries/q1/control
wait
```

The results of a query are also always available in the results directory, in the format described above, and while the query is active, will be updated in response to changes in the filesystem:

```
$ ls queries/q1/results
file1.txt
file2.txt
file3.txt
$
```

Finally, to remove a query, which automatically stops it and frees associated resources (including its resultset and control files), it suffices to remove its directory:

```
$ rm -rf queries/q1
```

# Chapter 5

# Implementation

We have implemented a prototype of FindFS as a FUSE[26] filesystem application, in C, instead of a kernel module. This allows use of the facilities available to user-mode applications such as the POSIX API and C standard library, which eases implementation due to much of the functionality being inherited from its origins in the `find` utility. The development of FindFS began with creating a version of `find` with extended attribute support; a significant portion of which was then reused, for parsing and evaluating the query expressions. The implementation totals less than 2500 lines of code, approximately half of it for `find`-like expression parsing and evaluation and the other half for code specific to FindFS: query management, resultset modification, and filesystem request handling. Figure 5.1 shows how filesystem operations from applications are passed through to the FindFS application via FUSE, while FindFS itself performs operations on the underlying filesystem.

Although the use of FUSE allows for easier implementation and portability, it increases overhead and reduces performance; all filesystem operations in the FindFS mount subtree need to pass through the FUSE layer, including those that cannot cause resultset changes (e.g. reading and writing), and thus those that FindFS would not need to inspect. This could be solved by modifying the FUSE kernel module to perform transparent passthrough operations itself — based on what op-

erations the user-mode application desires to handle — or by having FindFS itself be a complete kernel-mode filesystem. The former retains most of the portability but also the extra kernel-user transitions and a relatively restrictive API (in terms of filesystem manipulation — see Section 5.2), while the latter, although requiring modifications for different kernel APIs, allows for more efficient filesystem manipulations and eliminates kernel-user transitions for each filesystem operation on the FindFS subtree.

## 5.1 Implementation Goals

In keeping with the theme of using the filesystem whenever possible, there is very little data the system explicitly manages in memory; only the identifying information, state, and parsed expression for each query are kept in memory by FindFS. In particular, this means the resultset is managed by the filesystem, and operations on it are handled by the filesystem driver. This is beneficial since it simplifies

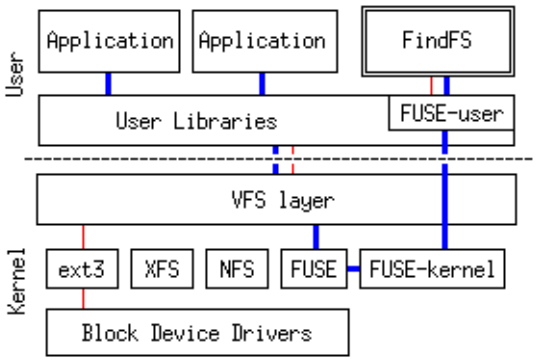


Figure 5.1: FindFS Component Interaction (thick lines show application↔FindFS requests, thin lines show FindFS↔filesystem requests)

FindFS and allows the number of results to not be limited by available memory, and also automatically makes the results persistent. One possible disadvantage is that accessing the filesystem is slower than a local in-memory data structure, but it was decided that the increased complexity and memory requirements (being at least proportional to the size of the resultset) of doing so, along with the goal of persisting the results, meant that access to the filesystem would be unavoidable; additionally, there is caching in the OS's filesystem and block device drivers, reducing the performance impact.

One additional effect of this design choice is that if FindFS is mounted onto the same location as its underlying filesystem root, even when it is not mounted, the contents of queries and their result directories remain on the filesystem and thus can be accessed by applications; the only difference is that there is no dynamic update, so they may not correspond to current filesystem conditions.

Furthermore, since FindFS queries are intended to be long-lived and filter the filesystem while the user performs other work, their presence should not have a large impact on the performance of the system. This agrees with the design principle of not using more memory than necessary in an attempt to improve its performance, since that will be detrimental to other applications the user may have more importance on; FindFS should be unobtrusive, and without any queries active, should not consume any additional resources.

This is unlike indexing-based systems that need to perform regular index updates regardless of whether the information in the indices will eventually be used, at the risk of wasting that effort and causing unnecessary filesystem activity which can impact the performance of other applications on the system. Our implementation

of FindFS occupies less than 26MB of memory initially, and each additional query (of a typical complexity, i.e. 2-3 terms) adds less than 1KB. As there is no persistent index, the only filesystem storage needed is proportional to the number of queries and the number of results they contain. Since our implementation does not attempt to merge duplicate parts of query expressions or resultsets, both space and time are directly proportional.

## 5.2 Resultset Processing

Using the standard POSIX filesystem interface imposes certain restrictions on the way in which resultset modifications can be performed; for example, the only way to determine the number of entries in a directory is to exhaustively read them — which is not a constant-time operation. This is due to early Unix filesystems' storage of directories as a series of variable-length entries, without any count of the number of entries. The inode number of a file, which is easily obtained via `stat()`, cannot be used to refer to it for any other operations — the standard API accepts only textual paths and file descriptors. Textual paths are problematic in that their manipulation requires operations on null-terminated strings, which are neither constant-time nor space.

There is no way to find the path(s) associated with an inode number except via exhaustive search of the filesystem. Determining if a filename exists in a directory can vary from nearly constant-time (hashed lookups) to proportional to the number of entries in it, depending on the directory format and implementation of the underlying filesystem. However, even with a linear search, directory lookups

are usually not a problem except for the very largest of directories since the entries are relatively small, relative to the speed at which the storage system can retrieve them. Caching further improves on this, making it likely that that the contents of frequently accessed directories are entirely in memory.

### 5.2.1 Path Existence

All resultset modifications require determining if a path exists in one, which may be followed by addition or removal (or neither), depending on the result of evaluating the query expression. Since these tests of existence need to be performed whenever an operation on the filesystem may cause its membership in a query's resultset to change, they must be efficiently implemented. Using the flat format, a path not in the resultset requires one directory lookup on the name; otherwise, a `stat()` operation follows, and then either a single `readlink()` in the case that there is one existing result, or opening the directory and reading all the entries and performing `readlink()` on ones with a matching inode number, if there are multiple. With the hierarchical resultset format, a single directory lookup of the path relative to the result directory suffices; the presence of a link signifies membership.

### 5.2.2 Path Addition

If path existence equals the value of evaluating the query's expression with it, nothing more needs to be done; otherwise the given path must be added or removed from the resultset. For a flat resultset, adding the first result of a given filename is a single `symlink()`; the second requires `unlink()`, `mkdir()`, and two `symlink()`s;

and the third and subsequent results, depending on presence of hardlink collisions, one or more `symlink()` calls. With a hierarchical resultset, adding a path results in a number of `mkdir()`s proportional to the path depth (it is futile to create directories which exist, but since the `mkdir()` function implicitly checks for this condition, explicitly doing so — for example, with `stat()` or `access()` — would be even more inefficient) in a similar fashion to the behaviour of the `mkdir -p` command, followed by one `symlink()` call.

### 5.2.3 Path Removal

Removing a path from a flat resultset is a single `unlink()` operation for one or more than two existing results of that name, but reducing a resultset from 2 to 1 result requires two `unlinks()`, a `rmdir()`, and a `symlink()` operation. For a hierarchical resultset, an `unlink()` is performed, followed by as many `rmdir()`s as there are empty parents in the path (relative to the FindFS root); as with addition, we have utilised the implicit test for non-emptiness of `rmdir()` to automatically remove empty directories, and furthermore its return value to stop further futility upon reaching the first non-empty one.

### 5.2.4 Path Modification

Renaming a component of a path is perhaps the most complex operation that can be performed relative to updating of the results, since it has the potential to alter the paths of a very large number of files. The simplest case, where the path to be renamed refers to a file, can be considered as a combination of deletion and creation,

so that a rename of `/a/b/c/file1` to `/a/d/file2` can be reflected in the resultset as a removal of `/a/b/c/file1`, followed by addition of `/a/d/file2` (provided that the new path still satisfies the query expression and scope.) The complex case occurs when renaming a directory; this affects all resultsets containing paths that involve it, and thus requires iterating over the resultset to both remove entries that no longer match the query and modify entries which still do, but need their links adjusted to target the new directory's path. The entire contents of a flat resultset need to be examined, while a hierarchical one isolates the requires changes to the renamed subtree; a `rename()` operation moves the subtree in the resultset in the same way it modified the original filesystem, and then a recursive traversal through the files it references adds/removes/updates the resultset accordingly. In addition, since renaming effectively deletes a subtree and creates another, it is necessary to re-evaluate all the files in the subtree as their changed paths may now satisfy a query that was not satisfied before.

### 5.2.5 Integrated Algorithm

The operations required are summarised in Table 5.1, and Figures 5.2 and 5.3 depict graphically the algorithms used for flat and hierarchical resultsets, respectively. As it does not make sense to remove a result which does not exist, or add paths that are already present, these algorithms integrate the existence checks with addition or removal, controlled by one variable ("add" in the figures) — depending on this boolean, they either add a path if it does not exist (if it already exists, no action is taken), or remove one that exists (similarly, one that does not exist causes no

| **Operation** | **Resultset** | **Duplicate Names** | **Operations** | **Total** |
|---|---|---|---|---|
| Existence | Flat | 0 or 1 | `stat()`, `readlink()` | 2 |
| | | 2 or more | `stat()`, `opendir()`, $n$ * (`readdir()`, `readlink()`), `closedir()` | $3 + 2n$ |
| | Hierarchical | - | `stat()` | 1 |
| Add (if not exists) | Flat | 0 | `symlink()` | 1 |
| | | 1 | `unlink()`, `mkdir()`, 2 * `symlink()` | 4 |
| | | 2 or more | $k$ * `symlink()` | $k$ |
| | Hierarchical | - | $p$ * `mkdir()`, `symlink()` | $p + 1$ |
| Remove (if exists) | Flat | 1 | `unlink()` | 1 |
| | | 2 | 2 * `unlink()`, `mkdir()`, `symlink()` | 4 |
| | | 3 or more | `unlink()` | 1 |
| | Hierarchical | - | `unlink()`, $q$ * `rmdir()` | $q + 1$ |

($n$ = number of results, $k$ = number of inode number and name collisions; $p$ = depth of path; $q$ = number of 1-child deepest directories in path)

Table 5.1: Resultset Modification Effects

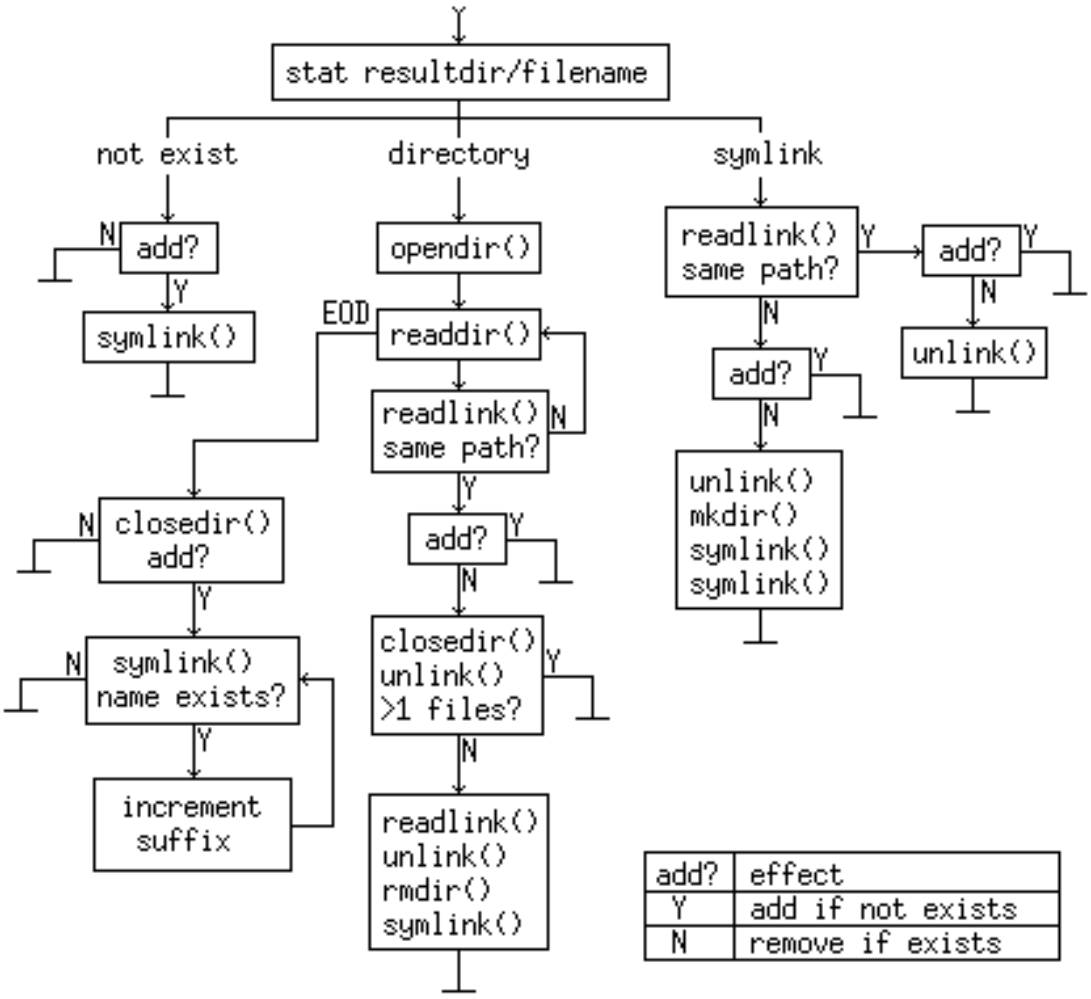


Figure 5.2: Modification of Flat Resultsets

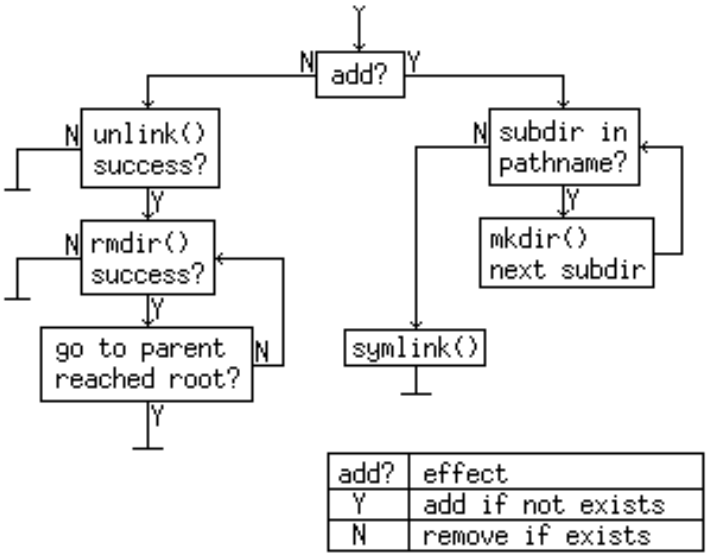


Figure 5.3: Modification of Hierarchical Resultsets

action.) Integration of addition/removal into one algorithm eliminates much of the duplication and work — particularly for the flat resultset — that would otherwise occur as the code paths for these operations are very similar. For example, the number of duplicate results in a flat resultset is partially determined during existence checking, and this is also used when both adding or removing is performed.

Observe that a hierarchical resultset, owing to its similarity in structure to the existing filesystem, requires far fewer decisions to be made than for a flat one; the majority of the complexity in the latter arises from a need to handle the different cases of duplicate names.

# Chapter 6

# Evaluation

Specific benchmarks were performed to characterise the system, to determine the overhead introduced by evaluation of queries and updating of their results. All of the results are obtained with this configuration: Ubuntu 12.04 Linux 3.2.0-24 with FUSE 2.8, i7 920 (2.67GHz), 1GB RAM, 2xWD2003FZEX (7200RPM, 2TB, 64MB cache) HDDs (in RAID 1) running as a VM on a host which is otherwise idle. Unless otherwise specified, the filesystem used in these benchmarks is a small 27.85GB Ext3 partition; while the time taken for the `FIND` state of each query is dependent on the size of the filesystem, in a similar manner to how the `find` utility operates, the benchmarking of that is not the main concern. The performance of resultset updates in response to changes in the filesystem (i.e. the `WAIT` state) is of significantly more interest; we use microbenchmarks to determine the cost of these operations, as well as a macrobenchmark to evaluate the overall performance of the system. Unless otherwise specified, all the results were obtained by averaging time over 1000000 iterations.

## 6.1 Basic FUSE Overhead

Filesystems which utilise FUSE are known to be slower than native kernel-mode implementations [27] due to the additional context switches involved, and the fact that all requests to the filesystem go through a device special file; the purpose of this benchmark is to quantify the overhead of various filesystem operations via FUSE, and compare that to the native (Ext3) filesystem in the benchmarking environment. Note that this also includes the overhead of the user-mode FUSE library, which FindFS relies on. For this benchmark, FindFS is used with no active queries; in this mode, all filesystem activity is passed through transparently. The operations tested are open/close, open/read/close (reading only the first byte of a small file), `stat()`, `creat()`/`close()`/ `unlink()`, `mkdir()`/`rmdir()`, and reading all the entries of a directory containing 20 files. Table 6.1 shows the results.

| **Operation** | **Native** ($\mu$s) | **FUSE** ($\mu$s) |
|---|---:|---:|
| `open()/close()` | 2.44 | 40.2 |
| `open()/read()/close()` | 3.06 | 68.0 |
| `stat()` | 1.10 | 1.17 |
| `creat()/close()/unlink()` | 12.89 | 119.3 |
| `mkdir()/rmdir()` | 54.91 | 153.8 |
| `opendir()/readdir()/closedir()` | 7.86 | 109.4 |

Table 6.1: FUSE Overhead

From these results, it can be seen that file accesses through FUSE are roughly 20x slower than accessing the native filesystem directly, while creation and deletion nearly 10x slower; however, there is a difference of less than 7% with `stat()` operations, and directory creation/removal, at ≈3x, is not quite as slow as open/close. The other operation that FindFS frequently needs to perform for flat resultsets,

reading the contents of a directory, is almost 14x slower. Since all the benchmarks in the following sections include these overheads, it is important to note that even a small relative change in time could appear much greater if FindFS were implemented as a native kernel-mode filesystem instead of with FUSE.

## 6.2 Expression Evaluation

Each (active) query requires processing time, which is comprised of two parts: evaluation and modification. The former involves evaluating the metadata of the file against the query expression, while the latter may add or remove it from the resultset depending on the result of the evaluation. To determine precisely the costs of each of these operations, we use benchmarks which only exercise the respective parts. For expression evaluation, this is achieved by performing filesystem operations that do not cause any resultset modification, while varying the number of queries or their complexity. We timed an open/close and open/read/close operation, for both filename and extended attribute queries in the matching and non-matching cases.

Since the times scaled almost proportionally with the independent variable, linear regression was used to obtain the per-query cost. The results of this benchmark are shown in Table 6.2.

### 6.2.1 Filename Match

This benchmark determines the time taken to evaluate a single expression using the `-name` predicate, which is a typical application of the `find` utility. It represents the use case of FindFS as a "persistent invocation of `find`", and involves accessing a file

| **Match Type** | `open()`/`close()` ($\mu$s) | `open()`/`read()` /`close()` ($\mu$s) |
|---|---|---|
| Match Filename | $5.95 \pm 0.25$ | $10.0 \pm 0.61$ |
| Match EA | $3.32 \pm 0.31$ | $4.37 \pm 0.73$ |
| Non-match Filename | $0.198 \pm 0.072$ | $1.76 \pm 0.02$ |
| Non-match EA | $2.87 \pm 0.52$ | $2.87 \pm 0.49$ |
| Per Filename Term | $-0.242 \pm 0.086^*$ | $0.520 \pm 0.147$ |

* See text for explanation of negative value

Table 6.2: Matching Benchmark Results

existing in the (flat) resultsets of N queries, such that it remains in those resultsets. This performs a `stat()` and a `readlink()` operation on each iteration, followed by a path comparison. As expected, the time scales nearly linearly with the number of queries, but the case in which the file is only opened and closed but not read takes only 60% of the time when it is read; this is likely due to (CPU instruction) cache effects, with the quantity of executed code being greater in the latter case, and in actual use an extra 10$\mu$s would be expected per file access.

### 6.2.2 Extended Attribute Match

When queries are used to create "containers" like a tag-structured filesystem, extended attributes are employed; more specifically, name and value matches. Thus it is important to determine the time required to evaluate such queries. The testing is performed with the same setup as 6.2.1, except the queries use the `-eav` predicate to examine an extended attribute's name and value. A 13-byte extended attribute name (`user.filename`) is used, and matched with a value of 9 bytes, comparable to the filename match in the previous section. In this case, there is the same trend of reading from the file increasing the matching time — giving more evidence in

support of the theory that this is a CPU cache effect — and it can also be seen that retrieving extended attributes and matching on them adds less than 5$\mu$s, meaning that Ext3 extended attribute retrieval is unlikely to be a performance bottleneck for FindFS.

### 6.2.3 Filename Non-Match

Many files in a filesystem, particularly system files and temporary data, are not often the targets of search queries; however, they may be frequently and rapidly accessed by system services during normal operation. For an indexing filesystem they would not be indexed, but since FindFS monitors all operations it is important to determine how accesses to non-matching files are affected. This benchmark is identical to 6.2.1 except that the file being accessed is not in, and will not be added to, any of the resultsets. Additionally, we test the creation and deletion of a non-matching file. The cache effect is strongly visible in these results, adding less than 0.2$\mu$s when the file is only opened and closed, but close to 10x that when it is read; creation and deletion also nearly doubles that number. Nevertheless, an extra 1.76$\mu$s for accessing a non-matching file, or 3.20$\mu$s for creating or deleting, is less than any matching one, following the principle that FindFS should be relatively unobtrusive on files which are not targeted by queries.

### 6.2.4 Extended Attribute Non-Match

This benchmark tests the opposite condition to 6.2.2, and its goal is similar to 6.2.3; files not matching queries utilising extended attributes should not incur significant

overhead when they are accessed. The same query as for 6.2.2 is used, and a file that does not match is repeatedly accessed. This adds the same overhead regardless of whether the file is read, and is relatively close to the times for 6.2.2, indicating that the code path for extended attributes does not differ much between matching and non-matching cases.

Overall, access to files that do not match non-EA queries are significantly faster than those that do, but there is a smaller difference between accessing files matching and non-matching EA queries.

### 6.2.5 Query Complexity

More complex query expressions may take longer to evaluate; this benchmark is aimed at determining how evaluation time scales with the number of terms in a query expression. A single query is used, and access to a matching file timed while varying the number of disjunctive terms in the expression; the `-name` predicate is used for each term, and the matching term is placed at the end of the disjunction chain (`-name fileX.txt -o -name fileY.txt ... -name file_match.txt`) to force thorough evaluation. The obtained result for opening and closing without reading is negative — adding terms to the query actually caused it to become slightly faster, as there was a downward trend in the timing; this is clearly the same caching effect seen in the earlier benchmarks, but when the file is read before closing it, causing more code to be uncached, the per-term evaluation time is still approximately 0.5$\mu$s. Even with 10 terms, which is far more complex than a typical query expected of this system, the time taken is still only 5$\mu$s; this is an indication

| Resultset Type | Condition | Time ($\mu$s) |
|---|---|---|
| No modification | (no match) | $3.20 \pm 0.10$ |
| Flat | $0 \leftrightarrow 1$ results | $17.43 \pm 0.23$ |
| | $1 \leftrightarrow 2$ results | $153.6 \pm 1.4$ |
| | $2 \leftrightarrow 3$ results | $47.98 \pm 0.26$ |
| Hierarchical | depth 0 | $16.06 \pm 0.30$ |
| | depth 1 | $101.7 \pm 1.0$ |

Table 6.3: Resultset Modification Benchmark (`creat()`/`close()`/`unlink()`), times per query-result

that query evaluation comprises a very small portion of the total processing time in FindFS, and thus complex queries with many terms should not cause the system to become unusably slow.

## 6.3 Resultset Modification

Resultset modification occurs following expression evaluation if the existence of the path differs from the evaluation result. The benchmarks in this section aim to elucidate the overhead associated with adding and removing files from a resultset in response to creating and deleting files which satisfy a query expression. Since the operations performed differ between flat and hierarchical resultsets, and also depend on specific conditions of the resultset, we test a few representative resultset conditions intended to exercise the cases shown in Table 5.1, and the times are shown in Table 6.3. As with expression evaluation, we vary the number of matching queries from 1 to 10, causing the same number of additions and removals to occur with each creation and deletion; the times scale linearly, and obtaining the slope of this line with regression gives the per query-result time. The baseline of no resultset

modifications is 3.20$\mu$s, for comparison purposes.

### 6.3.1 Flat

These benchmarks repetitively add and remove a file from the resultset of a query configured with the (default) flat resultset mode, caused by creating and deleting a file elsewhere in the filesystem. A resultset initially containing 0, 1, or 2 files of the same name is used, testing the differences that arise due to duplicate name handling. The results show expected behaviour with adding or removing a uniquely-named file being the fastest since it is a single `symlink()` or `unlink()` call, and transitioning between 1 and 2 duplicate names being the slowest due to requiring multiple manipulations of the underlying filesystem. The times for creating and removing the link are comparable to those taken by the native filesystem for creating and removing a regular file, as shown in Table 6.1.

### 6.3.2 Hierarchical

This benchmark repetitively adds and removes a file from the resultset of a query configured with the hierarchical resultset mode, using a path with 0 and 1 sub-directory levels to cause either none or one directory creation/removal with the creation/deletion of the file. The time taken when there are no directory additions/creations is expectedly similar to that of the flat mode, since it performs the same `symlink()`/`unlink()` pair. However, it is much slower when directory creation/deletion is required; observing that even the native filesystem takes nearly 55$\mu$s, the time it takes is not surprising. The results suggest that hierarchical result-

sets should be avoided for queries intended to match frequently added and deleted files in deeply nested, sparse directory trees.

### 6.3.3 Microbenchmark Summary

Comparing the results in Tables 6.2 and 6.3 with the FUSE overhead in Table 6.1, in particular the `creat()`/`close()`/`unlink()` sequence, it can be seen that a significant portion of the total time spent is FUSE overhead. For example, creating and removing a file takes 119.3$\mu$s, and each query's resultset that is modified due to this operation adds an additional 16.06 to 153.6$\mu$s depending on the type and state of the resultsets. Thus, it is very likely that a kernel-mode implementation of FindFS, although seeming to have higher *relative* overhead, would achieve far better *absolute* performance than the current one with FUSE.

## 6.4 Macrobenchmarks

The benchmarks of the previous section access only a single file and have very small resultset sizes, and thus their frequent accesses to the filesystem will have been cached; this benchmark accesses a sufficiently large number of files such that not all of them can be cached, to investigate the performance of FindFS as it would be in a typical application. For this purpose, we used the Filebench[28] utility with the standard `fileserver` workload. This is a more intensive workload than is typical for a desktop application, but because it contains a variety of file access operations such as reading, writing, creating, and deleting, it is useful for assessing overall performance. The test filesystem is parameterised to use 10000 files, with an

average of 20 files per directory, 128KB per file, and a tree depth of 3.1. This results in a total data volume of 1240.757MB. The runtime parameters are 50 threads, 16KB average write size, and 1MB read size, operating for 5 minutes over which the total throughput is measured. Table 6.4 summarises these results.

| **Configuration** | **Throughput (ops/s)** |
|---|---:|
| Native (ext3) | 191.57 |
| FindFS, no queries | 184.85 |
| FindFS, 5 flat queries | 160.92 |
| FindFS, 5 hier queries | 152.72 |

Table 6.4: Filebench Throughput

### 6.4.1 FUSE Overhead

As with the microbenchmarks, it is useful to obtain a baseline of the expected performance of the native filesystem in the test environment and compare that with the overhead that FUSE adds; as before, the latter is accomplished by running the workload on the filesystem through FindFS with no active queries. We obtain 96.5% of the throughput of the native system, which likely indicates that the underlying storage device is becoming the bottleneck as most of the accessed data can no longer be cached.

### 6.4.2 Flat Resultset

In order to determine the effect of active queries on overall performance, we create 5 queries, each encompassing a non-disjoint subset of the files, and run the same workload after they have entered the `WAIT` state. The resultsets total to approxi-

mately 84% of all the files in the filesystem, and follow the same distribution of sizes. They are then added to and removed from by the file operations of the workload. The resulting throughput is 87% of that achieved without any queries, or 13% less. This is unlikely to be a problem in a typical desktop application where file operation frequencies are low.

### 6.4.3 Hierarchical Resultset

We expect that a typical usage will have a mix of resultset modes, but due to the slower modification of a hierarchical resultset, the lowest throughput can be observed if all active queries use them. This benchmark uses the identical queries and workload access patterns as the previous one, with the exception that the queries are configured to use a hierarchical resultset. The obtained throughput, 83% of that achieved without any queries, is only 5% less than that with 5 flat queries, and so we expect the typical overhead to be in the 83 to 85% range; this is not a large difference, and on the occasion that the overhead of FindFS does cause a significant impact on performance, such as an unusually intense sequence of filesystem operations, queries on the affected subtree(s) can be easily made inactive (`STOP` state) and later commanded into `FIND` to "batch-update" the resultsets afterward.

## 6.5 Discussion

Although the microbenchmarks show that operations which cause resultset modifications under FindFS become 5x to 48x slower due to the additional filesystem operations, with a more general workload their impact is reduced significantly to

under 15%. Opening and closing a file while reading very little data from it, or creating and deleting files, which show the worst-case behaviour, are operations seldom performed at such extreme frequencies on a typical desktop application or even a file server. It is even rarer that these rapid changes would cause repetitive addition and removal from a resultset. In typical applications, overall throughput is constrained more by storage access times — as the amount of data read or written increases, the percentage of time spent opening and closing the file decreases; since FindFS transparently passes reads and writes through to the underlying filesystem, those operations which account for the majority of time in a typical workload are not affected. This trend agrees with the FUSE benchmarks in [27].

# Chapter 7

# Conclusion

FindFS is a filesystem-based extension which allows adding tag-based views to a traditional hierarchical filesystem, making it easier to organise files and access them in ways that their metadata naturally suggest, without requiring significant changes to existing applications. The overhead added by monitoring filesystem operations and modifying resultsets can be relatively small, and thus not have great impact on perceived performance in applications; even the filesystem-intensive use-cases can benefit from the flexibility and simplicity of FindFS' approach to on-demand persistent search queries with its fine-grained control over query processing.

FindFS retains backwards-compatibility and provides the familiarity of selecting files using `find` and creating symbolic links to allow building alternate hierarchies, but also leverages extended attributes and dynamic resultset updates to give users the more powerful and convenient ways to group and find files which semantic filesystems are known for.

*The source code of the FindFS prototype is available at* `http://www.cs.ubc.ca/labs/dsg/findfs/` *. FindFS is usable on platforms with FUSE, a POSIX-compatible API, and filesystems supporting extended attributes.*

# References


[1] Gifford, D.K., Jouvelot, P., Sheldon, M.A., James W. O'Toole, J.: Semantic file systems. In *SOSP '91, Proc. of the thirteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, ACM Press (1991) 16-25.

[2] Seltzer, M., Murphy, N.: Hierarchical file systems are dead. In *HotOS '09, Proc. of the 12th conference on Hot topics in Operating Systems*, Berkeley, CA, USA, USENIX Association (2009) 1-1.

[3] Ngo, H.B., Silber-Chaussumier, F., Bac, C.: Enhancing Personal File Retrieval in Semantic File Systems with Tag-Based Context. In *EGC 2008, Revue des Nouvelles Technologies de l'Information*, RNTI E-11, Cepadues, Volume I, (2008), 73-78.

[4] Amoson, J., Lundqvist, T.: A light-weight non-hierarchical file system navigation extension. In *Proc. of the 7th International Workshop on Plan 9*, Dublin, Ireland, Bell Labs (2012) 11-13.

[5] Voit, K., Andrews, K., Slany, W.: TagTree: storing and re-finding files using tags. In *Proc. of the 7th conference on Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society: information Quality in e-Health*, Graz, Austria, (2011) 25-26.

[6] Bloehdorn, S., Gorlitz, O., Shenk, S., Volkel, M.: TagFS - Tag Semantics for Hierarchical File Systems. In *I-KNOW 2006, Proc. of the Sixth International Conference on Knowledge Management*, Graz, Austria, (2006), 304-312.

[7] Gopal, B., Manber, U.: Integrating content-based access mechanisms with hierarchical file systems. In *OSDI 1999, Proc. of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, USA, USENIX Association (1999) 265-278.

[8] Ballesteros, F.J.: File indexing and searching for Plan 9. In *Proc. of the 4th International Workshop on Plan 9*, Athens, Georgia, (2009) 131-138.

[9] Padioleau, Y., Ridoux, O.: A Logic File System. In *USENIX Annual Technical Conference*, USENIX Association, 2003.

[10] M.A. Olson.: The Design and Implementation of the Inversion File System. In *Proc. of the Winter USENIX Technical Conference*, San Diego, California, USA, USENIX Association (1993) 205-217.

[11] Apple Spotlight Search - `https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html`, June 2015

[12] Windows Desktop Search - `http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx`, June 2015

[13] Saved Search File Format (Windows) - `https://msdn.microsoft.com/en-us/library/windows/desktop/bb892885(v=vs.85).aspx`, June 2015

[14] OS X Mavericks: Create or modify a Smart Folder - `https://support.apple.com/kb/PH14047`, July 2015

[15] `symlink` - The Single UNIX(R) Specification, Version 2, The Open Group, 1997

[16] `link` - The Single UNIX(R) Specification, Version 2, The Open Group, 1997

[17] `find` - The Single UNIX(R) Specification, Version 2, The Open Group, 1997

[18] `mdfind(1)` - OS X Man Pages - `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/mdfind.1.html`, July 2015

[19] `xattr(7)` - Linux Man Pages - `http://man7.org/linux/man-pages/man5/attr.5.html`, June 2015

[20] File Metadata Query Expression Syntax - `https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/SpotlightQuery/Concepts/QueryFormat.html`, July 2015

[21] Ritchie, D.M., Thompson, K.: The UNIX Time-Sharing System. In *Communications of the ACM*, Vol. 17 No.7, July (1974) 365-375.

[22] Ousterhout, J.K., Welch, B.B.: Pseudo-devices: user-level extensions to the Sprite file system. In *USENIX Association Summer Conference Proceedings*, San Francisco, California, USA, USENIX Association (1988) 37-49.

[23] Pike, R., Ritchie, D.M.: The Styx Architecture for Distributed Systems. In *Bell Labs Technical Journal*, Vol 4, No 2, April-June (1999), 146-152.

[24] Xu, Z., Karlsson, M., Tang, C., Karamanolis, C.: Towards a Semantic-Aware File Store. In *HotOS '03, Proc. of the 9th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, USA, USENIX Association (2003) 31-31.

[25] Dekeyser, S., Watson, R.: Metadata manipulation interface design. In *Proc. of the Fourteenth Australasian User Interface Conference*, Melbourne, Australia, (2013) 33-42.

[26] FUSE: Filesystem in Userspace - `http://fuse.sourceforge.net/`, April 2014

[27] Rajgarhia, A., Gehani, A.: Performance and Extension of User Space File Systems. In *Proc. of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, ACM (2010) 206-213.

[28] Filebench File System Benchmark - `http://filebench.sourceforge.net/`, May 2015