Birch: A Metadata File System

Casey Marshall
University of California, Santa Cruz
CMPS 221: Advanced Operating Systems*
csm@soe.ucsc.edu

November 25, 2006

Abstract

We preset a dynamic file system for Mac OS X that renders metadata search results as files and directories, and describe the issues that arose in developing this system, and the lessons learned about the usability and performance of such a system.

1 Introduction

The advent of computers into our lives has brought a mixture of both endless possibility and intense confusion. While we now have the ability to store and use a myriad of different file types for anything we may need — text files, images, videos, music, each with dozens of file formats — our ability to manage and find things we need has gone down as the number of files we use goes up. The common way computer users organize media is through a hierarchical file system, that stores data (files) in lists (directories); the file system is, then, a list of lists, forming a rooted tree. This model is an excellent, simple way to organize data, and it has served computer users well. The issue with hierarchical file storage is that it depends on the user to create and manage these hierarchies, and so the quality and usability of a particular hierarchy is only as good as the effort put into defining that hierarchy. Good classification schemes are difficult for the average computer user to define — we have historically relied on professionals to organize large collections of data. Furthermore, some data may simply defy hierarchical classification, and can fit into many different taxonomies.

These limitations of hierarchical file systems have been expounded upon by many others [10, 9, 5, 8], and yet, few alternatives seem to gain traction in wide use. Birch is an experimental file system that uses a simple model — directories are queries, files are matches to queries, and a path of directories

*This work was done in fulfillment of the project requirements for the UCSC graduate course CMPS 221, Advanced Operating Systems, professor Scott Brandt, Fall 2006.

forms a conjunction — to push an alternative data organization into the file system.

We don't consider Birch a success in its present form; a combination of the quality of searches possible, and the difficulties in making a highly dynamic system operate when serving highly synchronous I/O primitives make it a far cry from usable. Some of these issues are undoubtedly due to our simple definitions of directories and files, and some are because of the short development time — about two months, part time.

2 Related Work

We have already seen alternative interfaces for computer files through client applications that access certain kinds of computer files in a non-hierarchical way: Apple's iTunes accesses music and video files in ways based on content, and through playlists the user can create and edit. The Finder application supports so-called "smart folders," which use the Spotlight metadata service to display a simple metadata query as a folder. All of these are useful, but are orthogonal to one another — groupings made in one application cannot be used in other applications.

There have been efforts to provide similar, general-purpose user interfaces — a database file system, as an alternative file manager and graphical shell for the GNOME and KDE desktop environments [5]. This database file system solves much of the problem of application-specific searches by augmenting the file dialogs that GNOME and KDE programs use, when a person opens or saves a file in an application. Metadata queries and arbitrary groupings are made global at the UI interaction level. The Haystack project [6] is a loose collection of applications that present new ways of working with semi-structured data.

The semantic file system [4] closely resembles Birch, both in goal and in function. In the semantic file system, special directory names are used to form queries by referencing "transducers," which match files against various criteria.

Tagging, in which users add keyword-style categorizations of files, is extremely popular in recent web sites such as Flickr and delicious, and can form ad-hoc user-generated taxonomies [7]. Tags are put into use in the TagFS system [2].

The Spotlight system built in to Mac OS X, starting with version 10.4, provides a rich language for interacting with various file attributes. Spotlight uses a key-value mechanism, where a metadata key — for example, `kMDItemFSName`, which is the key for the file's name — is mapped to a value for each file. Metadata keys can reflect file system information, such as the file's name or modified time, or they can be *content-based,* giving a richer overview of the file's content; this can include anything from a list of authors, to the bit rate for digital video and audio, to the color space of digital images. Files that don't have a sensible metadata value for a given key have a nil result for that key. Spotlight is the metadata system we use in Birch, and we find it usable, if lacking in quality in some areas.

3 File System Concepts

3.1 Why a Stacking File System

We could implement our alternative file access system purely in a client application, similar to the existing Finder application, and some existing efforts [5, 8] work from this model. We do, however, have an existing wealth of applications that use the standard C library calls `open`, `read`, `write`, etc. Presenting new views of data through a file system allows us to use existing programs to access these data, without modifying the underlying program. This idea is a powerful one, and is not new — the design of the GNU Hurd [3] emphasizes the power of this idea, and new APIs such as FUSE [12] bring this idea to other operating systems.

3.2 Directories as Sets

The key concept in our file system design is that a directory represents a *set:* an unordered collection of unique files, and any particular file may exist in multiple sets. This definition doesn't stray very far from the formal definition of a directory — which is a list of files and subdirectories. The difference between directories and directory-sets is that sets are not necessarily part of a rooted hierarchy, and can instead be composed with one another in arbitrary ways.

Subdirectories, and paths, define a path through the rooted file tree to a specific file, using the character `/` to separate components. Thus, `/Users/csm/file.txt` defines a path through the file system tree that identifies a file `file.txt`. With directory-sets, the `/` character becomes a *conjunction operator,* and so the path of directories becomes an intersection of all named sets in the path. Thus, if we have two named sets `foo` and `bar`, each of which has some number of files, some of which may be common, the path `foo/bar` (or `bar/foo`) contains the intersection of files in `foo` and the files in `bar`. The union of two sets can be gotten by reading the contents of one set, then the next, then removing duplicates; in other words, `ls foo bar | sort | uniq` would produce an accurate union of two sets `foo` and `bar`.

3.3 Directories as Predicates

Managing the file system using sets, whose contents may be completely arbitrary, is a fine notion in itself; what we lack then, is a way to define these sets. We can imagine this being done by hand: copying, or perhaps linking, a file from one set into another places the file in both sets. Limiting ourselves to simple POSIX file system operations makes this kind of management difficult and error-prone. If we were to try to maintain set semantics given the low-level file I/O functions, the system would quickly collapse.

Instead, in Birch, every directory represents a predicate — a simple metadata query, such as "name equals 'foo'," that is, an assertion about a file's attributes that has a boolean result; such a predicate, when tested on a collection of files, defines a set. The intersection of two sets is then defined as the conjunction of the two directory-sets' predicates. There is currently no explicit way to define a disjunction of two or more predicates, but this may be a future extension to the system: wherein each directory-set represents a disjunction of one or more metadata assertions, and path composition forms the conjunction of statements, as above. This is the conjunctive normal form, assuming that each predicate operator has an inverted companion, and thus it is of sufficient power to express any query.

3.4 Naming

In order to reference an existing set when building a path, our system requires that set names are global. This has some obvious drawbacks, since it replaces, in the user's experience, a heirarchical naming burden — the burden of organizing files in a sensible directory heirarchy — into a flat naming burden — that of coming up with a unique name for each query.

This is not, we believe, as bad as it sounds. Since queries are inherently more powerful than directories, and because they are reusable in many places, there should be fewer queries than directories. Query names can be evocative of their function, so the query named "Author is Brian Eno" would be a query that looked for files where the "author" is Brian Eno. It is still up to the person using the system to invent a good name for a query, but naming queries can be simple, if the name is evocative of the function.

4 Implementation

We now describe our file system implementation. This file system is implemented as a Cocoa application that creates a user-space NFS system, and NFS RPC calls (which, naturally, only come over a local socket instead of over the network) are interpreted to render metadata search queries as files and directories. A simple Cocoa user interface (shown in figure 1) manages the search queries the filesystem renders. Queries are composed using the `NSPredicate` and `NSMetadataQuery` classes, which access the Spotlight metadata database.

The basic object is the system is the Query, which contains three fields:

- The *predicate,* which is a Spotlight query stored in an `NSPredicate` object. An example query is `"kMDItemAuthors == '*Brian Eno*'"`.
- A boolean attribute *isLeaf.* "Leaf" queries show their results in the file system view; non-leaf queries do not. The purpose of non-leaf queries is to store a partial query, which can be composed with other queries. Since the partial query may match too many files, the non-leaf attribute prevents large, possibly uninteresting, result sets from being displayed.
- A set of names of other sets, called *subordinates.* Subordinate sets are always shown in the file system, and provide a way to save metadata search

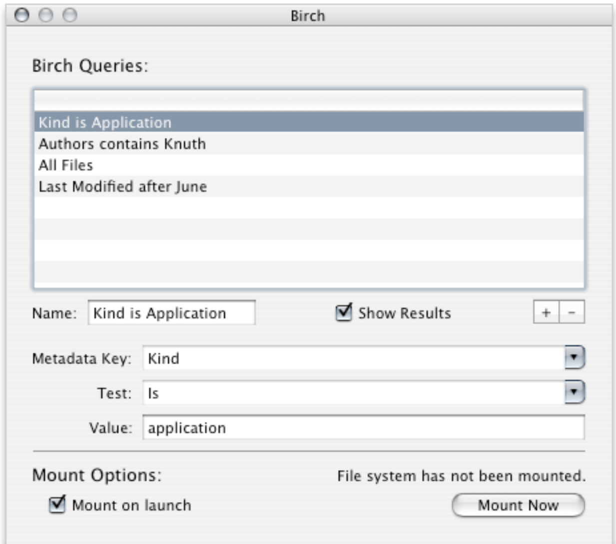

Figure 1: The Birch control interface.

> paths. For example, one could store a query, like `"kMDItemAlbum == '*Music for Airports*'"`, inside a query that it is commonly used with, such as `"kMDItemAuthors == '*Brian Eno*'"`. There is no hierarchy implied here; two queries may contain one another in their subordinate sets, and conjunctions can be formed by specifying either one first.
>
> This provision is a concession that some ways that people access files doesn't allow them to type in (or even remember) the names of queries. For example, we can type the name of any set when using the shell, but in Finder, for example, we get a graphical display of the working directory, and while a user could use the "New Folder" command, giving the new folder the name of the query, having them immediately available is friendlier.

Each query also has a name, which must be unique amongst all other query names. A request for a path, say `/Kind is PDF/Authors contains Knuth/`, where the query named "Kind is PDF" corresponds to the predicate `kMDItemKind == 'PDF'`, and the query named "Authors contains Knuth" corresponds to the predicate `kMDItemAuthors == '*Knuth*'`, the compound predicate for the path would be `kMDItemKind == 'PDF Document' && kMDItemAuthors == '*Knuth*'`. If either query has its *isLeaf* attribute set to true, then the path will display all results that match the conjunction.

In order to simplify the implementation of the NFS server, no I/O requests are handled by the NFS server itself. Instead, files referenced in a query always appear as symbolic links, where the content of the link is the full path to the file elsewhere on the file system. This mechanism greatly simplifies the objects the Birch file system needs to keep track of: everything in the file system is either a directory, and thus a predicate, or it is a link to a file matching a predicate. We think this kind of file system — which only uses links to reference resources — is a very useful implementation strategy, since it simplifies and optimizes file access, once the file is looked up. This makes the file system analogous to other forms of search results, like Internet search engines, which produce as output not the content found, but merely links to the content. It is also much easier to properly implement POSIX semantics if the capabilities of the file system are restricted, which was invaluable given the short development time of this project.

There are four "kinds" of metadata supported by Birch, and each kind supports different tests. The kinds of metadata available for search are [1]:

- **Strings**. Valid tests include eqality, inequality, starts with, ends with, and contains. The operand of the test may be any string.
- **Dates**. Valid tests are equality, inequality, before date, and after date.
- **Numbers**. Valid tests are equality, inequality, less than, and greater than.
- **Arrays of strings.** Valid tests are contains, and does not contain. Containment is, in spotlight queries, equivalent to a substring match in the concatenated list.

The root directory of the file system is a pseudo-query itself, which matches all files, is not a leaf, and contains all other queries as subordinates. Thus, the root directory shows a view of all other queries. Creating a new directory in any directory will do one of two things: if the new directory is named the same as an existing query, then that query is placed into the containing query's subordinate set, and from then on can be accessed to form a conjunction with the containing query; if the name is new, a new query is created in the database, which has a null predicate (i.e., it matches anything), is not a leaf query, and has no subordinates. That query can then be modified through the Birch user interface.

4.1 Lookups and Cacheing

In the NFS protocol, version 2, a "file handle" — an opaque identifier used by NFS clients to identify files that have been looked up — is 32 bytes long. In Birch, a file handle is always composed by using the MD5 [11] checksum of the file's path, with the upper 16 bytes padded with zero. The only exception is the root file handle, which is always 32 zero bytes, and the root path always begins with the string `"/Birch"`. Thus, a virtual path `"/Birch/foo/bar"` will have the file handle

```
2c804fc3 5adb5c62 96f46043 b1a09cec
00000000 00000000 00000000 00000000
```

These file handles are mapped internally to directory entry ("dentry") structures, which track the kind of file, the file's name, its parent dentry, and so on. We can thus build the full path of any dentry by walking up the list of parent dentries, and thus we can reconstruct a file handle if we are only given a parent directory file handle and a file name — we just compute the MD5 hash of the parent file path concatenated with the file name. Thus, the NFSv2 `LOOKUP` command can quite quickly look up a file by name, given its parent directory's file handle, using only two hashtable lookups, one string concatenation, and one MD5 computation.

In order to support certain core OS X applications like the Finder better, Birch does support memory-only read/write files, for things such as the `.DS_Store` file (which Finder uses to store window state information). Birch also always contains an empty, non-modifiable file `.metadata_never_index` in every directory, which prevents Spotlight from indexing the mounted Birch file system.

5 Interpretations and Limitations

5.1 Metadata Quality and Availability

Birch uses the Spotlight database for its metadata searches, mostly out of convenience, since this database exists on, and is constantly updated by, Mac OS X by default. We have found Spotlight to be less than ideal for robust, usable searching, and efforts to improve metadata indexing would benefit Birch, and Spotlight users in general. The problem with Spotlight is that it imports metadata keys (other than the obvious ones, available through the file system for all files) via "tagged" content. Many file formats — AAC or MP3 audio, PDF documents, video formats, etc. — support metadata "tags," or key-value pairs that describe the content. Spotlight uses this information, where it is available, to populate its database — so if an AAC file is tagged with artist "Bob Dylan," Spotlight can populate the `kMDItemAuthors` key with that info. If a file is not tagged, or is tagged incorrectly, then a proper index cannot be built.

One example of Spotlight's limitations is the "authors" key, when applied to PDF files. Ideally, one should be able to find any PDF file on a system — say, if it was in a collection of academic papers — by searching for one of the authors. Many programs that produce PDF output don't tag the document with this information, of if it does, it requires that the creator fill in these fields manually. This is a general problem when doing content-based indexing [?].

5.2 Leaves are more sparse than internal nodes

A set-based file system like we have presented here necessarily, and literally, turns the file system hierarchy on its head: since a leaf node is defined as a series of conjunctions, leaves are always more sparse than nodes higher up in the tree, except in cases where nodes on a path are all maked as "non-leaf."

This can be a usability problem, since a person using the file system would, in general, be presented with more files early on in her browsing, and may be overwhelmed with the number of files present so high up in the tree.

There are potential refinements we can make, to help alleviate this scenario: one could be to make the subdirectory relationship a disjunction, instead of a conjunction, and make each directory predicate a conjuction of assertions. This would make the file system layout more "tree-like," while retaining the power of a full logical language (the disjunctive normal form).

A perhaps better alternative would be to make all query directories non-leaf — that is, make them so they never show their results — and then in each directory present a pseudo-directory called "results," which "leafifies" the parent statement. This way, a person is never overwhelmed by intermediate results: she either is browsing through available queries, or is inspecting the results of a query.

We recognize, too, that these issues stem directly from our insistence on making the file system have POSIX semantics: without that requirement, and the tree structure it implies, a new interface may, in fact, be better. Using POSIX semantics is valuable when working with existing programs, but may pose barriers to usability.

5.3 Getting Lost

Because directory names are global, it is easy to get "lost" while browsing the file system: the person browsing has no indication that the directory he is in is part of a more complicated query or not. That is, it is very hard to tell if a query has a parent query (which, if it does, will produce a conjunction) and thus opening a particular directory may give unexpected results.

5.4 Synchrony

A significant problem with Birch is how it interacts with synchronous I/O operations — in some early versions of the program, metadata queries took so long to produce any results, even if those results returned a nil result set, that the NFS client timed out. Applications like Finder are completely unresponsive while some I/O operations, like `readdir`, are taking place. A complicated query with few results can render some applications unusable for minutes at a time.

We have taken some measures to improve this, such as adding an internal timeout for metadata queries, such that if the query produces no new results within 5 seconds, we end the query with the partial results. This is of only moderate help, of course, because the application will remain unresponsive for up to 5 seconds or longer, if (for example) incremental updates to the result set happens every five seconds.

Another issue is with queries that match many, many files. These can produce directories that contain hundreds of thousands of files, and again, some applications are unresponsive until *all* files have been read out of a directory. A simple workaround that we use here is a hard cap of 10,000 files per directory.

This is not a robust solution, but we see no other way of forcing large queries to finish in a reasonable amount of time.

We are unsure if this limitation is inherent in the POSIX API and — more importantly — applications that use that API, or a limitation in NFS version 2 and the client-side support built into OS X. We don't see any way to force applications to update with partial results, while still polling the directory — many applications simply insist on reading an entire directory before returning any results.

5.5 Forming Conjunctions

As we have mentioned, a directory path forms a conjunction of all predicates in that path. The implementation of this simply forms a new `NSPredicate` with all of the terms joined with the conjunction operator, `&&`. Running complicated queries with many terms is relatively slow; this can, in principle, be optimized by using the results from the partial query and running the additional terms on sucessive partial results.

5.6 File Lookups, Caching, and Consistency

When a client performs a `readdir`, all matched files are inserted into the dentry cache, so when the client comes back to `lookup` each of these, we will have the entries cached. If a file is not in the dentry cache, we are left with the problem of looking up a file, which both matches a query (defined by the containing directory) *and* the requested file name. This is simple enough to define, by forming a conjunction of the query and a term that matches `kMDItemFSName`. This kind of query is very slow, especially if it fails to match any files; our only recourse, to maintain responsivenes, is to only run these queries briefly, returning `ENOENT` if nothing is found.

This caching strategy works well to avoid this issue: after a `readdir`, chances are that any file that will be looked up will exist in the dentry cache. We then, however, have a problem with when to expire dentry cache entries: flushing them early will mean that subsequent `lookup` calls will likely fail. Also, since the dentry cache can be extremely large, removing the least-recently-used entries (the cache is an associative map from file handle to directory entry) is time-consuming.

6 Performance Measurments

In this test we chose to run some "primed" metadata queries — that is, we ran the query once before running the test, to ensure that the metadata server is caching that query – using both `mdfind` (the command-line interface to Spotlight) and `ls -l` on a mounted Birch file system. These tests were run on a MacBook Pro with a 1.83GHz Intel Core Duo, and 1GB of RAM. The system has a single 80GB SATA hard disk about two thirds full with the author's files. The results of these runs are presented in figure 2.

Query	`mdfind`	`ls -l`	**result count**
`kMDItemKind == Application`	0:00.688	0:02.043	216
`kMDItemFSName == *.txt`	0:20.090	0:22.701	1146
`kMDItemFSSize > 1024 && kMDItemFSSize < 1280`	2:12.543	2:48.835	12655
`kMDItemFSSize > 1024 && kMDItemFSSize < 1280 && kMDItemFSName == *.txt`	2:08.499	2:06.944	39
`kMDItemKind == *Image*`	0:01.111	0:30.731	6010
`kMDItemNumberOfPages == 12`	0:00.545	0:00.668	5

Figure 2: Query benchmarks

These results seem to show that the bulk of the time spent in the file system is running the query, and that the NFS protocol overhead (which includes a `lookup`, `getattr`, and `readlink` call for every search result) is low. An excellent refinement to Birch would be to run queries in the background, caching the results until a client arrives to read them. The numbers also indicate that complicated queries that use wildcard patterns, have many terms, or read certain attributes take more resources to run.

7 Conclusion

We have presented a prototype file system that renders search queries as files and directories. There are some technical hurdles left to overcome, especially in the area of performance, and given that many applications wind up reading dynamic, asynchronous results with a highly synchronous API. The Spotlight metadata service is useful in this project because it already exists, and provides a number of interesting fields to search, but is in our opinion too limited

In the course of developing this project, we uncovered a bug in the OS X NFS client support, which resides in the kernel. An early version of Birch was able to cause a kernel panic, simply by returning malformed NFS RPC replies.

References

[1] Apple Computer, Inc. Spotlight metadata attributes reference. `http://developer.apple.com/documentation/Carbon/Reference/MetadataAttributesRef/MetadataAttributesRef.pdf`, April 2006.

[2] Stephan Bloehdorn, Olaf Görlitz, Simon Schenk, and Max Völkel. TagFS — tag semantics for hierarchical file systems. In *Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06)*, 2006.

[3] Michael Bushnell. Towards a new strategy of OS design. `http://www.gnu.org/software/hurd/hurd-paper.html`, January 1994.

[4] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Semantic file systems. In *ACM Symposium on Operating Systems Principles*, 1991.

[5] Onne Gorter. Database file system: An alternative to hierarchy based file systems. Master's thesis, University of Twente, August 2004.

[6] David R. Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A customizable general-purpose information management tool for end users of semistructured data. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*. MIT Computer Science and AI Lab, 2003.

[7] Adam Mathes. Folksonomies: Cooperative classification and communication through shared metadata. `http://adammathes.com/academic/computer-mediated-communication/folksonomies.pdf`, December 2004.

[8] Seth Nickell. A cognitive defense of associative interfaces for object reference. `http://www.gnome.org/~seth/storage/associative-interfaces.pdf`, 2006.

[9] Jakob Nielsen. The death of file systems. `http://www.useit.com/papers/filedeath.html`, February 1996.

[10] James W. O'Toole and David K. Gifford. Names should mean what, not where. In *ACM SIGOPS European Workshop*, 1992.

[11] Ron Rivest. RFC 1321: The MD5 message-digest algorithm. `http://tools.ietf.org/html/rfc1321`, April 1992.

[12] Miklos Szeredi et al. FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`, 2006.