MetaView: Dynamic metadata based views of user files

James Bunton
School of IT
The University of Sydney
NSW 2120 Australia
jamesbunton@fastmail.fm

Judy Kay
School of IT
The University of Sydney
NSW 2120 Australia
judy@it.usyd.edu.au

Bob Kummerfeld
School of IT
The University of Sydney
NSW 2120 Australia
bob@it.usyd.edu.au

Abstract *Hierarchical file systems are the most common way of organising large collections of documents. However, there are several desirable features they do lack. These include: good support for placing files in multiple locations; dynamic views on the users' data; and explicit ordering of files. This paper introduces MetaView, a new approach to enhancing file systems so that they can present users with a fluid and dynamic view of their files based on metadata. MetaView allows users to describe how they wish to view their files by specifying an organisational structure based on a metadata path. Experiments indicate that this approach is viable for collections of up to several thousand files in size, enabling flexible organisation of substantial parts of a user's file system.*

Keywords Document Management, Metadata, Non-hierarchical file system.

1 Introduction

File system organisation is a very important area of computing. Millions of people use computers daily to store critical, often irreplaceable data. This includes text, spreadsheets, photos, videos, music and many other types of documents. Most of this data is stored as ordinary files in a traditional file system.

The file systems provided by the commonly used desktop operating systems [1] are implementations of the hierarchical model. From a conceptual perspective, this is essentially the same model used in the 1969 version of Unix [17]. In a hierarchical file system, users can create folders and subfolders to categorise their files, with an arbitrary level of nesting. This structure can be browsed to locate existing files and create new ones. Once a file is located, applications can read from and write to it.

There are some well known limitations with this system which MetaView aims to address. One of these is that operating systems have a deep model that means each file may only be saved in one location, even if it logically belongs under two folders for the purposes of categorisation that suit particular user tasks. Additionally the burden of organising files is placed on the user; switching to a different categorising scheme can require much tedious, manual effort to move files around.

Consider the example where a user has a collection of music which comes with extensive metadata. The user may wish to organise this music by genre, artist and then title. Many songs fall under multiple genres. However, there is no well supported, easy mechanism enabling the user to express this in existing hierarchical file systems. The operating system provides poor support for the case where a user organises files initially in one structure, say artist, but then later decides they would like to switch to viewing their music files by genre and then title. The larger the collection of files, the more onerous this task becomes.

While most operating systems support links as a method of extending the hierarchical model, these have several problems. Firstly, there are so many different kinds: Windows shortcuts, Unix symlinks and hardlinks as well as Mac OS aliases. Each of these has different semantics with positive and negative tradeoffs and none of them meets both the goals described above of allowing files to be organised automatically for users and seamlessly saving files allowing a file to exist in multiple locations.

MetaView's approach is to provide users with a mechanism to specify the structure of a "view" in which they want their files to appear. A user may tell the system to make use of metadata to display all of their music in a format such as: `pop/The Beatles/Yellow Submarine.mp3` where the genre is first, then the artist, followed by the title. The user could later choose any other organisation structure that suits their needs simply by creating or updating a "view". The same user may occasionally wish to browse by artist, then genre, so their music would appear in this format: `The Beatles/pop/Yellow Submarine.mp3`. This applies equally for any types of files, including collections of completely different file types, for which automatically collected or user-specified metadata is available.

Importantly, MetaView allows any existing application to use the standard open/close/read/write file API to access document, making use of metadata as in the examples above. This means the applications can op-

[1] Linux, Mac OS and Windows.

Proceedings of the 13th Australasian Document Computing Symposium, Hobart, Australia, 8 December 2008.

erate consistently with the user's mental model at the same time as allowing developers to use the existing well-known and understood tools for interacting with the file system.

The ultimate vision is that rather than specifying a location on disk when saving a file, users would have the option to tag the file with metadata, or simply save it. The system will collect the optional user-specified metadata, as well as automatically extracted metadata and use this to display the file in any appropriate views which the user defines. This relieves users of the burden of organising their files manually, but still allows for the familiar and powerful browsing interface. MetaView currently implements the "view" part of this interface; changes would be required to the operating system's save-dialogue API in order to support this alternate saving behaviour in graphical applications.

2 Related Work

There have been several studies of how users work with their personal data, such as [4, 18, 5, 16, 12]. These point to some of the limitations in current systems causing difficulties for users in organising documents within multiple locations in file hierarchies as well as the lack of support for the user to define an arbitrary, custom ordering of files within a directory. Reflecting the recognition of the limitation of the prevailing hierarchical file systems, both the research community and commercial organisations have explored alternative approaches designed to improve the situation. We now consider some key examples.

One class of systems has taken the approach of enhancing search functionality. These include SFS [11], BeFS [10], Connections [19], LISFS [15] and Spotlight [3]. All of these systems provide alternate ways of accessing the information stored in a hierarchical file system. However, even this facility does not address the basic goals of our work, to overcome the limitations of hierarchies. These search tools operate in the context of the existing hierarchical file systems and still leave the user with the burden of manually reorganising files if the user wants to organise documents into a different hierarchical view.

By contrast, some research systems have attempted a complete breakaway from the hierarchical model. Examples include LifeStreams [8], Presto [7], Placeless Documents [6], PosCFS [13] and LIFS [2].

Selected systems particularly relevant to MetaView will now be described in greater detail.

2.1 SFS

The Semantic File System [11] was the first implementation of a file system that was semantic in that it provided virtual directories for queries based on metadata extracted from files. For example, it made use of author, title or words contained in the file. The way that it worked was that a virtual directory was created on request, the name of the directory being the query. This virtual directory concept allowed for compatibility with all existing software.

A process called a transducer ran automatically in the background to keep the store of metadata up to date with the contents of the files. The virtual directories were provided using a custom NFS server.

The evaluation of this system was primarily focused on system performance. The amount of disk space required for the metadata store and its index was determined. The total time to index files as well as incremental updates to the index were also measured. The authors concluded that the performance and space requirements of realtime indexing were not overly taxing for the file server. The system was found to be useful for sharing files with other research groups, although there was little detail of this. It was also noted that in the case of file types for which no transducer had been written, these files were difficult to locate.

2.2 LISFS

The Logical Information Systems File System (LISFS) [15] is also an implementation of a semantic file system. Like SFS 2.1, queries are supported as virtual directories.

The system is implemented as a Linux VFS plugin. Transducers operate in the background to collect metadata and update indices.

To make a query, a directory path is constructed out of expressions about the collected metadata and can include logical constructs such as 'and', 'or', 'not'. Queries can be constructed incrementally. At each stage in the query the user may choose to look at the result set so far, or at a list of possible extensions to the query. This allows for a kind of browsing with flexible organisation of files, as the user proceeds.

Also supported is the ability to make a query within a file, for example queries within a BibTex database will return a portion of that file. The result set can be edited and changes propagate back to the original in the way that one would expect. This aspect of the system enables the user to think about abstract *documents* even when these are actually stored within a single file.

No formal evaluation of the system was mentioned in the paper, although several examples were given of the system in use. It operates on several file types, including mp3 collections, source code, BibTex files and email. LISFS is important in that it extends the file system metaphor to allow more powerful and flexible access to data in a manner compatible with existing software.

2.3 LiFS

The Linking File System (LiFS) [2] extends the hierarchical file model to include the concept of links between files. The authors point out that many applications have developed their own systems of storing and searching file metadata, including links. The main shortcoming of this approach is that any one computer system will

have several incompatible stores of metadata using a different interface.

The solution given by the authors is a file system that includes support for traditional file metadata as well as links between files. These links are intended to allow desktop search tools to detect importance, relevance and relations between documents in a similar manner to the way web search engines use hyperlinks in web pages.

New system calls were added to support creating, reading and modifying links between files. Unfortunately, this means that LiFS compromises compatibility with existing applications.

The file system is designed to operate on high speed, high capacity non-volatile memory. Several tasks were performed with LiFS on standard volatile RAM, compared with ext2 and XFS. LiFS outperformed these systems in metadata access, and was close in other areas such as creating/deleting files and read/write operations.

2.4 Connections

Connections [19] is a desktop search tool. It works similarly to content search tools like Spotlight [3], but augments the results with contextual information gathered from monitoring user activity.

The system traces all file activity by the user, building up a relation graph between files. Links are defined between files which were accessed at similar times where these links represent a weighted relationship. This graph is analysed, to discard irrelevant, and append relevant entries to the standard content search as well as to help rank these results.

The user evaluation performed by the authors showed that Connections improved both average recall and precision over a standard content search. Additionally the performance impact of collecting the trace data and analysing it to build the context graph was found to be minimal.

2.5 Presto

The Presto [7] system's primary method of locating documents is through "collections", where these are live queries over document metadata combined with an explicit document inclusion and exclusion list which the user may manipulate. Collections may also be nested.

One particularly novel feature is personal metadata. When attaching metadata to a shared file, a user may decide to keep that metadata private. This could be useful for marking files as "interesting", metadata that is not necessarily relevant to other users of that document.

Presto sourced documents from many locations including the local file system, network shares, web sites, email clients, etc. These data sources were implemented as plugins, each of which was responsible for allowing read and write support to its respective source data, as well as extracting metadata.

Two APIs were provided by Presto. A custom NFS server acted as a compatibility layer intended for existing applications. The primary API allowed full access to all of the functionality offered by the system.

A custom document browser was built in the primary API to allow users to create and manipulate collections and the files within them. When a user wishes to work with a document in a Presto-unaware application, that application is launched and given the path to a file on the NFS server.

There was no formal evaluation. However, the authors reported that their goal, creating a system where attribute-oriented access was the primary method of document interaction, was successful.

3 Approach and Overview

Many of these novel approaches to explore alternatives to the prevailing hierarchical file system structure use virtual directories, whose path name constitutes a form of search query string, as a user interface for finding files. By contrast, our approach in MetaView is to create views, which are populated with directories and files such that the path to a file describes its contents according to the metadata which the user is interested in. If we think of the full path name of a file in a conventional hierarchical file system as an ordered series of metadata tags, essentially MetaView makes it possible to generalise that to allow other orderings of the metadata to create different views.

To implement this, we considered several possibilities and implemented one based upon symbolic links on a regular file system. This means that MetaView retains full compatibility with all existing software and requires no kernel modules or modifications. This makes installation simple and means that users do not need to trust their collection of files to an unknown file system.

Like Presto, MetaView attempts to provide an alternative to the hierarchical file system for organising and retrieving files. Presto's "collections" are similar in purpose to a "view"; however, there are some important differences. Where a collection is a flat list of files which may contain other collections, a view consists of a possibly nested set of files and folders kept in a structure managed by MetaView.

The following figures are screenshots from the Mac OS X Finder. Each shows a different view of the same collection of files, as described below. The goal of MetaView is that users should be able to specify arbitrary views over their files as they wish, reflecting their immediate needs for different structuring of their files. For example, a user may decide to have several views of their music files. Let us suppose that the first is a flat list of files as shown in the small subset of a user's music files in Figure 1. This shows just the first few of a large number of music files. Each of these has metadata containing various attributes describing the file.

We now illustrate MetaView's power to enable the user to automatically reorganise this set of files for more

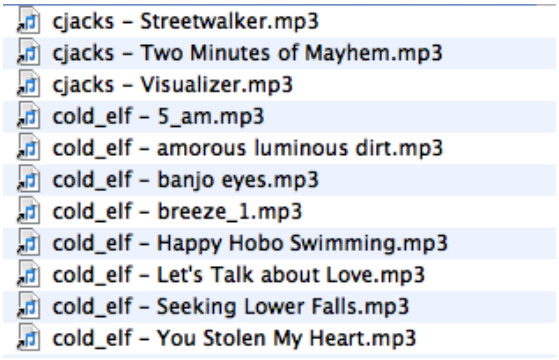

Figure 1: `Artist-Title.mp3`

convenient browsing. For example, suppose that over time, the user has amassed large numbers of these music files and at some stage, they decide to browse their music, thinking of it in terms of the genre first and within this, the artist. MetaView enables them to issue a command to define this new view as illustrated in Figure 2. Note that in this case, many files fit into multiple genres and the view takes care of this in the way that the user would reasonably expect, placing the music file into each genre folder it belongs in. This can be seen in Figure 3, the file 'Flat Ed - Growing Crows.mp3' appears under both the 'country' and 'indie' genres.

Figure 2: `Genre/Artist-Title.mp3` #1

Now we may suppose that at a later time, the user decides they want to be able to browse their music in terms of a different organisation, this time making the artist the first aspect and within that the title. This is illustrated in the example shown in Figure 4. Once again, some music files have multiple artists and the view will handle this correctly.

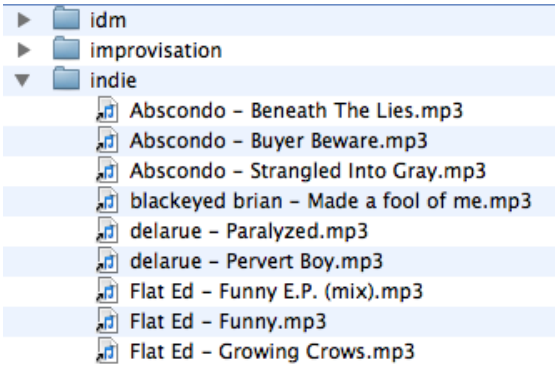

Figure 3: `Genre/Artist-Title.mp3` #2

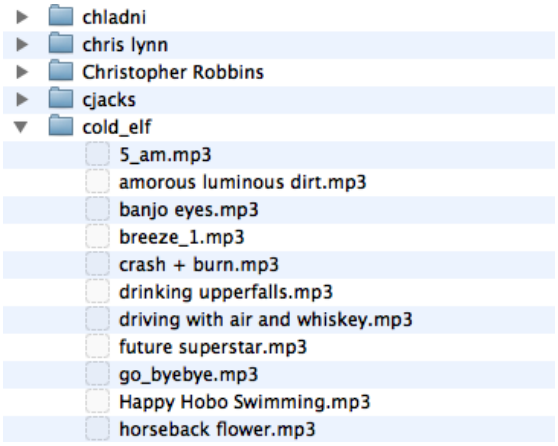

Figure 4: `Artist/Title.mp3`

The final example we will consider is of a combination of the previous two, browsing by genre, then artist with the filenames being the title of the song (Figure 5). This demonstrates the power of MetaView; users can easily construct these alternate views of their files according to their needs and wishes, enabling them to adapt their file structure to changes over time.

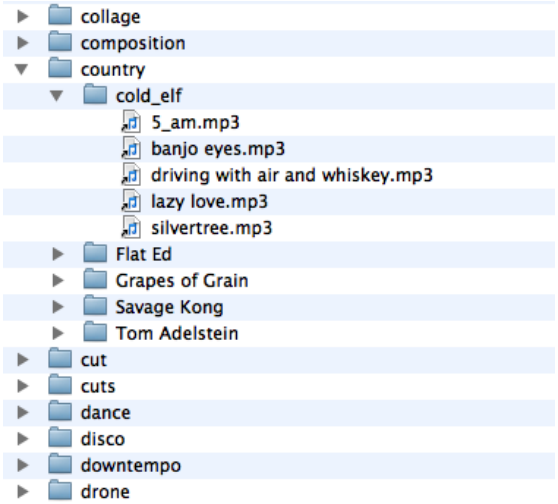

Figure 5: `Genre/Artist/Title.mp3`

4 Architecture

File metadata can be thought of as an extension to the hierarchical model. For our purposes, file metadata is both structured data extracted from a file as well as annotations or tags that a user may provide. For example, an email message may have the Subject, From, To, etc headers extracted as metadata and users may tag photos with the names of people in them.

MetaView has been implemented to run under Mac OS X 10.5, making heavy use of the metadata capabilities of the Spotlight API.

Apple's Spotlight was chosen for this project due to its wide support for existing file types and relative maturity compared to other systems. Spotlight has "importers" for many file types, these are called to examine a file by Spotlight and extract any metadata from it. This metadata is then stored and indexed in the Spotlight database. Apple provides good documentation on writing programs that make use of this database as well as writing new importers to support additional file types. MetaView can be used with any file type supported by Spotlight, these include: email, audio (MP3, AAC), office documents (Microsoft Office & OpenDocument), images (JPEG, PNG) and many others. There is also a commonly used technique that allows Spotlight to index individual documents even when they are all stored in a single file. In addition to the automatically extracted metadata, users may also annotate files with their own custom metadata such as tags for project names, or to mark a file as 'todo'.

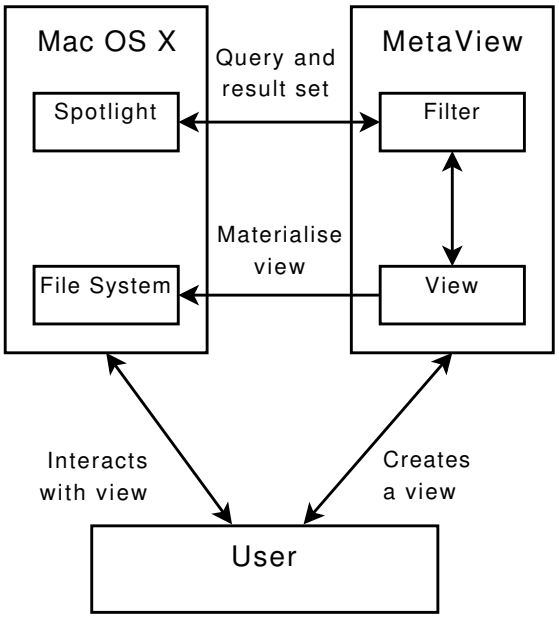

Figure 6: MetaView architecture

The system is implemented in two parts; the search filter and the metadata view (Figure 6). MetaView is written in Python and uses Apple's Spotlight API for searching and access to file metadata.

As input to MetaView, users provide an optional search query and a view specification which they wish the search results to be displayed in. The search query may be as simple as restricting the view to files in a particular directory, or it may be omitted entirely.

An example query for all MP3 files on the system would look like:

```
kMDItemContentType == 'public.mp3'
```

This query is expressed in the standard Spotlight query language. It could easily be constructed by users interacting with an application, such as the Mac OS X Finder.

The metadata view is the most important part of MetaView. In the case of this music example, metadata is extracted by Spotlight from the ID3 tag of the MP3 files. This is a standard tag format that is included with most MP3 music files. Users specify how they want to view their files in the form of a structured hierarchy with different metadata at each level. A view specification for browsing music by genre, artist and then title is expressed as:

```
$(genre)s/$(artist)s/$(title)s.mp3
```

View specifications are Python format strings that are evaluated with respect to each file to be placed in the view. This too could be made intuitive for users to formulate, with the support of a graphical interface.

MetaView creates a Spotlight search for each view that has a unique query. Spotlight then gives a list of query results which MetaView uses to populate a directory with links to the original files according to the view specification. In this way the user's existing workspace is not disrupted at all but they can still take advantage of the advanced functionality the MetaView offers.

It is important to note that the user's files remain on their existing filesystem and can still be accessed in the usual way if so desired. The view is implemented as symlinks to the original files in a directory managed by MetaView. As the user works on their system, creating, deleting and editing files, MetaView keeps the view up to date in real time.

Spotlight notifies MetaView whenever there is a change to any of the files that are being watched, whenever a new file becomes relevant and whenever an existing file is no longer relevant. This notification consists of a list of all the files which currently match the search results. MetaView takes this list and performs a `stat()` on each file to check its last modification time and sorts the Spotlight list into removed files and added files. A changed file is removed and then re-added.

These lists are then passed to the 'view' component which removes all links to the removed files and then inserts links to the added files into the appropriate places. By this process unchanged files are left in place rather than recreated each time Spotlight sends an updated result set.

The user may interact with the constructed view using any existing software. Views can be browsed from the command line or using the Finder. As symlinks are used, opening any link from any application causes that application to work on the original file.

In this way users can create views of their file collection and work with them using their existing software.

The concept of a view, providing flexible organisation of a users' file collections, not specific to any particular operating system. It is a generalisation of existing hierarchical file systems and is a generic concept that could be implemented under any operating system. However the prototype discussed is tied closely to Mac OS X. For a Linux version, the Spotlight metadata backend could be replaced with Strigi [1] or Tracker [9] while the symbolic link implementation of views would remain unchanged. To port MetaView to Microsoft Windows would require larger changes; both the view and the search filter components would need to be rewritten.

5 Scalability Evaluation

Since MetaView is designed to support new ways to organise collections of files, it is important to assess whether it can do this efficiently and to have an understanding of the scalability as the numbers of files grows. We need to assess the scalability of MetaView for its two main actions, constructing a new view and updating a view after relevant changes in the part of the filesystem managed by MetaView. A relevant change is a modification to a file's data or metadata such that the file needs to be added to the view, removed from it or moved within it. We now describe two forms of scalability analysis, the first analytical, based only on the operations required and the second empirical, based on results with actual sets of files.

5.1 Analytical

One of the key costs of handling changes to the managed file set is due to our use of Spotlight for the current implementation. The Spotlight API does not give a list of changes to search results. This imposes a small constant time cost for each file in the result list whenever a change is made. For each change to the view, there may be several calls to the more expensive `mkdir()`, `rmdir()`, `symlink()` or `unlink()` as well as the cost of extracting metadata attributes from the Spotlight API in order to effect the update. For each update posted by Spotlight, there is a cost $O(Sn + Um)$; where n is the number of files in the result set, m is the number of changed files, S is the cost of a `stat()` and U is the cost of updating the view for a single file.

If the Spotlight API were enhanced to give incremental updates to search results, rather than posting the entire result set each time, this performance could be improved. The complexity would drop to $O(Sm + Um)$, a dramatic improvement, especially for the important case where there is a large set of files and there is a change to a small number of them.

The cost of creating the view is the expected $O(Un)$.

The cost of using MetaView would remain the same for any file type supported by Spotlight. This is because every Mac OS X computer has already indexed every Spotlight-supported file on the system, and the cost of extracting metadata from the Spotlight database is independent of the original type of the file.

5.2 Empirical

For this test, we chose to use a set of files that could be readily assembled and where we could also gain metadata that would be useful for creating structures. We chose to use a collection of creative commons licensed music which could be used freely by others to repeat the experiment. The evaluation was conducted with a 10GiB collection of music from opsound.org [14] with 1836 MP3 files. For each of these files Spotlight allows access to metadata such as artist, title and usually multiple genres. The testing was performed on an Intel iMac running Mac OS X 10.5.5 with a Core 2 Duo 2.16 GHz processor and 1GiB of memory.

The cost of a single `stat()` system call was measured to be an average of approximately 0.0015 seconds over 1000 uncached files. This drops to about 0.00001 seconds for 1000 cached files. However in general, we would expect that the files that MetaView will be performing a `stat()` upon will not be cached, giving the slower time as the expected average. This corresponds to the S variable in the analytical analysis.

For the majority of cases in a large collection of files, updating the view will be a matter of a `symlink()`, `unlink()` or both for each file that needs to be updated. Over an average of 1000 files, the cost of a `symlink()` and an `unlink()` was measured to be 0.0003 seconds. Note that for each file that needs to be placed in several locations, this will require multiple system calls. So the actual time for the update will depend upon this.

MetaView took 20 seconds on average to construct a view in the format: `Genre/Artist/Title.mp3`. To update the view after changing one file took 3 seconds on average. Almost all of this time was spent in the `stat()` call, required to determine which file in the list from Spotlight was the changed one.

Additional tests were performed with different sized file collections, these can be seen in Figure 7 and 8. All tests were performed 3 times while the system was under no load and the result was averaged. In all cases there was less than one second difference between the trials.

The difference in time between the two views reflects the number of links that must be created in each case. In Figure 8 exactly one link is created for each

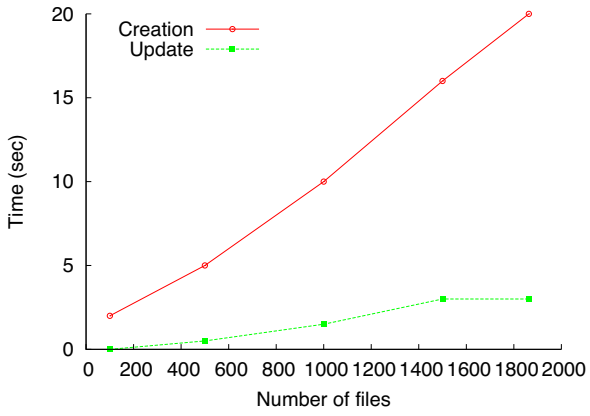

Figure 7: Results: `Genre/Artist/Title.mp3`

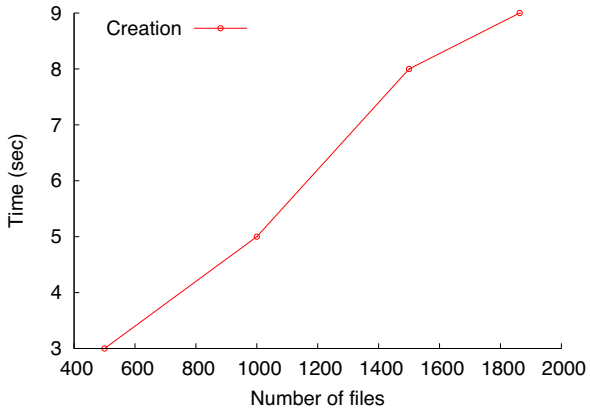

Figure 8: Results: `Artist - Title.mp3`

file, compared with Figure 7 where each file appears in several genres.

The empirical results match up with analytical analysis showing that MetaView scales linearly with the number of files it manages.

6 Conclusions and Future Work

A major performance cost is processing the complete set of results that Spotlight posts whenever a change is detected in even a single file. MetaView could use the FSEvents API to watch for changes to files on the system, in this way it would know which files have been changed in a Spotlight result set without resorting to a `stat()` of each file. This approach adds the cost of processing each file change event on the system, which may even be more work. A better solution would be possible if Spotlight optionally provided updates to existing result sets rather than reposting the complete list of files each time.

Presto provides users with the ability to add and remove files from a collection and have these actions stay persistent. MetaView nearly gets this ability for free by using the regular file system. If a file is added to a view or removed from a view by another program, MetaView will simply ignore it. However additional support for this would be useful. For example, the action of placing a file into a directory which contains files tagged as "todo" could tag that file. This means it would appear in other views that show files tagged as "todo".

There are several promising directions for future work. One of these would provide support for "bundles". These are groups of files which should be treated as one logical unit. Particularly on Mac OS X, many applications create directories full of files that appear to the user as one bundle. MetaView should also treat this as one when populating views and not delve inside the logical unit.

A complementary direction would support users in creating views of abstract "documents" which are stored within a single file. One important example of this is the standard mbox file which contains a collection of mail items, each of which the user may think of as a separate document. Another class of example is a file containing a collection of consistent elements, such as bibtex entries. Since the user may wish to organise these virtual documents into different structures, it would be valuable for MetaView to be able to operate at this level. The simplest way to do this is to make these abstract documents appear to Spotlight as individual files. Many Mac OS X programs already use this technique, including Apple's Address Book for contacts as well as Safari for bookmarks and history. Stand-in files are created for each abstract document with just enough information to allow the owner application to find the actual content.

The system could be further integrated with applications by giving users the ability to apply custom metadata to files when saving them. A save dialogue that allowed users to tag files, displaying a list of commonly used tags, might be a good way to achieve this.

MetaView represents an exploration of a new mechanism for supporting flexible organisation of personal information, in terms of arbitrary sets of hierarchical organisations of documents. With MetaView, users can flexibly create views of their file system where these views structure the documents in the ways that suit the user's current needs, even if this was not anticipated when the files were first saved and organised. These views are automatically updated as the contents of the file system changes, keeping the user's workspace up to date and organised with minimal effort on their part. Importantly, the user's existing file system organisation is left untouched by MetaView, allowing a user a safe entry point, so they can use MetaView without altering their existing work practices and without the risk of being unable to use the multitude of existing operating system services.

MetaView provides a flexible and adaptable new means to organise and access files. While it is consistent with the mental model of traditional hierarchical file systems that most users are familiar with today, it enables the user to extend that same mental model to arbitrary new organisation possibilities, while preserving compatibility with existing software and work practices.

References

[1] Strigi - the fastest and smallest desktop searching program. http://strigi.sourceforge.net 2008.

[2] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller and Scott A. Brandt. Lifs: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2006.

[3] AppleComputer. Working with spotlight. http://developer.apple.com/macosx/spotlight.html 2006.

[4] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: file organization from the desktop. *SIGCHI Bull.*, Volume 27, Number 3, pages 39–43, 1995.

[5] Richard Boardman, Robert Spence and M. Angela Sasse. Too many hierarchies? the daily struggle for control of the workspace. In *Proceedings of HCI International 2003*, pages 616–620, New Jersey, USA, 2003. Lawrence Erlbaum Associates.

[6] Paul Dourish, W. Keith Edwards, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry and James Thornton. Extending document management systems with user-specific active properties. *ACM Transactions Information Systems*, Volume 18, Number 2, pages 140–170, 2000.

[7] Paul Dourish, W. Keith Edwards, Anthony LaMarca and Michael Salisbury. Presto: an experimental architecture for fluid interactive document spaces. *ACM Transactions Computer-Human Interactaction*, Volume 6, Number 2, pages 133–161, 1999.

[8] Scott Fertig, Eric Freeman and David Gelernter. Lifestreams: an alternative to the desktop metaphor. In *CHI '96: Conference companion on Human factors in computing systems*, pages 410–411, New York, NY, USA, 1996. ACM.

[9] Gnome Foundation. Tracker - a personal search tool and storage system. http://live.gnome.org/Tracker/WhatIsTracker 2008.

[10] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, Inc., 1999.

[11] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon and Jr. James W. O'Toole. Semantic file systems. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 16–25, New York, NY, USA, 1991. ACM.

[12] William Jones, Ammy Jiranida Phuwanartnurak, Rajdeep Gill and Harry Bruce. Don't take my folders away!: organizing personal information to get things done. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1505–1508, New York, NY, USA, 2005. ACM.

[13] W. Lee, S. Kim, J. Shin and C. Park. Poscfs: An advanced file management technique for the wearable computing environment. *Lecture Notes in Computer Science*, Volume 4096, pages 966, 2006.

[14] OpSound.org. http://opsound.org 2008.

[15] Yoann Padioleau, Benjamin Sigonneau and Olivier Ridoux. Lisfs: a logical information system as a file system. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 803–806, New York, NY, USA, 2006. ACM.

[16] Pamela Ravasio, Sissel Guttormsen Schär and Helmut Krueger. In pursuit of desktop evolution: User problems and practices with modern desktop systems. *ACM Transactions Computer-Human Interaction*, Volume 11, Number 2, pages 156–180, 2004.

[17] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, Volume 17, Number 7, pages 365–375, 1974.

[18] C.A.N. Soules and G.R. Ganger. Why can't i find my files? new methods for automating attribute assignment. *Proceedings of the 9th conference on Hot Topics in Operating Systems*, Volume 9, pages 20–20, 2003.

[19] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. *SIGOPS Oper. Syst. Rev.*, Volume 39, Number 5, pages 119–132, 2005.