Spring 5-1-2014

A Novel, Tag-Based File-System

Aaron Laursen
Macalester College, aaronlaursen@gmail.com

AARON LAURSEN

STUFFS

STUFFS

AARON LAURSEN

A Novel Tag-Based File-System

Honors Project
MSCS Department
Prof. Libby Shoop, Adviser
Macalester College

Spring 2014

Aaron Laursen: *STUFFS*, A Novel Tag-Based File-System, Honors Project, © Spring 2014

SUPERVISORS:
Prof. Libby Shoop

LOCATION:
St. Paul, MN

TIME FRAME:
Spring 2014

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to my family

ACKNOWLEDGMENTS

The author would like to thank:

HONORS ADVISER: Prof. Libby Shoop

HONORS READER: Josh Aas

HONORS READER: Prof. Shilad Sen

CAPSTONE ADVISER: Prof. Susan Fox

ACHADEMIC ADVISER: Prof. Shilad Sen

without whom this would not have been possible.

CONTENTS

LIST OF FIGURES

Part I

PROLOGUE

In which the astute reader may glimpse things to come, the hurried reader may glean the Spark-Note version of this manuscript, and the disinterested reader may give up and do something else. . .

THE SPARK-NOTE TEASER

For decades, computer use has largely focused on managing and manipulating files– creating and consuming media, browsing the web, software development, and even, with such systems as UNIX and Plan9, direct device access can largely be reduced to locating, creating, reading, and writing files. To facilitate these operations, developers have created a vast assortment of file-systems, each presenting a unique framework underlying nearly everything people do with a computer.

For various reasons, these file-systems have historically represented only incremental improvements and alterations from their predecessors, leaving the basic design and interaction models relatively unchanged. Because of this, most common file-systems share a similar set of weaknesses and limitations, intrinsic to those models.

As an attempt to break with these traditional shortcomings, the author has created *STUFFS*, a **S**emantically-**T**agged **U**nstructured **F**uture **F**ile-**S**ystem. It is intended largely as a research platform for investigating fundamentally new ideas in storing, locating, managing, and otherwise manipulating files, their data, and their associated meta-data. As such, STUFFS does not claim to perfectly solve all of these problems – rather, it serves as a proof-of-concept and testbed for a number of promising new approaches.

Of these new features, users are likely most impacted by STUFFS's titular tag-based structure, which spurns the traditional folder hierarchy in favor of a folksonomy inspired, tag-centric approach to file organization. While this change retains backwards compatibility, and is therefore fully usable as a traditional FS, it has profound impact on potential user interaction. In order to support this high-level transition, STUFFS is implemented using a relational database for storage and tag-resolution, and, as an exciting side effect, it has gained proper transaction support and full ACID compliance.

Part II

BACKGROUND

In which the reader endures the obligatory exposition describing the magical world of our protagonist. It even has several dull pages worth of text describing trees. . .

1

THE PEACEFUL SHIRES OF AUXILIARY STORAGE

Readers knowledgeable of the intricacies of databases, files and file-systems may wish to skip or skim this chapter. It is provided largely as brief introduction to those unfamiliar.

Auxiliary storage devices – those which operate below the computers main memory (RAM) and allow for data retention without power[1]– are essential components of modern computers. However, they are, to understate somewhat, cumbersome to use in their raw form. This difficulty-of-use has led to the development of a number of wrappers and interfaces, which generally fall into two categories: File-Systems and Databases.

1.1 THE TOWER OF THE DATABASES

Databases[2], or, more properly, database management systems, represent the more heavy-weight approach to storage interfacing. Their somewhat Orwellian approach necessitates knowing as much as possible about the data they store – its type (integer, id, text, date, blob, etc.), uniqueness, etc. – and strongly defining all relations between data. Using this knowledge, databases are able to build complex, rigid structures expressing these relationships and massive indexes, provid-

1 Such devices include optical and magnetic disk drives, solid state drives, tape drives, etc. In general, it would seem that the term "auxiliary storage" applies to most anything dubbed a "drive" in colloquial usage (including flash-, thumb-, etc.). This is, of course, largely limited to its use in relation to computers and components thereof; such drives as the "Sunday" variety are usually distinct and unrelated.

ing an express-lane for queries. In order to maintain their positions as omniscient arbiters of storage-access, and to maintain their internal structure, databases typically handle all of the actual disk access and data manipulation internally. They are then free to expose higher-level APIs[3] that provide a number of supported operations which can be combined to achieve the desired effect(Ramakrishnan, 2003).

This complex framework and centralized access point allow databases to provide incredibly helpful functionality. Some features, such as fine-grained concurrency support and optimization, are largely invisible to the user, while simple searching and filtering based on virtually any defined property of an entry are generally essential components of day-to-day interaction. Additionally, many databases provide full ACID (Atomicity, Consistency, Isolation, Durability) transactions which are capable of making certain guarantees regarding the integrity and scope of database operations (see Section 3.1).

1.2 THE VALLEY OF THE FILE-SYSTEMS

While databases present a powerful means of storing and organizing data, it is often desirable, for reasons of simplicity, performance, or feasibility, to interact with the actual data without the intermediate layer and imposed structure of a database. However, raw devices are generally too cumbersome to interact with directly. This is where file-systems come in; they provide a thin veneer[4] of orga-

[2] It should be noted that throughout this section, I will be referring primarily to classical relational databases, especially the various SQL engines. However, this commentary is generally applicable to the vast majority of non-relational and NoSQL databases as well. A full comparison of the various approaches to database design and implementation is beyond the scope of this manuscript, and the interested reader is advised to look elsewhere.

[3] These higher level APIs can easily be as complex as an entire Turing-complete language. Examples include the various dialects of Structured Query Language (usually abbreviated SQL), a general purpose semi-standardized interface language for relational databases, and the Multi-User Multi-Programming System (usually abbreviated MUMPS), a database initially designed for high throughput in a medical environment which includes a full data query and manipulation language.

[4] The lightweight nature of file-systems, should not, however, be interpreted as somehow precluding a lack of functionality. Rather it gives users and programmers the freedom to implement higher-level functionality as they see fit. Indeed, nearly every computer with auxiliary storage uses a file-system, and the vast majority of software which interacts with such storage uses a file-system interface.

nization on top of the raw disk while still largely allowing users and applications to do as they like with the actual contents and composition of the data. The only guaranteed abstractions in a modern file-system are the idea of a file – an opaque, content agnostic bucket for data – and a simple organization scheme – the hierarchical directory tree (Giampaolo, 1999).

1.2.1 *The Secret Life of Files*

The first of these abstractions, the file, has traditionally been thought of as roughly equivalent to its analog polyseme[5]. Like their paper counterparts, digital files have been envisioned as a collection of data somehow bound together (Harper et al., 2013). Its actual content is irrelevant; it could be text, pictures, video, audio, some cryptic squiggles, etc.

Digitally, this content-agnosticism is directly baked into the representation of files. At their lowest levels, these mysterious packages are simple strings of bytes with designated beginnings and ends – no more, no less. There is nothing particularly unique about the form of a given file, and the only thing differentiating one from another is the particular contents.

On the most basic level, these objects (physical and virtual) need (and therefore, with human nature being as it is, can only be guaranteed to) support only a few simple operations[6]:

READ

It must be possible to read a file. By this it is meant that all or part of a file's contents may be copied into some form of more accessible, higher-tier memory.

WRITE

In addition to consuming content with a read, it is generally possible to produce content with a write. This operation sets the contents of a file, or part of the contents of a file to some arbitrary sequence.

5 In some senses then, this makes them more synonyms then polysemes. I wonder if there is a word for those things which are themselves reflections of a single essential archetype as seen through the multifold mirrors of the physical, virtual, and the mental/conceptual... But, I digress...

6 Of the operations described, none are technically guaranteed. Some file-systems are, or can be configured to be, "read-only", "write-only",etc. Which is to say that they support only some subset of the standard opperations. In this section, however, I will be focusing on general use file-systems configured in their most common, read-write-create-delete mode.

CREATE

Reading and Writing are likely the most salient operations one may perform on files, but they are of little consequence if the files do not exist. To rectify this, we must also acknowledge a third function which operates on files, or more accurately files *in potentia*, the create operation. As one might guess from its name, create simply takes a file which does not exist, but could exist, and makes it exist.

REMOVE

As the mirror operation of create, a remove operation takes a file which currently exists, and revokes said existence.

1.2.2 *The Tree of Order*

While simply having file abstraction is a large step towards a coherent system for managing auxiliary storage, it does not provide the organization one might desire for answering the simple question "where is my data?" While we can now refine such a question into "where is my file?", the net effect remains largely unchanged.

Early file-systems largely neglected this question. So called "flat" file-systems, including the original Macintosh File System[7] simply stored all of the files together and let users call them by name. Unfortunately this practice, the digital equivalent of simply paper-clipping *all* documents, photos, etc. together (possibly ordered by title), left something to be desired. Naming files quickly became a headache since each name needed to be unique (the same cannot be said of human names, book titles, etc.), in some cases forcing the user to manually check name availability or risk overwriting current files. This approach also lacked efficiency when dealing with large files.

In the analog world, when given a large number of documents which may be added to, retrieved or modified at any time, people often turn to folders and filing cabinets. Early developers, noting the problems with flat file-systems and seeing a ready made real-world solution implemented the same metaphor in their digital document collections, giving the modern hierarchical file-system.

7 Oddly, even while MFS was in widespread use, Macintosh Finder, the bundled GUI file manager, attempted to give the appearance of a hierarchical system to users. This obviously led to no small amount of confusion.

The hierarchical directory tree has traditionally been and largely remains the *only* way to organize files on a file-system level. At its most basic level, this methodology is nearly identical to its folders-and-cabinets analog. Every directory (folder) may contain some arbitrary number of sub-folders and plain files, beginning with some highest level "root" directory. Finding a given file then is simply a matter of knowing the path to it from root. For example, if a file `child` is inside a directory `parent` which is itself in a directory `grandparent` which resides directly within the root directory, one may find `child` by starting at the root and replaying the steps: `root` $\rightarrow$ `grandparent` $\rightarrow$ `parent` $\rightarrow$ `child`. Of course since early programmers generally valued economy of typing and the use of a standard qwerty keyboard, this is generally expressed as: `/grandparent/parent/child`.

2

THE GROWING UNREST

These early file-systems were highly effective in the environment in which they were conceived, but over time they have become somewhat outdated as their assumptions and abstractions remain rooted in a time and culture which is now long past.

2.1 THE CASE AGAINST SIMPLE FILES

Consider the fundamental idea of a file. Traditionally, it is an opaque chunk of data, but as new technologies have emerged, this simple has become inadequate. Relatively early on, programmers saw the need for meta-data, data about the data, to be added containing such simple things as names and timestamps, and so they amended file-systems to accommodate (Giampaolo, 1999), using this simple idea as the basis for a much more complicated system.

Of course once the floodgates on meta-data had been cracked, they were pushed farther. Beginning with simple access control bits and permissions, and quickly escalating to everything from what type of data the file contains, what software created it, and even logically separate information such as the full lyrics of an mp3. These additions have become increasingly more frequent as companies and individuals attempt to assert or maintain ownership over a particular piece of data, adding copyright information, owner information, and digital rights management software directly into their creations.

The net effect of these additions has been to transform files from simple buckets of bits into something much more complicated. Each container is now annotated with everything from expiration dates and advertisements to instruction manuals and a detailed autobiography.

However, even these highly annotated buckets cannot contain all of the information users are beginning to associate with files. It's not simply a matter of storage[1], but rather a limitation imposed by the essentially static nature of file-systems. Some of these qualities are highly dynamic, such as the number of "Likes" a picture has on Facebook. Some are ambiguous or non-finite, such as the conceptual content of a given picture, or the quality of a Wikipedia article. And still others are simply beyond the realm of computers, such as how a video makes a viewer feel.

Systems such as UNIX, with its "everything is a file" mentality have further complicated the question of "what is a file?". Plan 9 had arguably the most extensive expansion of the file metaphor – literally everything in the system is represented as a file. Users are files. Pieces of hardware are files. All of the graphical widgets in the GUI are files. Even the users themselves are represented by files. This extensive file-ization did away with the idea of files as static, human-controlled entities, and replaced it with files as interfaces, middle-men who facilitated communication with a wide array of content providers– not only static bytes on a disk, but also a vast array of dynamic, and in some cases even living data-sources (Pike, Presotto, Thompson, and Trickey, 1990) (Pike, Presotto, Thompson, Trickey, and Winterbottom, 1992) (Garcia-Molina, 2009) (Korth, 1991).

The third nail in the coffin of the traditional file metaphor is a matter of identity. Traditionally, a file referred to a specific set of bytes stored at a specific location. However, end-users tend to take a complex, dualistic view of files. An individual may refer to downloading a file from the Internet, despite the fact that what they have received would technically be a copy of said file, since the original never moved[2]. Two pictures of the same flower might be called the same file, despite existing in different locations and having different meta-data.

Files may also be referred to as "good" in the case of a particularly high-quality recording of a song ("that's a good FLAC"), or "bad" in the case of a laggy movie ("No, don't watch SomethingTotallyNotPirated.avi, its terrible"). In a technical sense both of these files may be perfect– they contain exactly the bytes that were written to them. In fact it may even be that the "good" audio file is technically imperfect, but it's corruption or truncation occurs in such a way that the user

1 Many modern file-systems utilize extended-attributes and forks to provide essentially unlimited meta-data storage.

2 This is to say nothing of the complexities created by peer-to-peer file transfer, in which "downloading a file" may actually entail downloading copies of small pieces of a number of files from a number of locations and then processing them into a combined form...

does not notice it, or that it otherwise does not, in their eyes diminish the overall quality.

How then can to files which are equally perfect as traditional files (their bytes are correct) be "good" and "bad"? In these cases, it is not the files themselves which are being commented on, but rather what is encoded in their data. Since two files which contain recordings of the same song with different representations (different formats, sampling frequency, etc.) could be ranked differently on the quality scale (one "good", the other "bad") it can not be the concept that the files have encoded which is being referred to, but rather the encoded representation.

At the same time, people actually *do* conflate files with the concept whose representation they encode. Consider, one might download a *song* from iTunes, or they might copy a *picture* of a cat from their camera. In both cases it is actually a digital encoding of a particular representation which is replicated, however, conceptually it is the ideas that are represented by those files which are manipulated[3]. The exact representations of the song or picture could vary (quality, format, etc.) as could the actual corresponding bytes, but the actual actions would be the same.

Humans, it would seem have created a sort of trinity of the file (actual data, representation, and concept). This is not entirely unexpected, and one can observe the same phenomenon with nearly any physical data container: books, (sheet) music, etc. The file abstraction, however, exposes only one face of this being– the opaque-bytes view, thus limiting file-systems and computers to that same restricted way of observing, and manipulating a file. While this restricted view no doubt allows for a more optimized, efficient and possible computer technical solution, it also forces users to translate their conceptual tasks (downloading a *song*, looking at a *picture*, writing a *novel*) into the physical bag-of-bytes model which the computer understands.

3 While files (more specifically, their meta-data) typically carry formatting information, access control bits, etc. this information is logically seperate from their human interpretation. A picture of elvis is a picture of Elvis is the same picture of Elvis regardless of whether it is a jpeg or a png and whether or not a given user has access to it – the *idea* is the same. In the analog world, one could say similar things about a book – *The Hobbit* is *The Hobbit* whether a particular copy is written in English, Esperanto, or Elvish (encoding invariant), and regardless of whether it is in a private collection or public library (access control invariant), etc.

2.2 ARBITRARINESS OF PHYSICAL IDENTIFICATION

Looking back at the file-system structure through the lenses of files-as-concepts and files-as-representations reveals a number of inadequacies. Consider the the hierarchical tree and its idea of a path – the singular and exclusive file identifier. Under the Free Hierarchy Standard, a simple path to a picture of Tim holding a horseshoe crab on his trip to Florida last spring may be found in `/home/tim/Pictures/May2012/Horseshoe_Crabs.jpg`. Under the classic interpretation, this is simply a series of directories contained within each other. However, this is only valid for a physical interpretation of files. After all, how can a concept, an abstract idea, exist within a folder? This physical space interpretation then simply fails to apply (James W. O'Toole and David K. Gifford, 1992).

At this point, attempting to reinterpret this path as a series of meaningful descriptors rather than an arbitrary identifier may help.

HOME
: The "home" directory, as part of the Free Hierarchy Standard (FHS), is largely a pre-defined part of the file-hierarchy. Theoretically, the "home" directory like its brethren "usr", "sys", "etc", "var" and the rest, represents a technical distinction between parts of a file-system. Historically, this directory has often been hosted on a separate disk, or disk partition from the root system. In modern systems, this is still common in enterprise environments, but relatively rare in personal computers.

TIM
: "tim", as a directory, serves to designate private storage for a particular user. In general then, this is merely a way of assigning a piece of meta-data to a set of files.

PICTURES
: "Pictures" is again a piece of meta-data, in this case specifying the file-type.

MAY2012
: The "May2012" directory is effectively a time-stamp, once again a piece of meta-data.

HORSESHOE_CRABS
: "Horseshoe_Crabs" is again meta-data. This time however, it does not represent a standard field, rather it is an arbitrary, user-defined descriptor.

.JPG
: The ".jpg" extension is largely beyond the user's control. In general, this, like all file extensions, is a standardized form expressing the file format.

In general then, each of these pieces of the path is simply a piece of meta-data. In fact, the location itself is simply meta-data (data about the data).

The particular set of meta-data used in the path is not necessarily the entire set of meta-data. Rather it is some arbitrarily selected subset with an arbitrary ordering. Logically then, one may ask "why this data?" or "why this order?". Why not save the same picture as:

- "/home/tim/May2012/Pictures/Horseshoe_Crabs.jpg"
- "/home/tim/Pictures/SpringBreak/Horseshoe_Crabs.jpg"

  or even
- "/images/Florida/2012/Tim's_pictures/awesome/$IMG_0 1234.jpg$"

Each of these options, as well as any permutation of any subset of the entire collection of meta-data, is arguably just as valid so long as it uniquely identifies the data in question. Locating a given file then requires remembering an entire, specific, and largely arbitrary subset of this meta-data, an increasingly more difficult prospect in a world where disk capacities are measured in terabytes and music collections alone may contain tens or hundreds of thousands of files.

If each of these paths is logically valid, then why are these technically invalid? Under traditional file-systems, this single valid path is entirely a consequence of the physical, location-centric structure of the system. Once again, it seems that the traditional view of files is at odds with their new conceptual identity.

Instead of viewing a file as being identified by a vector in physical space, one may instead view them as being a vector in concept-space. Unlike a physical space, this concept space does not, necessarily, have an independent basis – any given "location" can be accessed by some finite, but potentially non-singleton set of discrete and independent vectors. This new space then allows all of the many logically valid paths, rather than specifying unique directions to an end goal, paths simply give enough information to specify the end goal itself.

2.3 UNIQUENESS

A number of readers are probably thinking that, while these points may be conceptually valid, they "have a system" and are perfectly capable of using arbitrary paths and remembering the folder order. This may even be true. However, it is not only the arbitrariness of location-paths which is problematic, but also their *uniqueness*. While having paths map surjectively to files is perfectly reasonable and desirable, their injective mapping can cause chaos for even the best systems.

Consider a stock photographer who likes to be able to easily look at all available photographs containing a particular content – say people, bicycles, and cats. It is then perfectly logical to create a simple hierarchy containing `/photos/{people/, bicycles/, cats/}` with each directory containing the appropriate corresponding pictures. This system works perfectly well until this photographer takes a picture of a person (or a cat for that matter) riding a bicycle. Logically, this picture should be in both `/photos/people` *and* `/photos/bicycles`, but can only exist in one.

The obvious solution, which does not entirely break the content categorization, involves simply creating directories for the overlaps: `/photos/{peopleANDbicycles/, peopleANDcats/, bicyclesANDcats/, peopleANDbicyclesANDcats/}`. However this requires an impractically large number of directories for more than a small number of base categories. Specifically, for n base categories, it may require as many as $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$ directories[4].

One could, of course, contest that this is entirely a function of the categorical organizational system, and that using some known-to-be-discrete descriptor, such as the the date the picture was taken, the first x characters of a hash of the image, the size of the image, etc. While this is true, it is beside the point. The photographer wants to be able to easily find all of his pictures containing cats/people/bicycles/etc. (and is willing to identify them as such on creation). This is not an uncommon or unreasonable request, but it is one which the traditional file-system metaphor is ill-equipped to support or even allow (Seltzer and Murphy, 2009).

4 for reference, on a typical file-system with 4Kb blocks, as few as 28 base categories requires over a terabyte of storage just to hold the empty directories (assuming 4KB directory entries, the default for ext3 and some others)

2.4 THE JOHNNY MNEMONIC EFFECT

This arbitrary unique path concept seems to be overly taxing on human memory – as described, in order to retrieve a given file, a human must perfectly remember an arbitrary piece of text. For a single, relatively short path, this is fairly simple: `/myfile`, `/pics/spot.jpg`, but longer paths and names become significantly more difficult: `/var/abs/core/linux/0001-x86-x32-Correct-invalid-use-of-user-timespec-in-the-.patch` or `/media/net_drive/home/aaron/CloudBkp/files/Macalester/CompSci/Honors/STUFFS/Thesis/Chapters/Chapter03.tex`. To compound this, users may have thousands of files, all of which have unique, and often similar, and therefore potentially confusable, paths. Even a dozen paths easily exceeds the working memory capacity of most users (Miller, 1956). And so like this section's titular character, users memories are being asked to store large amounts of what is essentially *computer* data in a storage medium which is neither optimized for the encoding, nor particularly spacious (at least as far as short-term memory is concerned).

This effect is not limited to local file-systems, and has been addressed in in a number of different areas. As an analog, consider the World Wide Web – it is, at its heart, a distributed file system. URLs (effectively paths) uniquely specify a particular item (a file, which could be html, css, video, etc.) somewhere on the network.

3

THE CRACKS IN THE WALL

The shortcomings of traditional file-systems are not, however, limited to theoretical disconnects with user perceptions of files. The systems themselves exhibit a much more practical issue: fragility. Despite their common use, file-systems have historically[1] suffered from a disturbingly high failure rate – corrupt files, inconsistent state, outright file and directory loss, etc. Unfortunately, a single file-system error can bring entire systems, and by extension entire networks or organizations, to a halt[2]. This is unacceptable. Errors happen – this is undeniable and a function of the human condition – but the capacity for harm of file-system errors rivals even errors in core kernel code[3] (Traiger et al., 1982) (Gray, 1981).

3.1 TRANSACTIONS AND ACID

Databases, the go-to solution for highly reliable data-storage, have achieved their place via the use of *transactions*. Transactions, state changes which follow the ACID – Atomicity, Consistency, Isolation, and Durability – principles, are capable of making some guarantees regarding the limitations of damage possible due to errors. In general, they ensure that, when errors happen, they are detected and do not affect the overall state of the system.

1 Modern file-systems have made significant strides towards the goals outlined in this section, but have not yet reached a consensus on the ideal solution. See chapter 4 for more detail.
2 Computer system and network design has attempted to mitigate this possibility through the use of redundancies and backups, but none are entirely perfect. However, the very need for these sorts of systems is further evidence of a problem.
3 Since a file-system error can induce a kernel error, and vice versa, determining which is more dangerous is non-trivial. On the other hand, a given file-system error is more likely to affect stored data, which, unlike hardware and software, may not be interchangeable or replaceable, causing more permanent damage.

3.1.1 *Atomicity*

The first of the ACID principles, *Atomicity*, describes the boolean nature of a transaction. It must either entirely complete, or none of it may complete. For example, if a transaction renames a file `A` to `B`, then, once it completes, exactly one of either `A`, if the transaction failed, or `B`, if it succeeded should exist with the contents of the original `A`. It should *not* terminate in an intermediate state with either both `A` and `B` or neither. Similarly, after a transaction creates and writes a new file, either the file must exist with its entire contents, or it must not exist at all. In simple terms this principle reduces to wise words of Yoda, 'Do or do not, there is no try'.

3.1.2 *Consistency*

The *Consistency* principle states that a transaction must transition a system from one valid state to another. This effectively means that at no point may a transaction generate an invalid file-system. For example, if a given file-system requires that there exist a valid path from root to every file (under a traditional hierarchical scheme), then transactions are forbidden from violating this by, say, deleting a directory with subdirectories or contained files (since these would be orphaned and therefore invalid). In short, *don't break it*.

3.1.3 *Isolation*

The third principle, *Isolation*, deals with multiple transactions. It requires that a set of transactions generate the same final state regardless of the order in which they apply, even if some number of them occur simultaneously. In file-system terms, this one is somewhat complicated since it is lacking in traditional implementations (see Section 3.2.2). In general this principle means that concurrent file-system access should not interleave their operations. For example, if two transactions append data to a file, the net result should be the original file, and then the appended data from one of the transactions and then the other. This can be summarized as *one transaction at a time*.

3.1.4 *Durability*

The final principle, *Durability*, ensures that, once transactions claim to complete, their results will be permanent. Even if the system crashes, the computer is turned off, or loses power, once, say, a file claims to have finished writing, that data will not be lost (and thanks to atomicity, if it has not yet finished writing, it will revert to its original state). Simply put, *transactions should not lie about their achievements*.

3.2 WHERE FILE-SYSTEMS FAIL

Many traditional file-systems do implement some subset of transactions – wrapping their basic operations into discrete functions. However, they generally fail to follow some or all of the ACID principles. Because of this, they fail to make many of the guarantees necessary for constructing a stable system.

3.2.1 *Update In-place*

Many early file-systems developed for hard-disks fell to what Jim Gray, inventor of the transaction principle, called the 'poison apple' of in-place updates (Gray, 1981). Eschewing the continuous ledger method of bookkeeping in effect since the age of clay tablets, these file systems began the new and innovative practice of *overwriting* data. Unfortunately, since they began writing immediately over the data to be modified without backups, Atomacity is impossible for any transaction which writes to the file-system (data, meta-data, structure etc.). In the case of writing transactions which fail part-way through and is incapable of completing, since there is no way of recovering the original (now over-written) data, the system cannot revert to its original state (Tamma and Venugopalan, 2014).

This also has a number of potential implications regarding consistency, as can any other Atomicity violation. Consider a transaction which manipulates the meta-data around a directory. If, through power failure, electrical surge, or EMP, an invalid configuration of bits is written, this cannot even be detected until the damage has already overwritten the disk. In the worst case, this can leave an entire sub-tree of the file-system unreachable due to a single corruption.

3.2.2 *Isolation? What Isolation?*

While modern file-systems have made some progress, with variable success, in implementing Atomic transactions, Isolation remains largely absent from traditional file-systems. The curious reader is encouraged to attempt to write (or for more fun, append) to a single file from half a dozen programs simultaneously and observe the result.

3.2.3 *Disk Buffers (a.k.a. "We'll save it later")*

Stability is not the sole goal of file-systems. In particular, the field has been and continues to be highly benchmark driven – seeking and celebrating even minute speedups. This has led to a number of risky innovations including disk write-buffers. Since, in general, the actual storage device is the slowest part of a file-system operation, buffers have been implemented to hold data before it is written to disk, spreading it out as needed to prevent the disk from bogging down. They are also used to lump small writes together into a single larger write to avoid repeated spin-up costs and reduce heating and power consumption.

File-systems then report completion once the data has reached the buffer, and possibly before it has actually reached the disk. This, *by definition*, violates the Durability principle. If, for whatever reason, the actual write either never occurs, or fails, the stored data is lost. As an additional consequence, if a read is performed after a write completes, it will return the data stored in the buffer. If, at a later point, the actual write to disk fails, then consecutive reads may return a result inconsistent with the the initial buffer read without any intervening writes (Sweeney, 1993).

4

THE GATHERING STORM

These issues with traditional file-systems are not new. They have been actively documented for decades, and numerous proposed solutions have been developed. Some fixes have involved complete re-designes of the underlying technologies, while still others have taken a much smaller view and addresses some specific symptoms on top of the traditional model.

4.1 BAND-AID FIXES

The latter case, on-top band-aid fixes, are likely the most common solution, thanks to their relatively simple implementation and deployment. Unfortunately, the scope of these solutions tends to be highly limited. Nonetheless, a variety of such solutions exist, and bear consideration.

4.1.1 *Duplication*

A simple method of overcoming the uniqueness restrictions of traditional file-systems involves simply placing copies of files into all of the desired locations. However, this is, for rather obvious reasons infeasible for files which are subject to change, moved or be deleted due to the need for synchronization. Files of any significant size are similarly prohibative due to space constraints. As such, it is not really a solution except in some niche cases.

4.1.2 *Links: Hard and Symbolic*

The earliest, generally effective, effort at solving the structural restrictiveness problem came in the form of hard and symbolic (sometimes soft, also sym-) links. In both cases, links allow a file to exist at multiple paths at one time. As such a movie about cowboys fighting aliens could exist in both `/movies/cowboys/` and `/movies/aliens/`, thus removing, or at least ameliorating the one-file-one-path restriction allowing for complex, non-standard systems such as that employed by GoboLinux (Muhammad, Detsch, and Leopoldo-RS-Brasil, 2002) (Homer, 2014). Hard links accomplish this by having multiple directory entries refer to the same actual inode and data storage, while soft links use one actual file with data and creates additional pointers to it at other locations as needed.

Unfortunately, while links are highly effective in a number of situations, they are not an effective solution to the general case of problems. Since they lack automatic management, they must be individually created, deleted, etc. This requires a significant amount of additional work on the part of users since they must manually recall and manipulate every valid path individually. This is especially important, and potentially problematic when using soft links since deleting or moving the linked file without updating every link leads to *broken links*, an inconsistent state in which links have no valid target.

Even though links avoid the unique path problem, they fail to remove the arbitrariness and ordered nature of paths. One could certainly construct a system under which files (or links to them) exist under all possible orderings and subsets of their path components, but this is somewhat overly-complicated in practice. This would require $\displaystyle\sum_{i=0}^{n} i!$ links for each file, where n is the maximum depth of the file. The number of hard links to a given inode must be stored in a filesystem dependent format. The ten bit counter used by NTFS overflows at a depth of seven, 32 bit UNIX systems overflow at a depth of thirteen, and 64 bit UNIX systems overflow at a depth of twenty-one, making this approach impossible for systems using hard links with more than a few descriptors. Soft links, on the other hand, do not need to be counted, but require a full inode for every link which can require something on the order of four kilobytes each, leading to over two terabytes of additional storage for a file depth of twelve.

4.1.3 *Single Purpose Databases*

The go-to solution for implementing highly stable storage and persistent, searchable data-stores, single purpose databases, are perhaps the most popular means for overcoming file-system limitations. Generally, these databases are housed atop traditional file-systems and are accessed by one, or, in some cases, a small number of highly integrated applications.

These sorts of systems are most visible to the average user via applications such as media players and mangers, including *iTunes*, *Windows Media Player*, and *Amarok*, which use them as a sort of domain-specific file management scheme[1]. These systems generally allow users to interact with a tabular interface, which directly maps to the database back-end, and exposes the native, indexed, column searching functionality at which databases excel. This has shown remarkable benefits over direct file-system interaction in regard to file-location and management. So much so, that systems such as Jody Foo's *DocPlayer* (Jody Foo, 2003) have been constructed in an attempt to extend this metaphor to general file-management.

Single purpose databases are also commonly employed for their stable ACID-compliant transactions. In this case, these databases are constructed to contain some amount of mission-critical data atop a traditional file-system (Subramanian et al., 2010).

Unfortunately, this method is highly limited. Because of the many possible database interfaces and configurations, data storage and application become tightly linked – each application must setup and maintain its own database. Naturally then, this requires any applications which wish to share a database to agree on a model and make sure that they do not interfere with each other. In practice, this is a relatively rare occurrence – in general, arbitrary programs, even those with similar purposes and functionality, cannot be assumed, or even expected to be capable of using the database of another program. As such, any particular piece of data in one of these systems becomes locked-in to a small, closed set of tools rather than being globally accessible.

While this limitation may be reasonable for storing internal data, systems such as DocPlayer which aim to solve the general file management problem run into

1 It should be noted that these systems typically take a hybrid approach. The files themselves are typically stored directly in the file-system, and the meta-data is put into a database. This allows the searchability of a database across the file meta-data (author, year, title, etc.) while preserving the efficiency of file-systems for actual data access.

serious issues. For example, while they can provide useful interfaces for finding a file within their application, those interfaces are unavailable to other applications so, if a user wanted to, say, open an image file from within their favorite graphics editor, they would be required to either use traditional methods or (assuming the manager uses a hybrid system and the files can be accessed directly within the underlying file-system via an exposed path) open the database-backed manager, find the file and copy its file-system path, and then input said path into the graphics editor. In effect, the file manager becomes a sort of card-catalog which must be consulted separately whenever a file is to be accessed rather than an integrated part of a streamlined process.

4.1.3.1 *Desktop Search*

A special case of the single-purpose database, the Desktop Search model, as implemented by such applications as Mac OSX's *Spotlight*, *Beagle*, and *Google Desktop*, swaps the tabular media-player like interface for a more general-purpose, natural language search engine. This makes it especially simple to locate arbitrary files, but limits the system's organizational and file-management capabilities. While this method creates a helpful, general-purpose file launcher, it suffers from most of the same limitations as other single-purpose database systems: limited scope, lack of standardized access, etc.

4.1.4 *Journaling*

Apart from implementing databases on top of traditional file-systems, effort has been made to graft additional capabilities onto existing systems. One of the most common innovations in the area of file-system stability comes in the form of *journaling*. File-systems which implement journaling store pending operations in a circular log. *Physical logging* pre-records every write to the file-system. Because this log can then be replayed in the event of a crash (and an incomplete journal can be detected via checksums), writes can be performed Atomically. Unfortunately, things like broken links can still lead to inconsistent file-systems, and the lack of Isolation is not addressed (Giampaolo, 1999).

Since physical logging effectively requires writing twice as much data for every operation, it incurs significant performance penalties, which has led many file-systems to adopt logical logging. Under these systems, only meta-data is pre-recorded and the actual data is written directly. Unfortunately, since data is not recoverable, corruption can occur if a crash occurs during a write operation. This adds some degree of increased stability, but is far from a complete solution (Sweeney, 1993).

4.2 FUNDAMENTAL REDESIGNS

Striving to avoid the limitations of simple band-aid fixes, a number of more fundamental changes have been proposed and developed. These span the gambit from new paradigms to a complete overhaul of the computer interaction.

4.2.1 *Remove Local General-Purpose File-Systems*

Perhaps the most extreme redesigners have proposed removing the local general-purpose file-system, or, for practicality reasons, keeping it but restricting it to use by essential operating system services and hiding it from the end-user. This then begs the question, *how does a user store data?* This is, after all, the purpose of file-systems and an incredibly common practice among computer users.

The most popular solution involves using cloud storage or similar technologies to replace or augment traditional file-systems (Zhang et al., 2014) (Cachin, Keidar, and Shraer, 2009) (Baron and Schneider, 2010) (Vokorokos et al., 2013). This does not, however, address the problems with the file-system metaphor. If the cloud service is accessed via some private API, then it suffers from all of the same issues with limited application support, etc. that plague single-purpose databases. If, on the other hand, it uses a file-system interface, then it, by design is vulnerable to the same issues as traditional file-systems.

Regardless of the method, removing the file-system has serious repercussions. Any changes to the storage interface potentially removes compatibility with current programs, requiring massive porting efforts, hampering adoption and leading to the same sorts of incompatibility as single-purpose databases.

4.2.2 *File-System Redesigns*

Since abandoning the file-system concept seems to create as many problems as it solves, it seems reasonable to keep the general concept but rebuild it from the ground up. In doing so, engineers are free to explore new techniques and avoid the pitfalls of their predecessors. The most promising approaches synthesize not only the lessons in stability learned from databases, but also new approaches in organization and information retrieval.

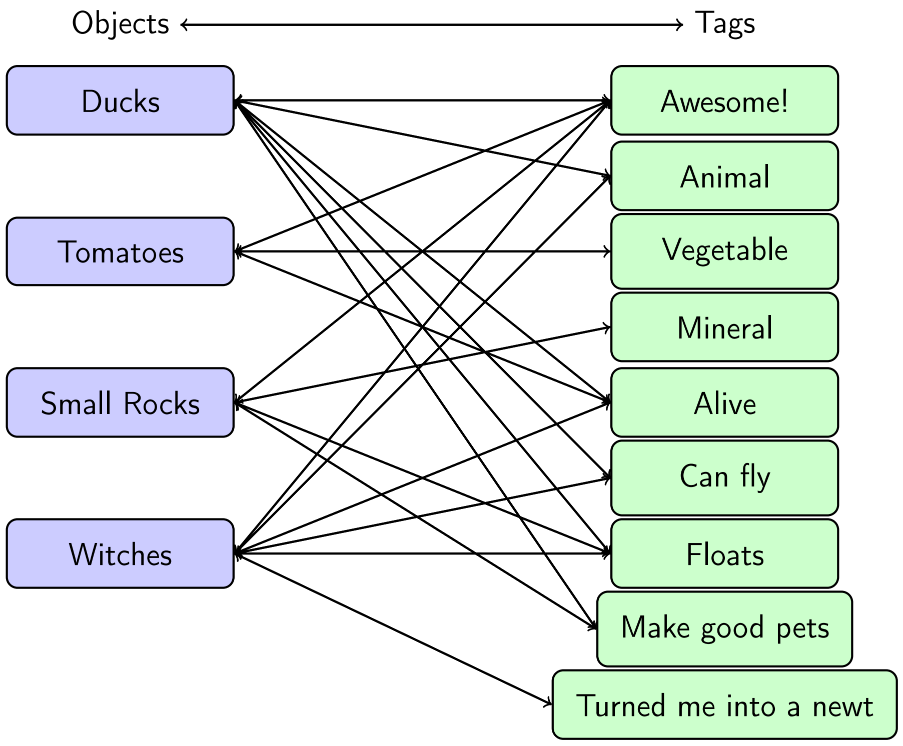
4.2.2.1 *Tagging*

Tagging represents a natural translation of location-paths to concept-paths. This common web 2.0 practice locates and identifies objects (files, web-pages,etc.) using a set of textual descriptors (tags). Every item in in object-space is reversibly, but not necessarily uniquely, mapped (usually by users) to a set of tags in tag-space as shown in Figure 4.1. These objects are then locatable and (not necessarily uniquely) identifiable via arbitrary set-queries over their associated tags. Depending on the tagging architecture, these queries can be as complicated as full boolean expressions which filter the corpus based on applied tags (some examples are given in Figure 4.1).

This approach largely implements the concept-space idea of files discussed in Chapter 2, with tags expressing concepts and being intrinsically unordered sets. various research systems have investigated the possibility of using tags to organize files, but have, for various reasons either failed to be developed, or been incapable of replacing traditional systems. (Hans Reiser, 2001) (Gifford et al., 1991) (Padioleau, Sigonneau, and Olivier Ridoux, 2006) (Dourish et al., 1999) (Seltzer and Murphy, 2009) (Jesse Phillips, 2006) (Schandl and Haslhofer, 2009)

4.2.2.2 *ACID*

A number of file-systems redesigns (You et al., 2013) have also focused on stability, generally by incorporating fully ACID compliant transactions within the file-system itself. One of the most successful research systems in the area, the Amino system (Wright et al., 2007) uses a sophisticated system-call interception scheme to allow applications to initiate and commit transactions within the file-system. This largely surpasses the capabilities of journalling systems and provides a provably stable base.



Some sample queries:

Tags	Objects
Animal	Ducks, Witches
Mineral	Small Rocks
Can fly AND Make good pets	Ducks
Floats AND NOT Animal	Small Rocks
...	...

Figure 4.1: A simple tagging illustration. Elements in object-space are associated with an arbitrary number of elements in tag-space. Objects can then be located via queries on their tags.

Part III

PROJECT STUFFS

In which the hero learns of its legacy and embarks on a daring quest. . .

5 ENTER STUFFS: A FEATURE OVERVIEW

While each of the alternatives discussed in Chapter 4 offer relief from some of the symptoms of traditional file-system maladies, none seems to fully address the causes. To this end, I propose, and have developed, a new experiment in file-system design, a **S**emantically-**T**agged **U**nstructured **F**uture **F**ile-**S**ystem. This project explores novel methods of improving both file-system integrity and usability by taking a radically different approach to the implementation of both the underlying system, and the overlying interface of the file-system metaphor.

5.1 BACKWARDS COMPATIBILITY

One of the biggest advantages of file-systems over other storage solutions is their simple and implementation-independent interface. This allows a wide variety of applications, including other storage solutions (data-bases, etc.) to efficiently utilize a single data-store.

In order to maintain this important property, STUFFS must maintain backwards compatibility with traditional systems. This is largely limited to the interface and API aspects of the file-system, since these are the only ones necessarily shared by traditional file-systems anyway. For specifics, see Chapter 6.

5.2 TAGGING

Inspired by the common web 2.0 practice and similar systems at other levels of the storage hierarchy, STUFFS implements a tag-based organizational system. Each file can be assigned an arbitrary number of textual tags upon creation, and this set may be modified at any later time. Files are *not* assigned a traditional location-based path, but one may be emulated through clever use of tags (see Chapter 6).

5.3 PATHS AS QUERIES

Having lost their meaning as locations, the standard file path has been re-purposed to serve as a search system for tags (For syntax specifics, please see Chapter 6). Each path encodes a tag query, complete with standard AND-OR-NOT combinational logic, and its contents are dynamically generated by the result of such a search. If the path is used to refer to a file, then the first file to match the query is returned. If, however, the path is used in reference to a directory, then it is dynamically generated with contents based on the files whose tags match the query.

As such, a path encoding "Tag1 AND (Tag2 OR Tag3)" would refer to a directory containing all files tagged with "Tag1" and at least one of either "Tag2" or "Tag3". On the other hand, a path encoding "file1.ext with Tag1 AND (Tag2 OR Tag3)" will point to a *file* named "file1.ext" that has been tagged with "Tag1" and at least one of either "Tag2" or "Tag3". If their are multiple files to which this path could refer, STUFFS will arbitrarily pick one (generally the first one created).

5.4 SEMANTIC RESOLUTION

Unfortunately, while tags are a major step towards implementing a concept-space view of files and file-systems, they remain, like location based paths, simply an arbitrary set of textual descriptors. While each tag represents a concept, the mapping is not necessarily injective. Consider the tags `pics`, `Pictures`, `images`, and `IconS`; each of these is a textually different tag, and therefore resolves to a distinct set of files under a tagging scheme. However, these tags each share a synonymous semantic identity, and therefore express a singular concept. Therefore, under a truly concept-centric system, all of these should map to the *same* set of files.

In order to facilitate this, more fully concept-centric, approach, STUFFS implements a Semantic Resolution system. This optional feature uses a semantic similarity metric to collapse semantically identical (or very similar) tags into a single file-set accessible via any synonymous tag (i.e. a file tagged as `picture` could be picked up by a query for files tagged as `images`). Since the query tag does not even need to exist in the file-system, users may search by whatever term seems most appropiate at search time rather than at creation time. This frees them from needing to remember a specific, arbitrary piece of *text* describing their file, and instead generate any text expressing a *concept* which describes their file.

5.5 UNIQUE GLOBAL FILE IDENTIFICATION

This tagging system fully replaces one of the central components of the traditional hierarchy – the unique path. Since any number of files may share some set of tags and have the same name, a given path may resolve to zero, one, or arbitrarily many distinct files. In the case of browsing, this is desirable – each of the files located by a given path is valid. Adding more terms to the path can logically differentiate the files, and therefore systematically differentiates the files.

On the other hand, simply identifying a known file is significantly more complicated using only tags. Even once a file has been identified via a unique set of tags, this identification cannot, in general be guaranteed to be unique and valid in the future. Adding tags to other files may induce conflicts, and modifying the tags of the file in question may remove it from its original path. Unfortunately, this violates the first rule of STUFFS by failing to be backwards compatible. Everything from loading kernel modules to setting desktop wallpapers relies on being able to specify a unique path to a specific file, and without this capability, STUFFS fails to be backwards compatible.

To rectify this, STUFFS introduces the concept of *file IDs*. Each file, on creation, is given a globally unique file ID which tracks that specific file for its entire lifetime despite any file manipulations (read/write, tag additions/deletions, renameing, etc.). This file can then be uniquely identified using only its ID. STUFFS supports this natively using the standard path interface (see Section 6.4.1.1 for specifics).

These IDs are not merely a replacement for traditional paths. They also add useful functionality. While conceptually similar to inode numbers, because these IDs are exposed via the typical file-system interface, they can provide this location-independent identification to general purpose applications. Now, when a user specifies desktop wallpaper it can remain selected, even when that user reorganizes their pictures, removes and adds tags, or even renames the file.

5.6 ACID TRANSACTIONS

As an experiment in developing a stable file-system, STUFFS implements fully ACID compliant transactions. By default, every syscall is wrapped within a transaction as implemented by a SQLite database. This provides all of the standard transaction guarantees implemented natively at the file-system level. By automatically wrapping standard syscalls, STUFFS provides all of the benefits of a transaction system to legacy applications without requiring software modifications.

6 SYNTAX AND OPERATIONS

As stated, STUFFS has been designed for maximum backwards-compatibility while implementing as many advanced features as possible. To that end, all operations use the standard path format and require no special libraries or mysterious system calls. This allows all features to be used by any program using standard, current tools.

6.1 THE TAG

Like most simple tagging systems, STUFFS employs purely textual tags. These textual tags apply to an arbitrary number of files and serve as a sort of analog to the traditional directory.

6.2 THE FILE

STUFFS new format for files converts them from the traditional, opaque series of bytes to more complicated objects:

File

TAGS
: The set of all tags applied to the file – for convenience, this includes the empty tag for all files.

NAME
: The user-assigned file name – equivalent to the traditional file name.

ID
: A globally unique file identifier.

DATA

The opaque byte data – identical to the traditional file data.

This new file metaphor does not impose the traditional constraint on file names being unique. Instead, each is assigned a unique ID which is appended to the display name as "$name@id@$". Each file can then be uniquely specified by its id: "$@id@$", a combination: "$name@id@$", or, in the case where the rest of the path removes ambiguity, only the name: "$name$".

6.3 THE PATH

Since the standard hierarchical file-system has been replaced by a tagging system, a path as a series of directories is meaningless. Instead, this form has been repurposed not as a traditional vector in physical-space, but rather in tag-vector space. This redefinition converts a physical map marking the location of a file into a query specifying its attributes. In short, the path has now been redefined from an ordered sequence of directories to a logical combination of tag-based filters.

Given the file mapping in fig. 6.1, one such path may be: "/Floats/%Make good pets%Alive%/!Animal/?Stone?/" which points to a directory containing the file "Small Rocks". In order to understand this mapping, one must first understand STUFFS's basic query components: intersection, union, negation, and fuzzy matching.

6.3.1 *Intersection*

In the simplest query case, we have intersection, typically defined for sets as:

$$S_1 \cap S_2 \equiv \{s \mid s \in S_1 \wedge s \in S_2\} \tag{6.1}$$

Extending this to the new file/tag object metaphor gives:

$$Tag_1 \cap Tag_2 \equiv \{file \mid Tag_1, Tag_2 \in file.tags\} \tag{6.2}$$

Unfortunately, the "$\cap$" syntax is is cumbersome to type and is potentially incompatible with some localizations. For convenience, STUFFS uses the "/" symbol:

$$Tag_1 \cap Tag_2 \Leftrightarrow Tag_1/Tag_2 \tag{6.3}$$

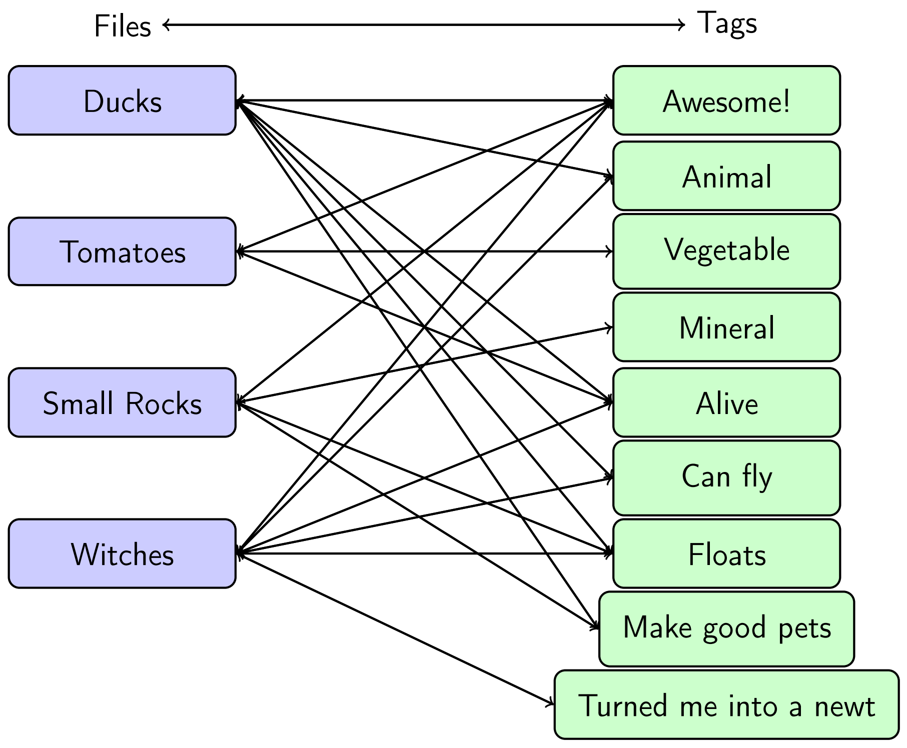


Figure 6.1: Object mapping from fig. 4.1. Note that each object is a *file*.

This convention has the added advantage of providing compatibility with traditional paths: "$/\mathtt{dir}_1/\mathtt{dir}_2/\mathtt{dir}_3/$" now maps to "$\text{␣} \cap \mathtt{dir}_1 \cap \mathtt{dir}_2 \cap \mathtt{dir}_3 \cap \text{␣}$", where "␣" is the empty tag. Since "␣" applies to every file, this is degenerate to "$\mathtt{dir}_1 \cap \mathtt{dir}_2 \cap \mathtt{dir}_3$". This means that a file can be made to exist in the "path" "$/\mathtt{dir}_1/\mathtt{dir}_2/\mathtt{dir}_3/$" by tagging it as "$\mathtt{dir}_1$","$\mathtt{dir}_2$", and "$\mathtt{dir}_3$". In this way, arbitrary classical paths can be formed in the expected way.

6.3.2 *Union*

Union, in many senses the opposite of Intersection, is defined for sets as:

$$S_1 \cup S_2 \equiv \{s \mid s \in S_1 \vee s \in S_2\} \tag{6.4}$$

with an analogous definition for tags:

$$\mathtt{Tag}_1 \cup \mathtt{Tag}_2 \equiv \{\mathtt{file} \mid \mathtt{Tag}_1 \in \mathtt{file.tags} \vee \mathtt{Tag}_2 \in \mathtt{file.tags}\} \tag{6.5}$$

Once again, "$\cup$" is difficult or impossible to type in many situations and so the following modified form has been adopted:

$$\mathtt{Tag}_1 \cup \mathtt{Tag}_2 \Leftrightarrow \%\mathtt{Tag}_1\%\mathtt{Tag}_2\% \tag{6.6}$$

Admittedly, this syntax is perhaps more complex than is strictly necessary. Unfortunately, a simple infix notation would potentially be more dangerous, since this would effectively prohibit the infix symbol from being used in tags entirely, since it would always be interpreted as a union. The three symbol notation is still limiting, but only complicates the simultaneous use of "%" as both the first and last symbol of a tag. It can still become complicated when using tags containing "%" in unions. However, this is largely unavoidable regardless of notation.

It should be noted that this operation is not restricted to two arguments, but generalizes to higher numbers as:

$$\mathtt{Tag}_1 \cup \mathtt{Tag}_2 \cup \cdots \cup \mathtt{Tag}_n \Leftrightarrow \%\mathtt{Tag}_1\%\mathtt{Tag}_2\%\ldots\%\mathtt{Tag}_n\%$$

6.3.3 *Negation*

Negation, the final standard logical operator implemented within the STUFFS path system is traditionally defined for sets as:

$$\neg S \equiv \{u \mid u \notin S\} \tag{6.7}$$

For files this gives:

$$\neg Tag \equiv \{f \mid Tag \notin f.tags\} \tag{6.8}$$

Again, mathematicians have chosen difficult to type symbols, and so the standard programming convention "!" has been used:

$$\neg Tag \Leftrightarrow !Tag \tag{6.9}$$

6.3.4 *Fuzzy Matching*

Beyond the logical operations, another syntax has been introduced to handle fuzzy matching. For performance and specificity reasons (a user may actually mean exactly what they type), fuzzy matching is disabled by default. Enabling it is as simple as surrounding the text to be matched with "?". So, "?pics?" will evaluate to "pictures", "images", "icons", etc.

6.4 OPERATIONS

Now that these basic path manipulations have been identified, we may move on to more complex operations.

6.4.1 *Locating a file*

STUFFS's flexible path-as-tag-filters simplifies the process of locating a particular file. It also provides a number of approaches based on known names, tags, and ids:

6.4.1.1 *By ID*

If the unique id of a file is known, then locating it is trivial. It will be found at "/ALLFILES/@id@". If the name is also known, it can be used as well:

"/ALLFILES/$name$@id@". It should be noted that, in the current implementation, if the last element of the path is an id, the rest of the path is ignored, so

- "/ALLFILES/@id@"
- "/ALLFILES/$name$@id@"
- "/@id@"
- "/some/!other/%combination%of%path%elements%/@id@"

are all the same file.

6.4.1.2 *By tags and name*

A files tags (or some knowledge of a subset of them) can also be used to narrow the file-system scope and identify the file.

Consider a file named "target" with tags "a","b","c", and "d".

If there is exactly one file named target in the file-system, then the the file can be uniquely identified by name alone, "/target" although the preferred method (which avoids potential conflicts with tags named "target", etc.) is "/ALLFILES/target".

If however, there exists some other file named "target" with tags "d","e","f" and "g". Then enough tags must be given to differentiate the two. This could be accomplished by such paths as:

- "/d/b/c/a/target"
- "/a/target"
- "/%a%b%c%/target"
- "/!f/target"
- etc.

However, filters which do not uniquely specify a "target" will lead to ambiguous results and should be avoided:

- "/target"
- "/ALLFILES/target"

- "/d/target"
- "/!h/target"
- "/%a%b%e%f%/target"
- etc.

6.4.1.3 *Browsing*

Browsing through files without a known goal is also possible.

The virtual-tag "ALLFILES" contains all files in the system, and therefore these can be viewed with standard tools for reading directory contents. Additionally, all tags exist as virtual-directories under the file-system root, and entering them performs an intersection by default.

In all other cases for the current virtual directory, a query is generated based on the evaluation of the current path filters. Any files which exist in the logical set returned by such an evaluation exist within the directory. Additionally, all tags applied to files in the directory which are not already used in a path intersection exist as virtual directories within the current directory. This is somewhat more intuitive to grasp in a graphical than textual form and a less murky explanation may be given by Figure 6.2

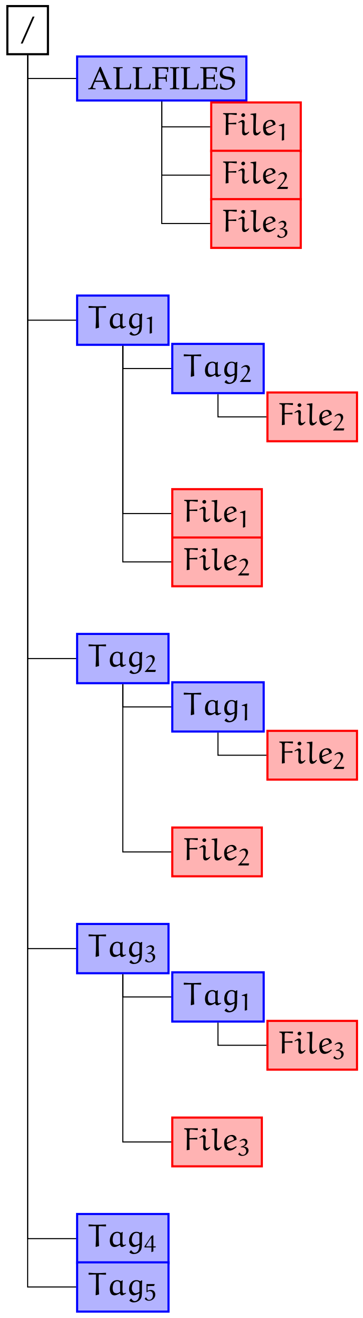
6.4.2 *Manipulating files*

In general, manipulating files in STUFFs is the same as in a traditional system. Once a file is located (see Section 6.4.1) ther location "path" can be used directly by standard tools (`cat`, `touch`, `rm`, `cp`, etc.) without problems.

6.4.3 *Manipulating tags*

Tag manipulation is somewhat more complicated than file manipulation due to its need to re-purpose standard tools. In general, a given traditional function (Create, Delete, etc.) performs its tag analog.

Figure 6.2: Simple STUFFS file-system tree. In this example, the system contains the tags $\{\mathtt{Tag}_n \mid n \in [5]\}$ and the files $\{\mathtt{File}_n \mid n \in [3]\}$. tag subsets $\{\mathtt{Tag}_1\}$, $\{\mathtt{Tag}_1, \mathtt{Tag}_2\}$, and $\{\mathtt{Tag}_1, \mathtt{Tag}_3\}$ apply to $\mathtt{File}_1$, $\mathtt{File}_2$ and $\mathtt{File}_3$ respectively.



6.4.3.1 *Creating tags*

Tags are created by the same means as traditional directory creation. They will not, however, show up in the current directory (unless the current directory is the root directory) (see Section 6.4.1.3). This is not a problem for the file-system, but some interfaces, file managers and the like, may take issue with this. Creating all tags in the root directory explicitly (i.e. '`mkdir /`*`some_tag`*') avoids this issue without changing functionality.

6.4.3.2 *Deleting tags*

Just as tags can be created by standard directory creation tools, they can be deleted by standard directory deletion tools. While there is no explicit preference for either absolute or relative paths, it may be better to use absolute simply for symmetry with the create commands.

6.4.3.3 *Adding tags*

Since files no longer have a hierarchical location, commands which move or rename a file have been re-purposed to support tagging. Moving a file to a given absolute path adds all elements of that path to the file's tag set (so '`mv file@1@ /a/b/c/`' will add tags "a", "b", and "c" to the tags of "file", referenced here by id 1, assuming that those tags exist and are not already in "file" 's tags.).

It should be noted that tagging a file with a non-purely-intersection path is currently undefined and unsupported. This means that commands such as '`mv file /%a%b%/`' will not work.

6.4.3.4 *Removing tags*

Removing the tag "t" from a file is logically the same as moving that file to the set ¬t, and this is reflected by the syntax. Removing a tag from a file is equivalent to tagging the file as the negative of the tag (so `mv file@1@ /!a/` will remove the tag "a" from "file").

Since there is conceptually and practically little difference between removing a tag from a file and adding one, support has been provided for mixing the two operations. A command such as `mv file@1@ /!a/b/!c/d` will add the tags "b" and "d" to "file" and remove the tags "a" and "c" from it. In the case of a conflict

(an operation attempts to remove and add the same tag) the removal takes precedence.

7

SCENERIO DEMONSTRATION

Here follow the exploits of three potential (and purely fictional) users of STUFFS. They illustrate not only common usage patterns, but also some innovative repurposing and extensions of the tagging framework to suit their needs and desires. The three are approximately ordered by technical prowess and complexity of usage.

7.1 BRAD: THE QUESTING

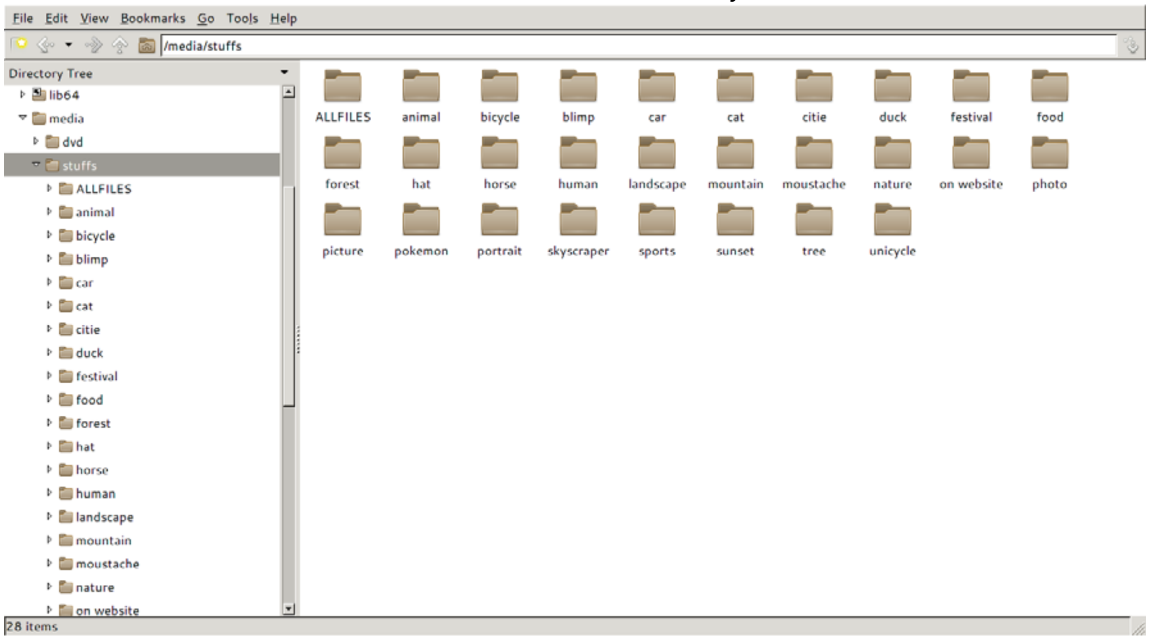
Brad is on a quest again. Fortunately, he has not been tasked with killing ten rats or destroying a great evil. Rather, he is on a quest for the perfect picture. As the freelance stock photographer from Section 2.3, he strives every day to provide his clients with the perfect picture for all of their needs. Along the way, he has picked up some technical skills and now hosts his own website and handles all of his business via his trusty computer. Unfortunately, his need for efficient organization has led him to sample dozens of different schemes, both in as file-system layouts and single-purpose databases, but, for now, he has settled on STUFFS, which seems to meet all of his requirements.

The Tale of Brad

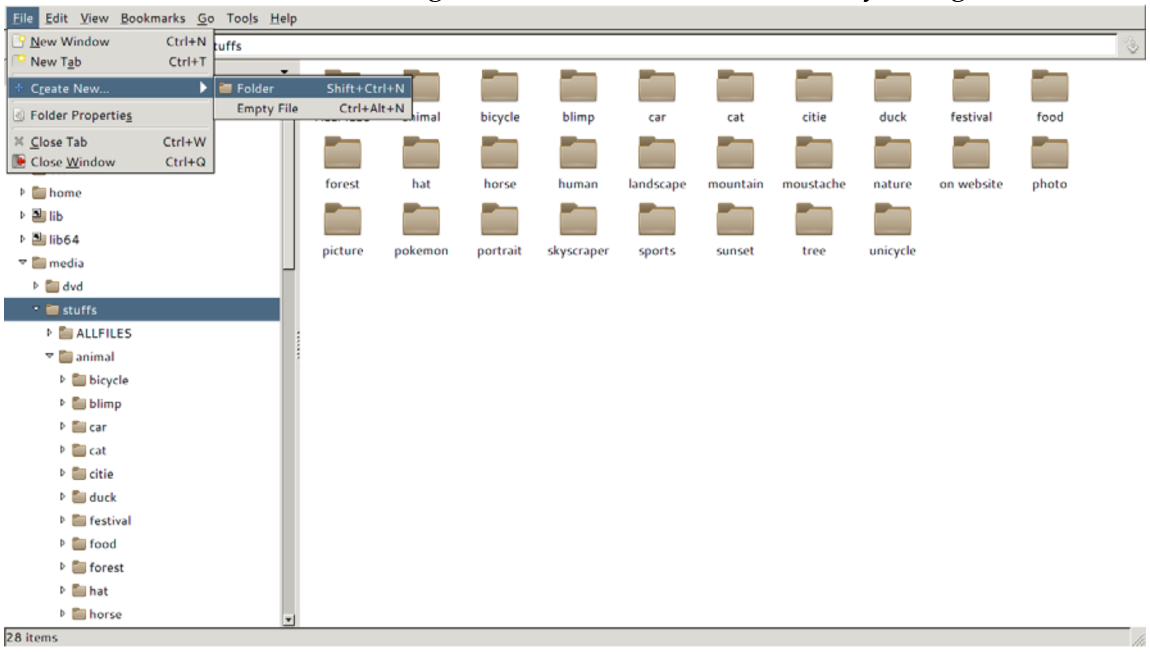
Jim is chatting with a client who needs some pictures. Fortunately, he has a large collection[1]. So, he brings up his file manager (PCManFM `http://http://wiki.`

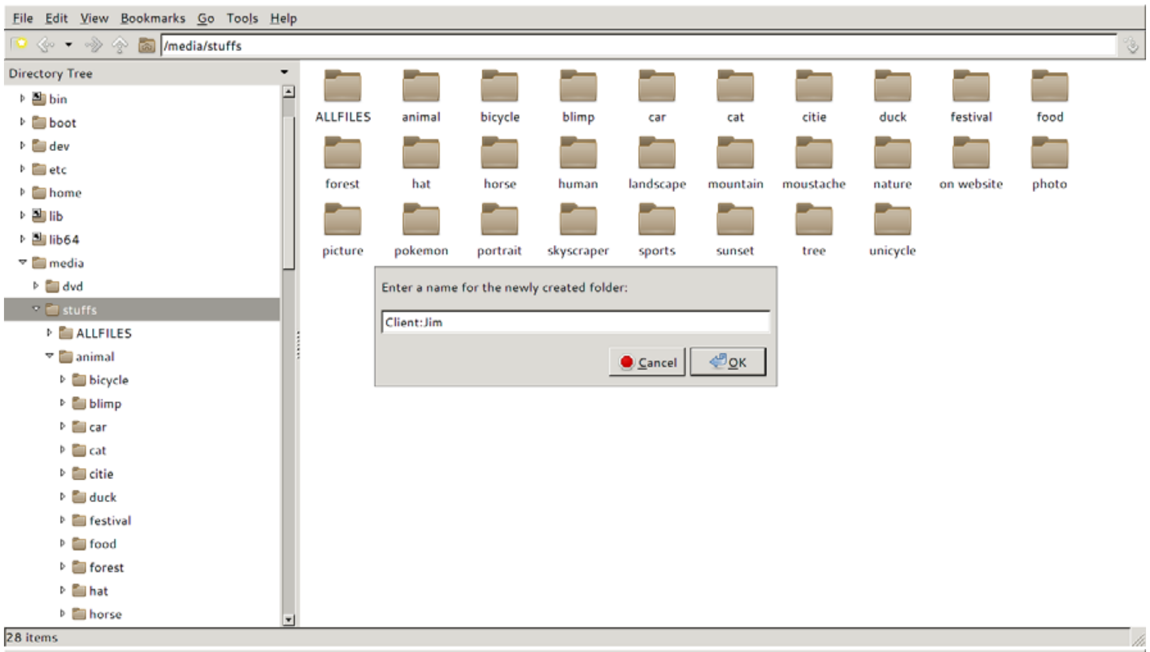
1 It is best to simply assume it is large. The screenshots etc. presented here actually show only a small number of files (~ 200 randomly generated) and tags (~27). This both reduces the visual clutter for readers (somewhat) and the work necessitated on the part of the author.

lxde.org/en/PCManFM) and takes a look (note the path shown near the top of the window. `/media/stuffs` is the root of a STUFFS file-system).



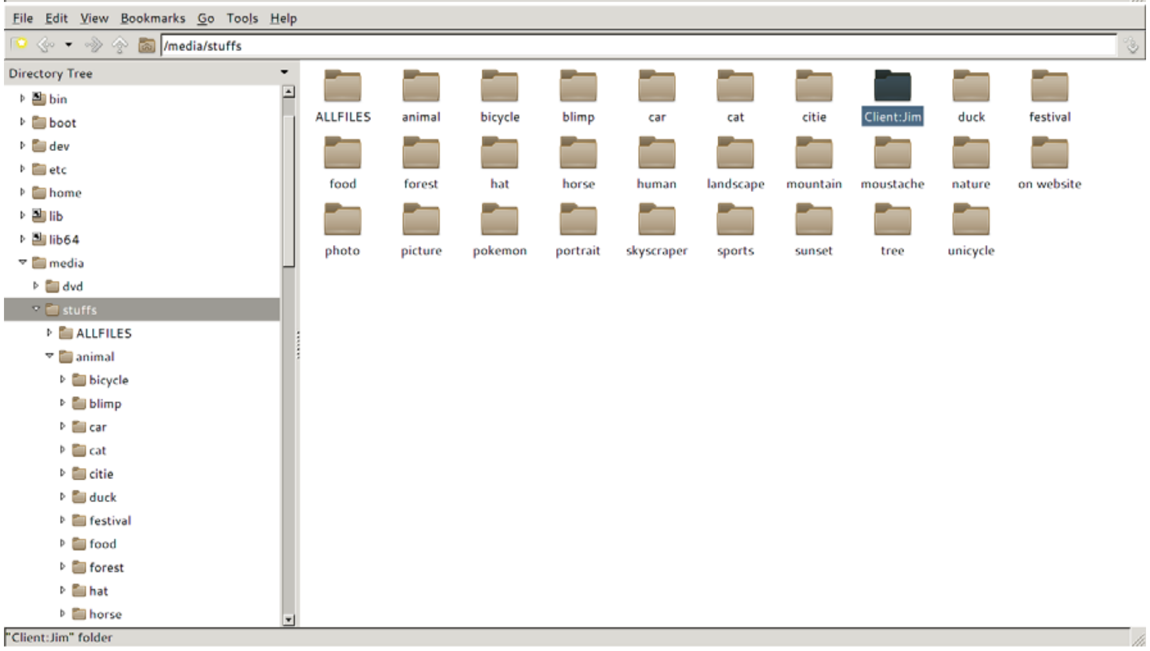
First off, Brad creates a new tag for his client, Jim. This is fairly straightforward:





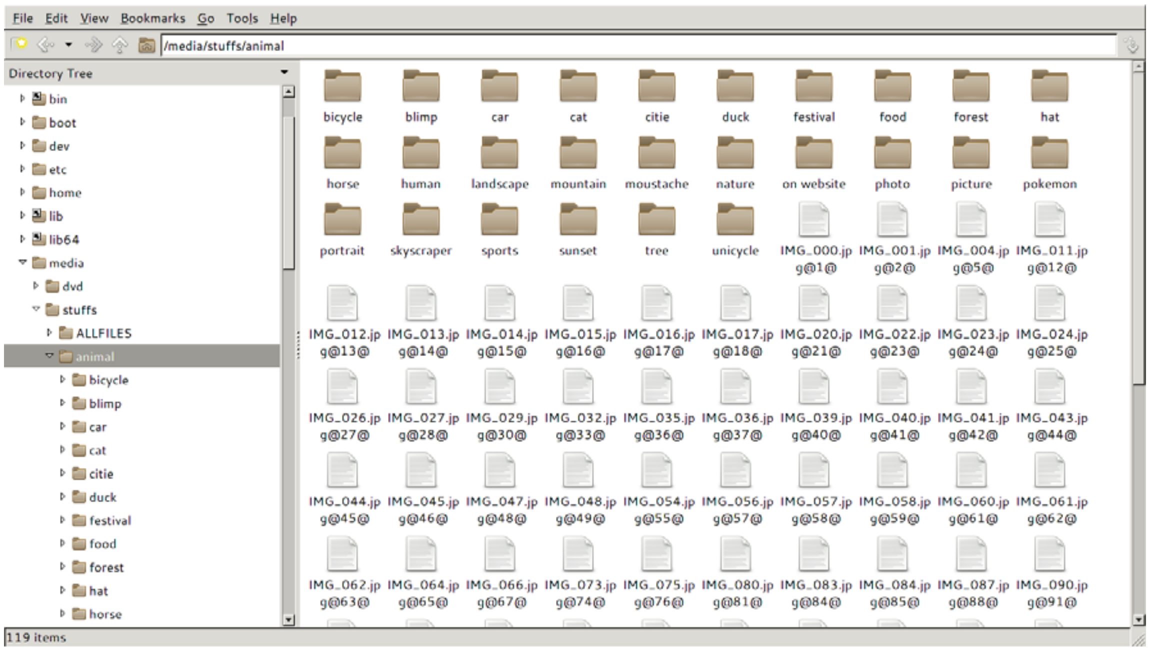
Enter a name for the newly created folder:

Client:Jim



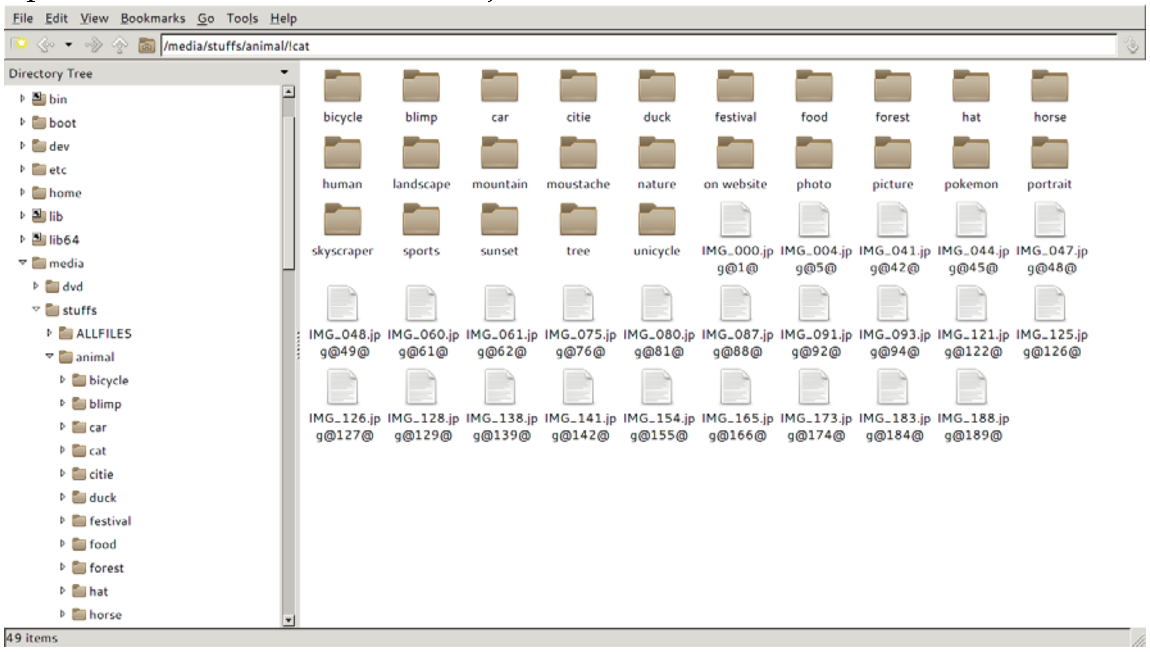
"Client:Jim" folder

Having done so, Brad begins looking for pictures. Jim wants animals, so Brad starts there:



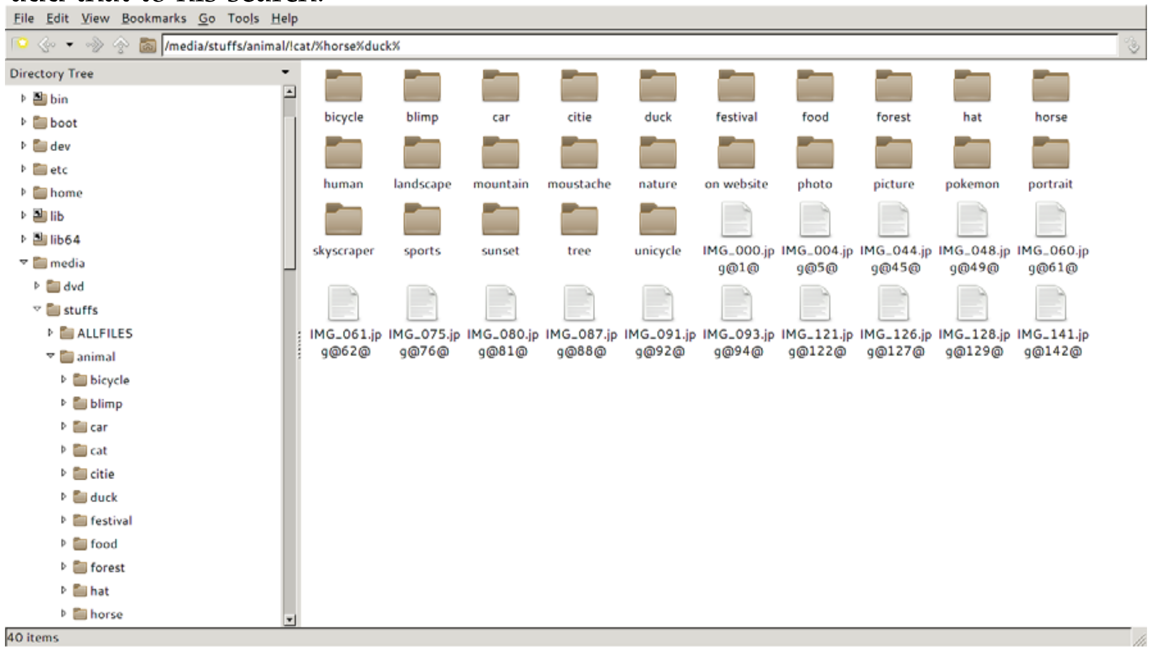
```
bicycle  blimp  car  cat  citie  duck  festival  food  forest  hat
horse  human  landscape  mountain  moustache  nature  on website  photo  picture  pokemon
portrait  skyscraper  sports  sunset  tree  unicycle  IMG_000.jpg@1@  IMG_001.jpg@2@  IMG_004.jpg@5@  IMG_011.jpg@12@
IMG_012.jpg@13@  IMG_013.jpg@14@  IMG_014.jpg@15@  IMG_015.jpg@16@  IMG_016.jpg@17@  IMG_017.jpg@18@  IMG_020.jpg@21@  IMG_022.jpg@23@  IMG_023.jpg@24@  IMG_024.jpg@25@
IMG_026.jpg@27@  IMG_027.jpg@28@  IMG_029.jpg@30@  IMG_032.jpg@33@  IMG_035.jpg@36@  IMG_036.jpg@37@  IMG_039.jpg@40@  IMG_040.jpg@41@  IMG_041.jpg@42@  IMG_043.jpg@44@
IMG_044.jpg@45@  IMG_045.jpg@46@  IMG_047.jpg@48@  IMG_048.jpg@49@  IMG_054.jpg@55@  IMG_056.jpg@57@  IMG_057.jpg@58@  IMG_058.jpg@59@  IMG_060.jpg@61@  IMG_061.jpg@62@
IMG_062.jpg@63@  IMG_064.jpg@65@  IMG_066.jpg@67@  IMG_073.jpg@74@  IMG_075.jpg@76@  IMG_080.jpg@81@  IMG_083.jpg@84@  IMG_084.jpg@85@  IMG_087.jpg@88@  IMG_090.jpg@91@
```

Unfortunately, many of these are pictures of cats (Brad loves cats) and Jim expressly requested pictures *without* cats. The customer is always right, so, through the power of STUFFS, Brad looks at just the animals that are not cats:

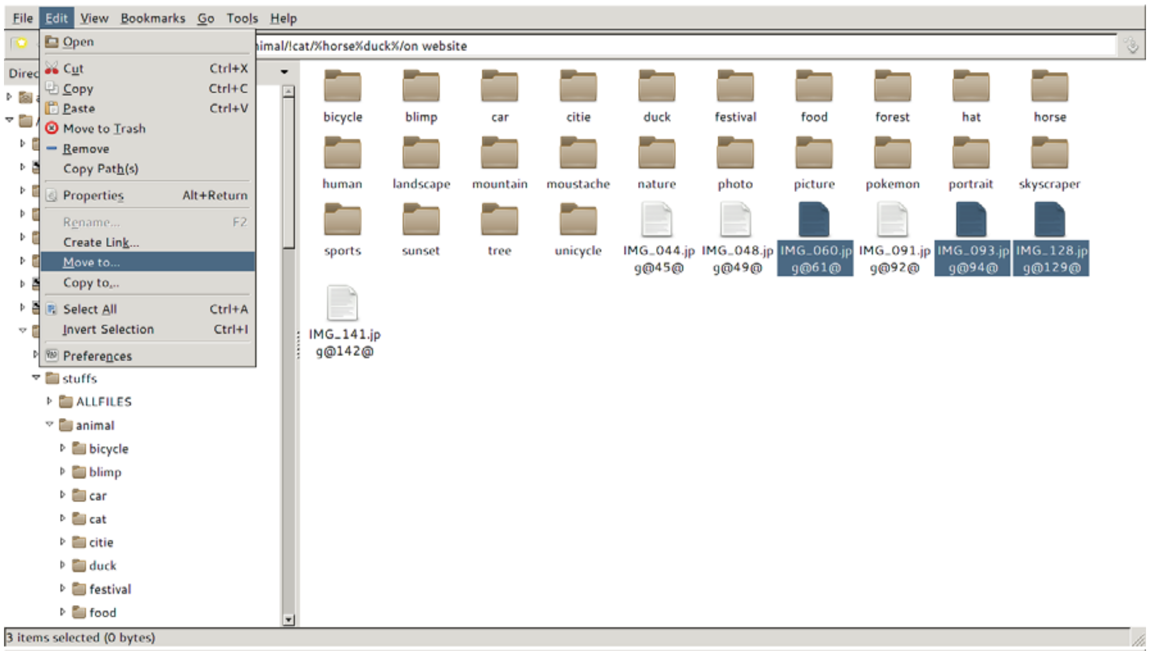


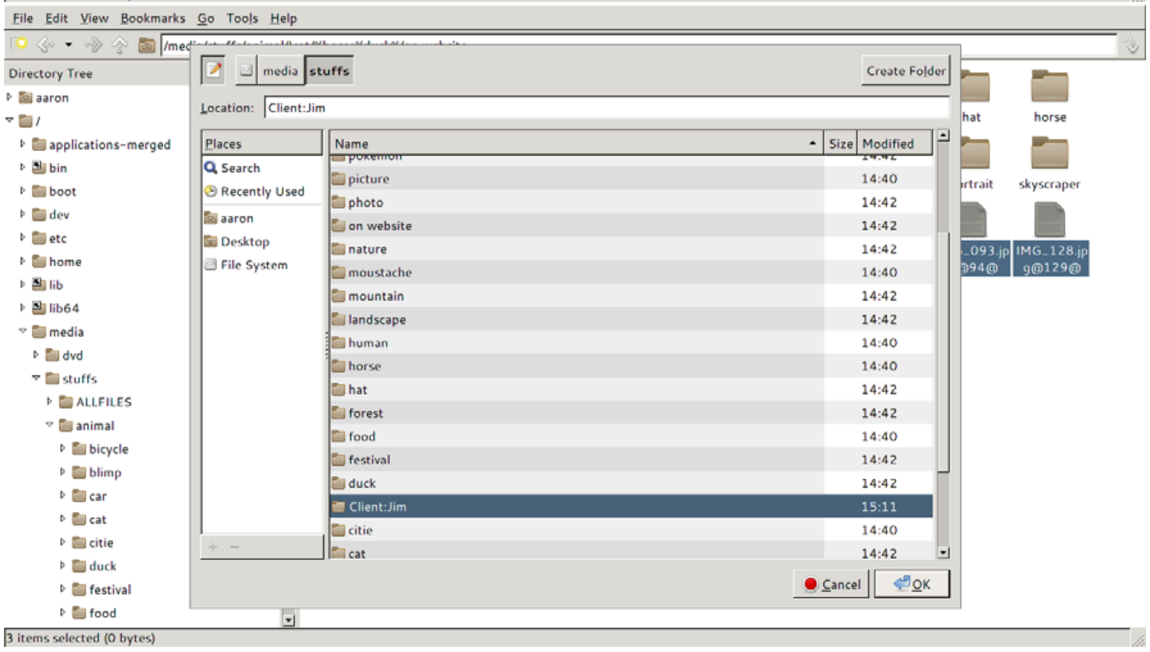
```
bicycle  blimp  car  citie  duck  festival  food  forest  hat  horse
human  landscape  mountain  moustache  nature  on website  photo  picture  pokemon  portrait
skyscraper  sports  sunset  tree  unicycle  IMG_000.jpg@1@  IMG_004.jpg@5@  IMG_041.jpg@42@  IMG_044.jpg@45@  IMG_047.jpg@48@
IMG_048.jpg@49@  IMG_060.jpg@61@  IMG_061.jpg@62@  IMG_075.jpg@76@  IMG_080.jpg@81@  IMG_087.jpg@88@  IMG_091.jpg@92@  IMG_093.jpg@94@  IMG_121.jpg@122@  IMG_125.jpg@126@
IMG_126.jpg@127@  IMG_128.jpg@129@  IMG_138.jpg@139@  IMG_141.jpg@142@  IMG_154.jpg@155@  IMG_165.jpg@166@  IMG_173.jpg@174@  IMG_183.jpg@184@  IMG_188.jpg@189@
```

There are still quite a few pictures, so Brad asks Jim if he has any other requests. "Ducks" says Jim, "or horses. Either one would be nice." Brad is only too happy to add that to his search:

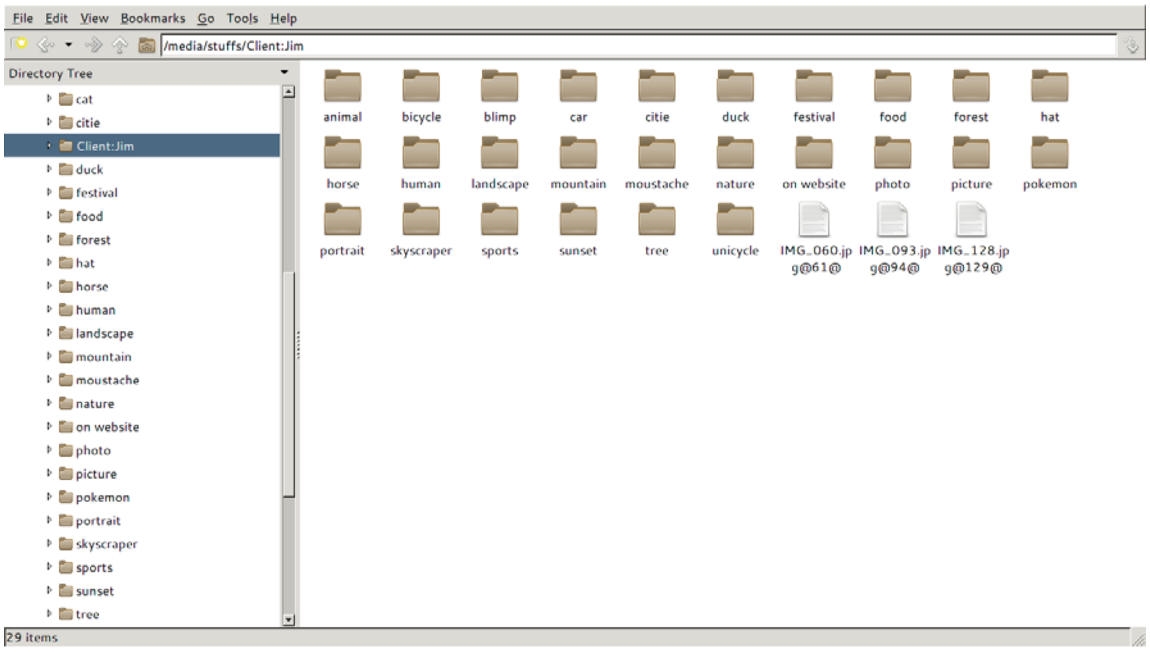


Brad shows these remaining pictures to his client and identifies a few should work. He adds Jim's tag to them so that he can find them later:

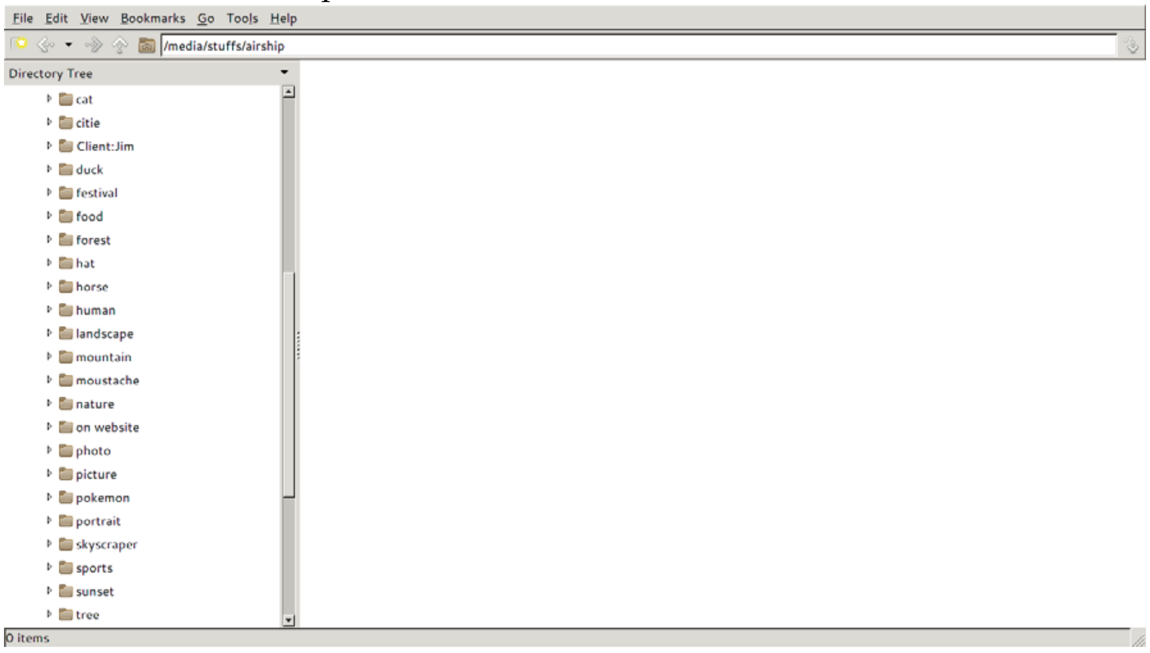




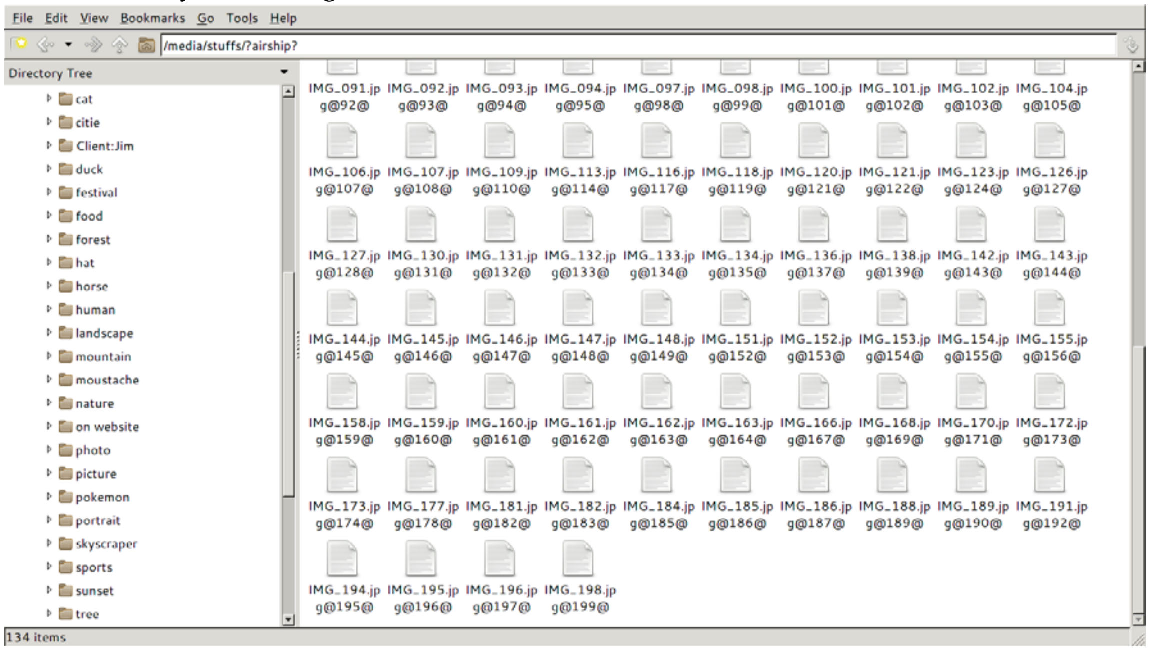
Name	Size	Modified
pokemon		14:42
picture		14:40
photo		14:42
on website		14:42
nature		14:42
moustache		14:40
mountain		14:42
landscape		14:42
human		14:40
horse		14:40
hat		14:42
forest		14:42
food		14:40
festival		14:42
duck		14:42
Client:Jim		15:11
citie		14:40
cat		14:42



Great! Brad has found some animal pictures for Jim. Jim wants more though – this time he wants airships. Brad looks to see what he has:



As the astute observer would expect, there is nothing there. However, Brad knows he has pictures of airships, or something like them. Thinking that he may have simply tagged them as something else ("blimp" in this case) and uses STUFFS's fuzzy matching feature:



```
/media/stuffs/?airship?

cat
citie
Client:Jim
duck
festival
food
forest
hat
horse
human
landscape
mountain
moustache
nature
on website
photo
picture
pokemon
portrait
skyscraper
sports
sunset
tree

IMG_091.jpg@92@  IMG_092.jpg@93@  IMG_093.jpg@94@  IMG_094.jpg@95@  IMG_097.jpg@98@  IMG_098.jpg@99@  IMG_100.jpg@101@ IMG_101.jpg@102@ IMG_102.jpg@103@ IMG_104.jpg@105@
IMG_106.jpg@107@ IMG_107.jpg@108@ IMG_109.jpg@110@ IMG_113.jpg@114@ IMG_116.jpg@117@ IMG_118.jpg@119@ IMG_120.jpg@121@ IMG_121.jpg@122@ IMG_123.jpg@124@ IMG_126.jpg@127@
IMG_127.jpg@128@ IMG_130.jpg@131@ IMG_131.jpg@132@ IMG_132.jpg@133@ IMG_133.jpg@134@ IMG_134.jpg@135@ IMG_136.jpg@137@ IMG_138.jpg@139@ IMG_142.jpg@143@ IMG_143.jpg@144@
IMG_144.jpg@145@ IMG_145.jpg@146@ IMG_146.jpg@147@ IMG_147.jpg@148@ IMG_148.jpg@149@ IMG_151.jpg@152@ IMG_152.jpg@153@ IMG_153.jpg@154@ IMG_154.jpg@155@ IMG_155.jpg@156@
IMG_158.jpg@159@ IMG_159.jpg@160@ IMG_160.jpg@161@ IMG_161.jpg@162@ IMG_162.jpg@163@ IMG_163.jpg@164@ IMG_166.jpg@167@ IMG_168.jpg@169@ IMG_170.jpg@171@ IMG_172.jpg@173@
IMG_173.jpg@174@ IMG_177.jpg@178@ IMG_181.jpg@182@ IMG_182.jpg@183@ IMG_184.jpg@185@ IMG_185.jpg@186@ IMG_186.jpg@187@ IMG_188.jpg@189@ IMG_189.jpg@190@ IMG_191.jpg@192@
IMG_194.jpg@195@ IMG_195.jpg@196@ IMG_196.jpg@197@ IMG_198.jpg@199@

134 items
```

There they are! The process continue in this manner for a while until Jim is satasfied.

7.2 clara: the great and powerful

Clara is powerful – at least in front of a computer. A sys-admin by day, Clara spend much of her free time tinkering with her private computers or listening to her extensive music collection. While perusing a forum, Clara stumbled across STUFFS and, being a bit of a neo- and technophile, she immediately tried it out. She was intrigued. Quickly seeing the value of STUFFS' new tagging metaphor, she imagined and implemented dozens of new uses for it.[2]

2 It should be noted that many of Clara's, seemingly magical, uses of the STUFFS system rely on an incredibly intricate file-system structure, and large number of complex tags. This level of sophistication is likely not practical for the average user, but illustrates some fairly simple, yet powerful capabilities of the system in the hands of an advanced user.

The Tale of Clara

Clara is driving home from work and stops at a red light. Through an open window, she hears a snippet of music – *. . . My mind rides and slides as my circuits are fried. . .* – from the car parked in the lane to her right, before the driver spots an opening in traffic and speeds away. Clara has heard this line of ska before and recognizes its particularly '90s sound, but she can't seem to place it. The remainder of the drive is uneventful, but that line sticks in her head until she reaches her home computer, determined to place it. Clara is certain that she has the full song somewhere, but, without knowing the title, artist, album etc., it would be nearly impossible to find in a traditional, hierarchical, music library. Fortunately, she has STUFFS. . .

Having, shortly after acquiring STUFFS, created a small program that STUFFS-isizes her music collection by extracting meta-data from the files and matching on-line meta-data and lyrics databases and generating corresponding tags, Clara pulls up a terminal (her preferred file management instrument) and locates the song:

```
> ls -p /Genre:Ska/Lyrics_Line:My_mind_rides_and_slide
s_as_my_circuits_are_fried/Year:199*/ | grep -v /
No_Doubt-Trapped_In_A_Box.ogg@98712@
No_Doubt-Trapped_In_A_Box (LIVE).flac@5420@
No_Doubt-Trapped_In_A_Box (VIDEO).ogg@312712@
```

Of course, No Doubt's "Trapped in a Box"! It seams quite obvious to Clara now. It is also quite obvious to her that she would like to hear the rest of the song, so she decides to try it out on a new audio player, maybe XMMS2...[3]

3 Note the use of "–prefix=/InstalledBy:xmms2". This simple option makes package management (and removal) simple by associating all installed files with a known tag. The end result is not unlike GNU Stow, but without all the symlink mess. The addition of "/Requires:ALSA" tags or similar to indicate dependencies could also be used to remove them in a similar operation or prevent unintended dependency deletion. This is somewhat beyond the scope of this project, but should be trivial to implement.

```
> mkdir xmms2temp && cd xmms2temp
> git clone 'git://git.xmms2.org/xmms2/xmms2- devel.git'
> cd xmms2-devel
> ./waf configure -prefix=/InstalledBy:xmms2/
> ./waf build
> sudo ./waf install
> nyxmms2 add /ALLFILES/Box.ogg@98712@
> nyxmms2 play
```

After a few verses, Clara remembers how much she loves No Doubt, Ska, and music from the 90s in general and adds all of it to the list...

```
> nyxmms2 add /Music/%Artist:No_Doubt%Genre:Ska%/*
> nyxmms2 add /Music/Year:199*/*
```

After a while, Clara decides that, while XMMS2 works fine, she really prefers mpd. So, she removes the newcomer in one of the most efficient ways possible, and then goes about her evening as usual.
[4]

```
> sudo rm -r InstalledBy:xmms2/
```

4 Note the "-r" flag which is needed to recursively remove the tagged files as well as the tag itself.

8

METHODS & IMPLEMENTATION

Unlike many current attempts to fix the perceived issues with current file-systems, STUFFS is composed of standard, readily available components held together by a relatively small amount of glue code. While this somewhat limits the functionality that can be implemented, it brings a slew of benefits. On one hand, it allows for a shorter development cycle with lower implementation cost, while on the other, it allows STUFFS to take advantage of a large body of work already performed and adopt new features and functionality from its components with minimal additional work.

8.1 PROGRAMMING LANGUAGE: PYTHON

STUFFS is primarily implemented in the Python programming language (version 3). This high-level, interpreted language allows for rapid prototyping with a short development cycle and cross-platform implementation. Additionally, Python bindings exist for a wide variety of external libraries, including all of the libraries used in the STUFFS system, allowing for simple integration and clean code. While a language such as C would be needed for integration with the mainline kernel and optimal performance, Python is a viable option for the proof-of-concept stage.

8.2 STORAGE BACKING: SQLITE

At its most fundamental level, even with all of its changes in file-system metaphors, STUFFS *is* a file-system. Therefore, the ability to actually store files is essential. While STUFFS ultimately uses a database-backed solution, all three of the options

0 Source code for STUFFS may be found in Appendix A.1

discussed in Chapter 1,raw device, database, and file-system backends, were initially considered before selecting an SQLite database as storage-backing.

8.2.1 *Raw storage*

In the lowest-level case, raw storage can be accessed directly, and, for obvious reasons, direct access is the most common option for general file-systems. However, as discussed in 1, this is highly cumbersome, and, since STUFFS was neither intended to do anything particularly innovative with its disk-access and allocation nor optimized for performance, there is nothing necessitating the level of control achieved by raw access.

8.2.2 *File-Systems*

Using an existing file-system as back-end storage may at first seem counter-intuitive, but a large number of semantic file-systems do just that (Sauermann et al., 2006) (Faubel and Kuschel, 2008). This storage file-system is typically mounted in such a way as to be hidden from the user, but available programmatically. The semantic system can then present an interface which transparently reads and writes files to the background file-system through standard application level calls.

These sorts of systems, when implemented properly, can be reasonably fast, and generally have fairly good space efficiency. Unfortunately, they also inherit the fragility of the underlying file-system implementation making the file-system-on-file-system method impractical for implementing a stable, ACID compliant system[1].

8.2.3 *Databases*

Using a database as storage for STUFFS presents an interesting third option. As with file-system backing, databases avoid much of the minutiae required by low-level raw storage manipulation. Unlike typical file-systems, however, a large

1 While one certainly *can* implement proper transactions and full ACID compliance on top of a file-system (most databases either by default, or optionally, do exactly that), it is a non-trivial process. In doing so, one would essentially construct a database, which seems excessive given their general size and complexity, and the fact that a wide variety already exist.

number of databases provide fully ACID-compliant transactions. This provides a known stable base, allowing for easier ACID implementation within the overlying file-system.

Databases are also well suited to file-systems lacking inherent organization (such as tagging systems). Their efficient, indexed, look-ups are nearly essential for quickly accessing arbitrary items.

8.2.3.1 *Key/Value-Stores*

The most conceptually simple breed of database, the key/value-store effectively manifests as an on-disk associative array. A set of unique keys, whose format is either arbitrary or implementation specific, map one-to-one to arbitrary chunks of data. Many of these systems, including the original UNIX *dbm* (DataBase Manager) and many of it clones, are *very* simple, and lack features such as concurrent access and ACID compliant transactions. The former places a significant restriction on overlying file-systems, which are often accessed by multiple programs or multiple threads within a single program at a time. The later removes one of the major advantages of using a database over a file-system.

Berkeley DB
Berkeley DB, a notable exception to this trend of simplicity, pairs a basic Key/-Value system with a plethora of advanced features (fine-grained locking, ACID compliant transactions, etc.). By avoiding complex structure, Berekley DB trades functionality for speed making it a particularly viable option for file-system storage as evidenced by the Amino file-system (Wright et al., 2007). This Berkeley DB-backed system achieves performance, both in terms of CPU utilization and throughput, comparable to traditional file-systems while implementing ACID compliant transactions (Olson, Bostic, and Seltzer, 1999).

8.2.3.2 *Relational Databases – SQL*

While the Amino system certainly shows that a simple key/value-store is sufficient to implement a hierarchical file-system, complete with ACID compliant transactions, its implementation is largely dependent on the traditional path concept – each unique path maps to its corresponding file. Under STUFFS' new tagging scheme, uniqueness is reserved only for IDs, and most of the the browsing and file location is done via potentially non-unique, volatile properties (tags). Without a unique key, this common use then devolves into a linear-search.

Relational databases, such as the various SQL implementations, present data in a tabular fashion. Each entry in the database corresponds to a row in the table with an arbitrary number of designated values as columns. In general, any of these columns can be indexed and searched efficiently without necessarily being constrained by uniqueness. The addition of features such as foreign-keys makes implementing many-to-many tagging systems fairly straight-forward (see Section 8.2.4).

SQL Servers: PostgreSQL, MySQL, etc.

A number of the most popular SQL database engines – PostgreSQL, MySQL, Oracle, etc. – implement a client-server model in which a single server process manages the actual storage and data-structures while providing an interface for IPC calls. Any number of client programs can then communicate with the server to access the databases.

These systems are very powerful, generally providing the most features and highest performance of any option. Abstracting away the actual data-structure also keeps the interface simple and provides some ability to choose an appropriate storage format, or alter the implementation as needs change without changing the client programs. However, this power is not without its price. These systems generally require significant computational load and significant setup and maintenance investments. The existence of a server process also complicates the use of such systems on the kernel level – generally the level at which file-systems are implemented.

SQLite

SQLite is, as the name implies, a lighter approach to SQL systems. It eschews heavyweight client-server architecture in favor of allowing clients to interact with the data directly. A single, relatively small C library (wrappers exist for many other languages, including a stable and feature complete module for Python 3) implements a full SQL implementation complete with ACID compliant transactions and concurrency support.

This self-contained approach allows a relational database to be constructed, administered and manipulated entirely within the client, with negligible setup and configuration. Since it has minimal external dependencies (SQLite can be configured to use built-in memory allocation and avoid almost all external functions, other than those needed to access hardware), it can be implemented on nearly any architecture even at the kernel level if needed.

While this lighter SQL generally displays somewhat less performance (slower searches, more significant slowdown durring concurrent operations, etc.) than its heavier cousins, it was selected for use in STUFFS due to its dramatically easier setup and administration coupled with lower resource utilization.

8.2.4 *Table Schema*

STUFFS's SQLite database uses four tables interconnected by a system of foreign keys. The first three, *Files, Uses,* and *Tags* (figs. 8.1 to 8.3), implement fairly standard many-to-many relationships. Entries in the *Files* and *Tags* tables correspond to file and tag objects respectively. Each one stores some basic descriptors (an identification number and a name in the case of files, and simply the text of the tag in their case). They also store some basic attributes (generally the same information that would be contained in a traditional inode)– timestamps, size, owner, group, etc. – while these could be implemented as normal tags, they are accessed very frequently and lower latency greatly improves the overall performance. The *Uses* table consists entirely of pairs of foreign keys into the *Files* and *Tags* tables, each of which represents an application of a tag to the corresponding file. Indexing both columns allows for rapid searches of files by tags and vice versa.

A fourth table, *Data* (fig. 8.4), stores the actual file data in 4 Kilobyte blocks coupled with foreign keys to the corresponding file and a unique ID. Splitting the files into multiple blocks dramatically improves database performance on large files and random read/write operations. Additionally, keeping rows to a manageable size allows arbitrarily large (but smaller than the storage device capacity) files to be expressed without overflowing main memory.

8.3 syscall interface: fuse

Of course, storage management is largely useless without a means of accessing it, and, in the case of a file-system, presenting some interface to client applications. Fortunately, file-systems already have a standardized[2] interface at the kernel and system-library level. Therefore, providing a file-system access method which works seamlessly on all standards compliant applications without modification reduces to simply providing the expected results to the already existing system calls when directed at a STUFFS system.

Files

id (Integer) *Primary Key, Unique, Indexed*	**name** (Text) *Indexed*	**attrs** (Text)
. . .		
18	“some_file.txt”	*encoded meta-data*
19	“some_other_file.pdf”	*encoded meta-data*
. . .		

Figure 8.1: The *Files* table stores entries for each file object along with some meta-data.

Uses

file_id (Integer) *Indexed*	**tag_name** (Text) *Indexed*
. . .	
18	“some_tag”
357	“some_other_tag”
. . .	

Figure 8.2: The *Uses* table serves as a helper table for mapping files to and from tags.

Tags

name (Text) *Primary Key,* *Indexed,* *Unique*	**attrs** (Text)
. . .	
“some_tag”	*encoded meta-data*
“some_other_tag”	*encoded meta-data*
. . .	

Figure 8.3: The *Tags* table stores entries for each tag object along with some metadata.

Data

id (Integer) *Primary Key,* *Unique,* *Indexed*	**parent_id** (Integer) *Foreign Key→Files.id,* *Unique,* *Indexed*	**datum** (4KB BLOB)
. . .		

256	37	011000100110110
		001101111011000
		110110101100100
		00000110001. . .

257	18	001100100110111
		001100100001000
		000110001001101
		00101110100. . .

. . .		

Figure 8.4: The *Data* table stores the actual file data in 4 Kilobyte blocks.

For STUFFS development, Filesystem in Userspace (FUSE, `http://fuse.sourceforge.net/`) was selected thanks to its simple, straightforward integration with other components and its extensive functionality without requiring modifications of client software. Unlike some of the other methods considered, FUSE was designed specifically for implementing user-space file-systems, and it does an acceptable job.

8.4 DIRECT KERNEL INTEGRATION

Perhaps the most obvious solution is simply to build STUFFS directly into the kernel beside more traditional file-systems. While this method is certainly possible, it requires modifying the kernel, a non-trivial undertaking, which greatly complicates deployment (any systems using STUFFS must also use a custom, patched kernel which must be installed and maintained). This method is also incompatible with other design decisions (Python does not run in kernel-space, and pulling SQLite and other needed libraries into the kernel raises security, stability, and performance concerns).

8.4.1 *Library Modifcations & LD_PRELOAD*

Putting the file-system into user-space can alleviate many of the concerns with in-kernel implementations. Doing so then requires gaining control of file-system related syscalls before it hits the kernel. One method for doing this is to simply replace the necessary library functions with custom versions which act differently for STUFFS targets, either by using LD_PRELOAD to dynamically load in custom versions, or by replacing standard libraries entirely with patched versions.

Unfortunately, this method has two large issues: statically linked programs and cyclic calls. The former would need to be recompiled in order to use the new libraries, which may or may not be possible and goes against the drop-in backwards compatibility goal of STUFFS. Cyclic calls appear since functions such as `fread` and `fwrite` (standard C functions for file input and output) would need

2 More accurately, one might say there exist a number of competing standards. However, under a given operating system, there is generally a single most commonly used and encouraged one. STUFFS has been developed for Linux and, as such, it is the Linux kernel APIs and Glibc standard C library which are being considered. This distinction is, however, largely irrelevant to this discussion.

to be replaced, but these functions, in turn, are used by the SQLite library in STUFFS, creating an infinite loop.

8.4.2 *ptrace*

The process tracing facility *ptrace* provides another above-the-kernel option used in the Amino system and others. This system call allows a single "monitor" process to observe and control another. In this case, a single monitor process can be used to intercept system calls before they reach the kernel and filter out file-system access calls to be handled by the STUFFS system.

Unfortunately, a number of programs, such as OpenSSH, forbid being run under a ptrace which would prevent them for interacting with the file-system. The ptrace method also potentially incurs a significant performance penalty since *every* syscall would incur a context-switch to the monitor before continuing as usual in most cases.

8.4.3 *FUSE*

Filesystem in Userspace (FUSE) takes a very different approach. Instead of attempting to capture syscalls before they reach the kernel, FUSE provides a loadable kernel module which interfaces with the Virtual File-System (VFS). Before reaching the VFS layer, the syscall proceeds as it would for a typical file-system, but the FUSE module takes control at the level of a file-system protocol (much like Sun Microsystems' NFS). At this point, the call is redirected to the user-space fuse library and the specific program responsible for the data being accessed. This process is diagrammed in fig. 8.5.

FUSE takes advantage of the best of both worlds. On one hand, because they enter the call-chain at the kernel-level, FUSE-based file-systems are entirely indistinguishable from their native siblings by any user application and most of the kernel without any modifications. On the other, the file-system itself exists in user-space, allowing the use of the vast wealth of use-space libraries, such as SQLite, and programming languages and interpreters, including Python.

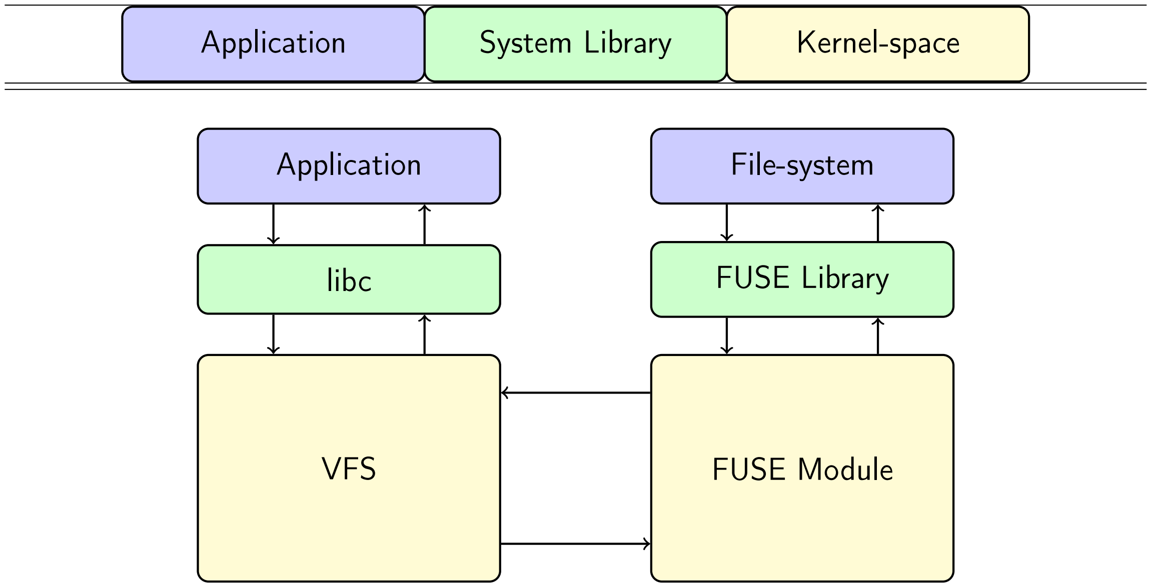


Figure 8.5: A simplified flow diagram of basic access to a FUSE file-system.

8.5 fuzzy matching: wordnet and nltk

STUFFS's semantic "Fuzzy Matching" of tags uses the WordNet library via the excellent Natural Language Toolkit (NLTK) available as a Python module. WordNet provides a massive lexical database of the English Language complete with word relations (conjugations, antonyms, synonyms, etc.). STUFFS's rudimentary proof-of-concept implementation uses a radically simplified, incomplete, and somewhat inaccurate method of defining similar terms as any direct synonyms and first order hyponyms, which are directly exposed by the WordNet system. In the future, a more advanced similarity metric would likely provide superior results, but this approach seemed reasonable for a proof of concept.

Part IV

THE END-GAME

In which the protagonist finally faces its Nemesis, armed only with the lessons learned on it's journey. . .

9

RESULTS

As stated, STUFFS is primarily a research system designed to establish a proof-of-concept system for a number of advanced file-system features. On those grounds, it appears to be a success. In its current state, STUFFS includes working implementations of all of its claimed features (see Chapter 5), including:

- Backwards compatibility
- Tagging
- Paths as Queries
- Semantic Resolution
- Unique Global File Identification
- ACID Transactions

Even while adding these features, STUFFS has remained fairly usable, largely thanks to its backwards-compatibility, and implements a fully functional file-system. Of course, comparing it with other file-systems in the most common way, performance benchmarks, paints a somewhat different picture. This is not particularly surprising or bothersome, but warrants noting.

9.1 USABILITY

As shown in Chapter 7, STUFFS is usable and beneficial under a variety of usages. Unfortunately, its relatively poor performance (as shown in Section 9.2) and lack of support will likely prevent widespread use. Additionally, while the file-system itself requires no configuration and an automated package build script exists for at least one operating system (Archlinux), installing STUFFS and its dependencies on a number of mainstream operating systems, including Windows, Mac OSX, and a number of linux distributions can be difficult.

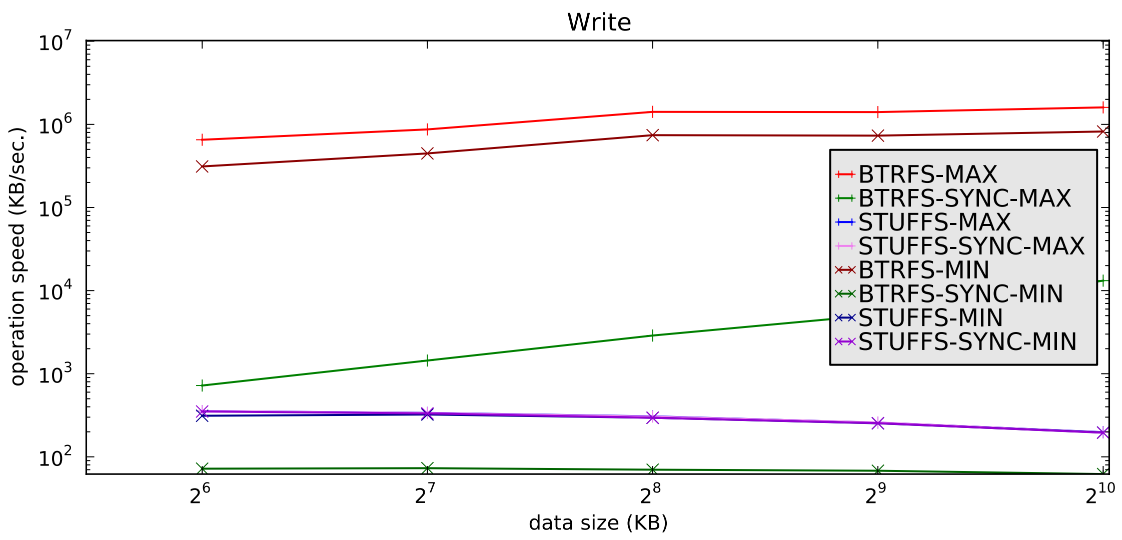
9.2 BENCHMARKS

The benchmarks presented in figs. 9.1 and 9.2 paint an interesting picture of STUFFS performance. These benchmarks were generated using the IOzone benchmarking tool (`www.iozone.org`) on a consumer grade laptop circa 2011 (Pentium B940 CPU, 5400RPM magnetic hard disk, 8GB DDR3 ram). Throughout the benchmarks, STUFFS is shown versus an on-disk BTRFS (`btrfs.wiki.kernel.org`) file-system. BTRFS is a modern, hierarchical file-system with performance on-par with other modern file-systems. The "BTRFS" and "STUFFS" plots indicate performance based on completion time reported by the respective file-system. The "-SYNC" variants include the time taken for the data to be synchronized to disk. For each variant, maximum and minimum performance ("-MAX" and "-MIN") across record lengths of $\{2^n | n \in \{6..\log_2(datasize)\}$.

Writing and rewriting under STUFFS exhibits remarkable consistency. Since STUFFS's ACID transactions require that data be committed to disk before a write can claim completion, the addition of explicit disk syncs is understandably negligible. More intriguingly, the STUFFS timings show minimal variation across record lengths, implying that there is little performance difference between a large number of small writes and a few larger ones – something not seen in traditional file-systems.

While a consistent write speed is desirable, STUFFS appears to have a consistently *slow* write speed, at least when compared to the speeds reported by BTRFS. This is, however, an unfair comparison. As mentioned, STUFFS is inherently synchronous, while BTRFS makes use of buffers, etc. to claim completion before the data actually reaches the disk. When looking purely at the synchronous timings, STUFFS and BTRFS appear to have much more similar performances, with STUFFS even pulling ahead for some record lengths.

Reading and rereading performance shows much more variance than writing and rewriting. Unlike writing, STUFFS does not inherently perform this operation synchronously and exhibits a noticeable speedup when not doing so. Furthermore, and again unlike the writing case, STUFFS (re)reading speed varies wildly across record length. For lengths near the total data size, non-synchronous



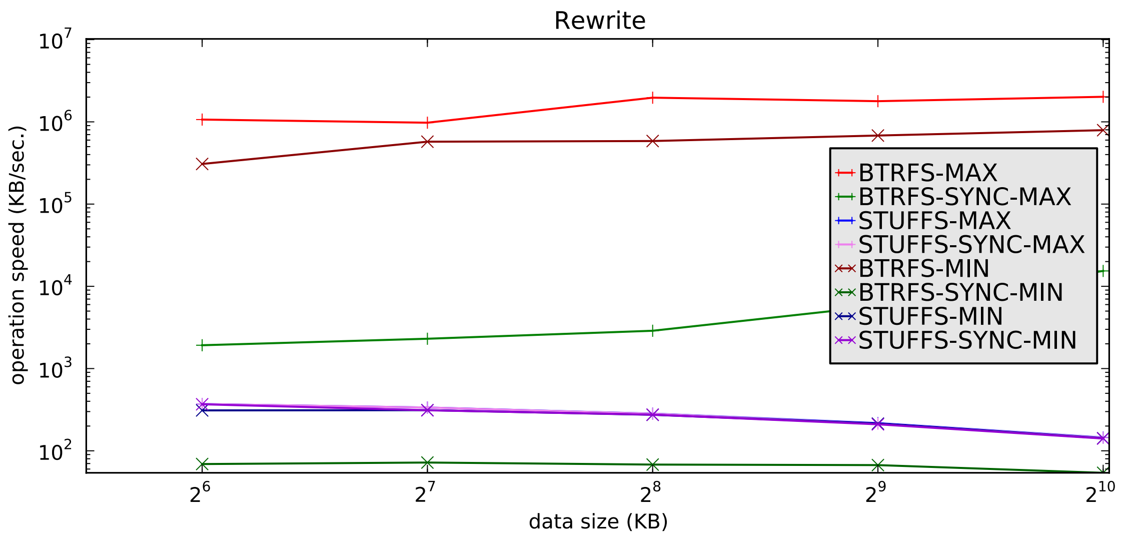


Figure 9.1: STUFFS write and rewrite performance benchmarks. Note, the write and rewrite speeds for "STUFFS" and "STUFFS-SYNC" (minimum and maximum) differ by, at most, tens of KB/sec. and, as such, their plots are largely overlapping.

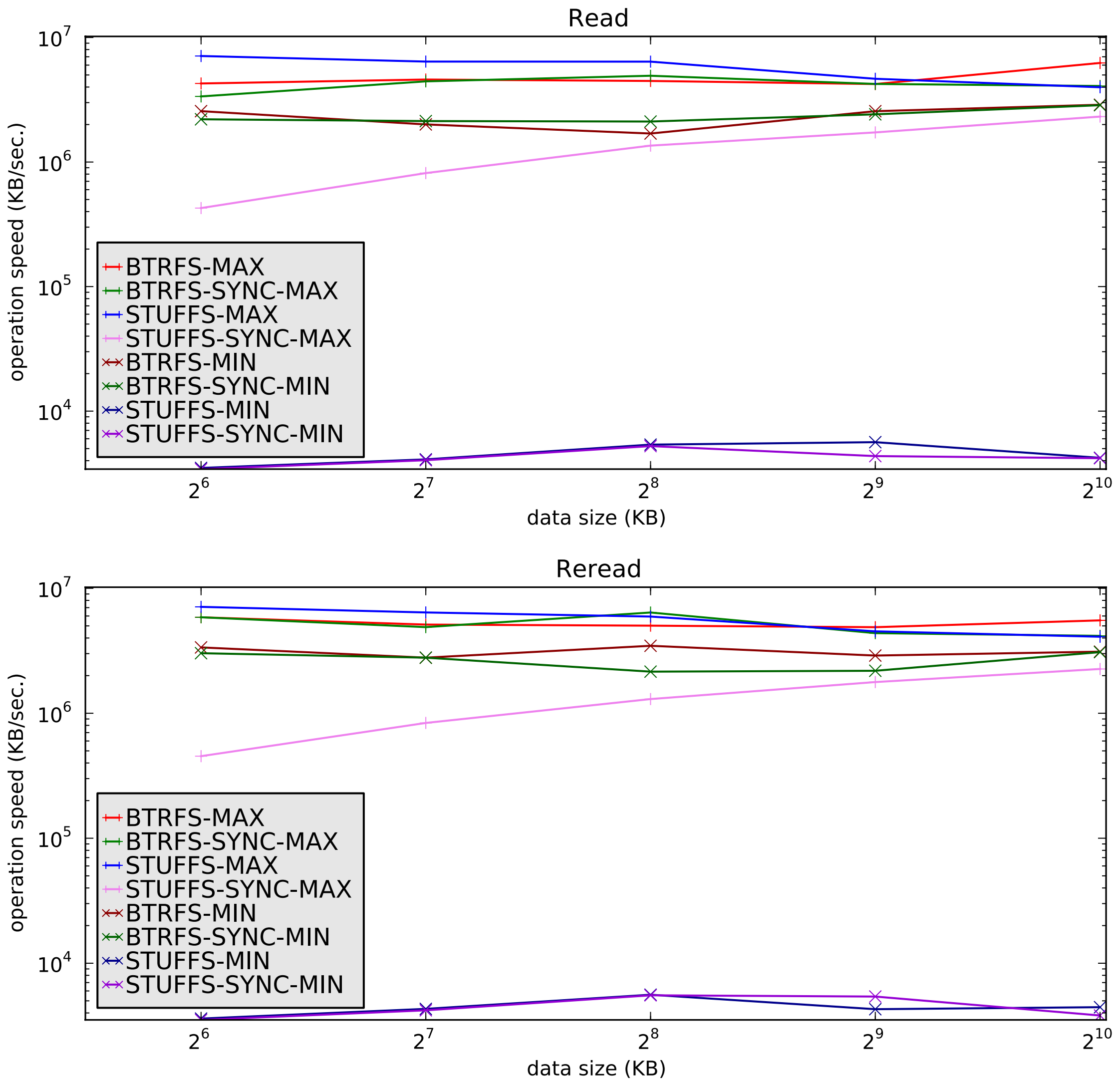


Figure 9.2: STUFFS read and reread performance benchmarks. Generation and analysis are largely identical to fig. 9.1

STUFFS performs even with or slightly better than BTRFS. On the other hand, for short record lengths, STUFFS slows by several orders of magnitude.

While these benchmarks are, of course, not entirely representative of performance, especially in real-world scenarios, they provide some amount of quantization. Real-world anecdotal evidence suggests that write speeds are in fact, very slow when one is accustomed to a modern traditional system. Read speeds, and intra-system moves (tagging and renaming) are not noticeably slower under normal use.

9.3 OUTSTANDING ISSUES

As with any sizable piece of software, STUFFS is not without bugs. While none of them are particularly show-stopping, they can be troublesome under particular circumstances which may appear during normal usage.

The most notable outstanding issue is less a bug, and more an unexpected, and undesired design decision. STUFFS development largely targeted command-line usage and considered GUI usage as secondary, largely based on the usage patterns of the author. Unfortunately this has left the GUI interface somewhat less polished. For example, while the syntax for union, negation, and fuzzy queries ("%tag%tag%", "!tag", and "?tag?") is accessible within a GUI file manger by explicitly entering them in a location bar or similar interface, there is no point-and-click method. Additionally, paths such as "/" with a large number of intersecting tags can be difficult to navigate due to the large number of displayed folders. These issues not exclusive to GUI file managers, of course, and largely hold for graphical FTP access and other point-and-click navigation schemes.

STUFFS also has compatibility issues with software which uses file extensions to infer file-type. Attempting to use such software with files specified using their id (i.e. "somefile.txt@1234@" rather than "somefile.txt"), as is the default for GUI file-choosers and auto-completion systems, will obviously cause problems since it clobbers the extension. This can be worked around by manually specifying the file without the id, but this removes the benefits of global file location and is less than ideal.

As a final outstanding bug, STUFFS lacks robust support for extended attributes. In the future, it would be desirable to implement a system which automatically maps these attributes to tags and their manipulation to tag manipulation.

9.4 FUTURE WORK

Apart from resolving the above issues and improving overall performance, STUFFS could benefit greatly from some extension and refinement of its existing implementation. As mentioned in Section 8.5, STUFFS's fuzzy matching algorithm is rudimentary at best and should be replaced by a proper semantic similarity system. Additionally, while STUFFS does implement transactions on a per-syscall basis automatically, implementing an interface to allow programs to manually create and commit transactions across system calls, as is done in the Amino system (Wright et al., 2007), would allow even more flexibility in application design.

10

CONCLUSION

STUFFS represents a natural exploration of alternative file-system design. While it cannot claim to be universally superior to current technology, it shows distinct advantages in a wide variety of applications, and serves as a proof that the old fragile hierarchies are not the *only* option. Hopefully, STUFFS and other such innovative redesigns will encourage continued work and bring new ideas to an otherwise slow-moving field.

Part V

APPENDIX

A

SOURCE CODE

A.1 STUFFS.PY

Most recent code may be found at http://www.github.com/aaronlaursen/STUFFS.
Here follows the current code at the time of writing.

```
#!/usr/bin/python3 -OO

#License, reuse, etc.
#--------------------
#
#This software was originally written by Aaron Laursen <aaronlaursen@gmail.com>.
#
#This software is licensed under the ISC (Internet Systems Consortium)
#license. The specific terms below for allow pretty much any reasonable use.
#If you, for some reason, need it in a different licence, send me an email,
#and we'll see what I can do.
#
#However, the author would appreciate but does not require (except as
#permitted by the ISC license):
#
#- Notification (by email preferably <aaronlaursen@gmail.com>) of use in
#products, whether open-source or commercial.
#
#- Contribution of patches or pull requests in the case of
#  improvements/modifications
#
#- Credit in documentation, source, etc. especially in the case of
```

```
#  large-scale projects making heavy use of this software.
#
#### ISC license
#
#Copyright (c) 2013, Aaron Laursen <aaronlaursen@gmail.com>
#
#Permission to use, copy, modify, and/or distribute this software for any
#purpose with or without fee is hereby granted, provided that the above
#copyright notice and this permission notice appear in all copies.
#
#THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
#WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
#MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
#ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
#WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
#ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
#OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.


from sqlalchemy import Table, Column, Integer, ForeignKey, BLOB, \
        Boolean, String, create_engine, MetaData
from sqlalchemy.orm import relationship, backref, sessionmaker, scoped_session
from sqlalchemy.ext.declarative import declarative_base
from time import time
#from sqlalchemy.dialects.mysql import VARCHAR, TEXT
from stat import S_IFDIR, S_IFLNK, S_IFREG
#from hashlib import md5
from fuse import Operations, LoggingMixIn, FUSE, FuseOSError
from sys import argv
from errno import ENOENT
from nltk.corpus import wordnet


#database stuff

from sqlalchemy.engine import Engine
from sqlalchemy import event

#'''
@event.listens_for(Engine, "connect")
def set_sqlite_pragma(dbapi_connection, connection_record):
    cursor = dbapi_connection.cursor()
    cursor.execute("PRAGMA foreign_keys=ON")
    cursor.execute('PRAGMA synchronous=OFF')
    cursor.execute('PRAGMA count_changes=OFF;')
    #cursor.execute('PRAGMA mmap_size=268435456;')
    cursor.close()
#'''

DBPATH="fs.db" if len(argv) <=2 else argv[2]
db = create_engine('sqlite:///'+DBPATH,connect_args={'check_same_thread':False})
#db = create_engine('sqlite:////tmp/stuffs.db')
#db = create_engine('mysql+oursql://stuffs:stuffs@localhost/stuffs_db')
db.echo = False
Base = declarative_base(metadata=MetaData(db))
Session = scoped_session(sessionmaker(bind=db))
#session=Session()

Table('use'
    , Base.metadata
    , Column('file_id', Integer, ForeignKey('files.id'), index=True)
    #, Column('tag_id', Integer, ForeignKey('tags.id'), index=True)
    , Column('tag_name', String, ForeignKey('tags.name'), index=True)
    #, mysql_engine = "InnoDB"
    #, mysql_charset= "utf8"
)

class Datum(Base):
    __tablename__='data'
    #__table_args__={
    #         'mysql_engine':'InnoDB'
    #         ,'mysql_charset':'utf8'
    #        }
    def __init__(self):
        self.datum=bytes()
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('files.id'), index=True)
    datum = Column(BLOB(length=4*1024))

class File(Base):
    __tablename__ = 'files'
    #__table_args__={
    #        'mysql_engine':'InnoDB'
    #        ,'mysql_charset':'utf8'
    #        }
    def __init__(self):
        pass
    id = Column(Integer, primary_key=True)
    #attrs = Column(String)
    attrs = Column(String(length=512))
    name = Column(String(length=256), index=True)
    data = relationship("Datum"
                    , collection_class=list
                    )
    tags = relationship("Tag"
                    , secondary="use"
                    , backref=backref("files", collection_class=set)
                    , collection_class=set
                    )

class Tag(Base):
    __tablename__ = 'tags'
    def __init__(self, txt):
        self.name=txt
    #id = Column(Integer, primary_key=True)
    name = Column(String(length=256), primary_key=True)
    attrs = Column(String(length=512))
```

```
Base.metadata.create_all()

def mkfile(name, session, mode=0o770, tags=None):
    f = File()
    session.add(f)
    if tags !=None:
        f.tags |= set(tags)
    now=time()
    a = {'st_mode':(S_IFREG | mode)
            , 'st_nlink':1
            , 'st_size':0
            , 'st_ctime':now
            , 'st_mtime':now
            , 'st_atime':now
            , 'uid':0
            , 'gid':0
            }
    f.attrs = convertAttr(a)
    f.name=name
    addBlock(f,session)
    #f.data=bytes()
    #print("****new file tags:", tags)
    return f

def mktag(txt, session, mode=0o777):
    t=Tag(txt)
    session.add(t)
    now=time()
    a = {'st_mode':(S_IFDIR | mode)
            , 'st_nlink':1
            , 'st_size':0
            , 'st_ctime':now
            , 'st_mtime':now
            , 'st_atime':now
            , 'uid':0
            , 'gid':0
```

```
                }
    t.attrs = convertAttr(a)
    return t

'''
def getAttrTag(obj, attr, session):
    q=session.query(Tag).filter(Tag.in_(obj.tags), Tag.name.like("attr::"+attr+"::
    return q.first()

def setAttrTag(obj, attr, value, session):
    obj.tags.discard(getAttrTag(obj,attr,session))
    t=getTagsByTxts("attr::"+attr+"::"+value)
#'''

def getSimTerms(term):
    t = wordnet.synsets(term)
    terms=set()
    for syn in t:
        print("???:",syn.lemma_names())
        for name in syn.lemma_names():
            terms.add(name)
        for hypo in syn.hyponyms():
            for name in hypo.lemma_names():
                terms.add(name)
        for hyper in syn.hypernyms():
            for name in hyper.lemma_names():
                terms.add(name)
    return terms

def getSimTagsFromTerm(term,session):
    terms=getSimTerms(term)
    tags=getTagsByTxts(set(terms),session)
    return tags

def getSimTags(tag,session):
    terms = getSimTerms(tag.name)
    tags=getTagsByTxts(terms,session)
    return tags

def convertAttr(attrs):
    attrdata=( ('st_mode',int)
            , ('st_nlink',int)
            , ('st_size',int)
            , ('st_ctime',float)
            , ('st_mtime',float)
            , ('st_atime',float)
            , ('uid', int)
            , ('gid', int)
            )
    if type(attrs) == type(dict()):
        s=''
        for i in range(len(attrdata)):
            s+=str(attrs[attrdata[i][0]])
            s+=','
        return s[:-1]
    if type(attrs) == type(''):
        attrs=attrs.split(',')
        d={attrdata[i][0]:attrdata[i][1](attrs[i]) for i in range(len(attrdata))}
        return d
    return None

def getIdFromString(s):
    t={'%':Tag,'@':File}
    if len(s) <3: return 0, File
    if s[-1] not in ('%','@'):
        return 0, File
    if len(s.split(s[-1]))<3:
        return 0, File
    i=s.split(s[-1])[-2]
    if not i.isdigit():
        return 0, File
    return int(i), t[s[-1]]
```

```
def genDisplayName(obj):
    if obj.__tablename__=='files':
        name=obj.name
        s='@'
        name += s + str(obj.id) +s
    elif obj.__tablename__=='tags':
        name=obj.name
        s='%'
    return name

def getByID(id_, session, typ=File):
    return session.query(typ).get(int(id_))

def getFilesByTags(tags,session):
    q=session.query(File)
    for t in tags:
        q=q.filter(File.tags.contains(t))
    return q.all()

def getFilesByLogicalTags(tags,session):
    if len(tags[0])+len(tags[1])+len(tags[2]) ==0: return None
    q=session.query(File)
    for t in tags[0]:
        q=q.filter(File.tags.contains(t))
    for t in tags[1]:
        q=q.filter(~File.tags.contains(t))
    #for op in tags[2]:
    #    q=q.filter(File.tags.isdisjoint(op[0]))
    #    q=q.filter(~File.tags.isdisjoint(op[1]))
    if len(tags[2])==0: return q.all()
    #t=set(q.all())
    t=set()
    for op in tags[2]:
        for i in op[0]:
            s=set(q.filter(File.tags.contains(i)).all())
            t|=s
        for i in op[1]:
            s=set(q.filter(~File.tags.contains(i)).all())
            t|=s
    return t

def getTagsByTxts(txts,session):
    q=session.query(Tag).filter(Tag.name.in_(txts))
    return q.all()

def getFilesByTagTxts(txts,session):
    tags=getTagsByTxts(txts,session)
    return getFilesByTags(tags,session)

def getTagsByFiles(files):
    tags=set()
    for f in files:
        tags |= f.tags
    return tags

def getTagsFromPath_logical(path,session):
    elems=set(path.split('/'))
    elems.discard('')
    parts=[set(),set(),[]] #[need,not,opt]
    if len(elems)==0: return parts
    for elem in elems:
        #or case
        if elem[0]=="%" and elem[-1]=="%":
            opts = elem[1:-1].split("%")
            p=set()
            n=set()
            for opt in opts:
                if opt[0]=="!" and len(opt)>1: n.add(opt[1:])
                else: p.add(opt)
            p=getTagsByTxts(p,session)
            n=getTagsByTxts(n,session)
```

```
                parts[2].append([p,n])
            elif elem[0]==elem[-1]=="?":
                print("asdf")
                e=elem[1:-1]
                neg=False
                if e[0]=="!":
                    e=e[1:]
                    neg=True
                #t=getTagsByTxts(set([elem[1:-1]]),session)
                #if len(t)==0: continue
                #simt=getSimTags(t[0],session)
                if len(e)<1: continue
                simt=getSimTagsFromTerm(e,session)
                if not neg: parts[2].append([simt,set()])
                else: parts[2].append([set(),simt])
                print(simt)
            elif elem[0]=="!" and len(elem)>1: parts[1].add(elem[1:])
            else: parts[0].add(elem)
        parts[0]=set(getTagsByTxts(parts[0],session))
        parts[1]=set(getTagsByTxts(parts[1],session))
        return parts

    def getTagsFromPath(path,session):
        #print("----------------------")
        #print("%"+path+"%")
        tagnames=set(path.split('/'))
        tagnames.discard('')
        #print(tagnames)
        #print("----------------------")
        if type(tagnames)==type(None): return set()
        if len(tagnames)==0: return set()
        idtags=set()
        for t in tagnames:
            id_,typ = getIdFromString(t)
            tag=getByID(id_, session, Tag)
            if tag: idtags.add(tag)
        nametags=set(getTagsByTxts(tagnames,session))
        return idtags | nametags

    def getEndTagFromPath(path,session):
        #if path=='/': return None
        path=path.strip('/')
        path=path.split("/")
        tagname=path[-1]
        if tagname=='': return None
        id_, typ = getIdFromString(tagname)
        tag=getByID(id_, session, Tag)
        if tag: return tag
        return getTagsByTxts(tagname,session)[0]

    def getFileByNameAndTags(name,tags,session):
        #print(tags)
        if len(tags)==0:return None
        q=session.query(File).filter(File.name==name)
        for t in tags:
            q=q.filter(File.tags.contains(t))
        return q.first()

    def getFileByNameAndLogicalTags(name,tags,session):
        if len(tags[0])+len(tags[1])+len(tags[2]) ==0: return None
        q=session.query(File).filter(File.name==name)
        for t in tags[0]:
            q=q.filter(File.tags.contains(t))
        for t in tags[1]:
            q=q.filter(~File.tags.contains(t))
        #for op in tags[2]:
        #    q=q.filter(File.tags.isdisjoint(op[0]))
        #    q=q.filter(~File.tags.isdisjoint(op[1]))
        #return q.first()
        if len(tags[2])==0: return q.first()
        for op in tags[2]:
            for i in op[0]:
                s=q.filter(File.tags.contains(i)).first()
                if s: return s
            for i in op[1]:
                s=q.filter(~File.tags.contains(i)).first()
                if s: return s
        return None

    def getFileFromPath(path,session):
        path=path.strip('/')
        pieces=path.split('/')
        fstring=pieces[-1]
        fid,typ=getIdFromString(fstring)
        f=getByID(fid, session, File)
        if f: return f
        if len(pieces) < 2: return None
        path = ""
        for p in pieces[:-1]: path +=p+"/"
        return getFileByNameAndLogicalTags(fstring,
                getTagsFromPath_logical(path,session),session)

    def getSubByTags(tags,session):
        if len(tags)==0:return genAllTags(session)
        subfiles=set(getFilesByTags(tags,session))
        subtags=getTagsByFiles(subfiles)
        subtags=subtags-tags
        #print("{}{}{}{}{}{}{}")
        #print(subfiles,subtags)
        #print("{}{}{}{}{}{}{}")
        return subfiles | subtags

    def getSubByTags_logical(tags,session):
        if len(tags[0])+len(tags[1])+len(tags[2])==0:return genAllTags(session)
        subfiles=set(getFilesByLogicalTags(tags,session))
        subtags=getTagsByFiles(subfiles)
        subtags=subtags-tags[0]-tags[1]
        return subfiles | subtags
```

```
def genSub(path,session):
    tags=getTagsFromPath(path,session)
    #print("\n tags from subpath", path,tags,"\n")
    sub=getSubByTags(tags,session)
    #print("###########")
    #print(sub)
    #print("###########")
    return sub

def genSubLogical(path,session):
    tags=getTagsFromPath_logical(path,session)
    sub=getSubByTags_logical(tags,session)
    return sub

def genSubDisplay(path,session):
    sub=genSub(path,session)
    return [genDisplayName(x) for x in sub]

def genSubDisplayLogical(path,session):
    sub=genSubLogical(path,session)
    return [genDisplayName(x) for x in sub]

def getAttrByObj(obj):
    return convertAttr(obj.attrs)

def getObjByPath(path,session):
    if path[-1]=='/':
        return getEndTagFromPath(path,session)
    objname=path.split('/')[-1]
    #print("============")
    #print(objname)
    #print(getIdFromString(objname))
    #print("============")
    obj=None
    id_, typ = getIdFromString(objname)
    obj = getByID(id_, session,typ)
    if obj: return obj
    pathpieces=path.rsplit('/',1)
    opts=genSubLogical(pathpieces[0]+'/',session)
    if pathpieces[1][0]==pathpieces[1][-1]=="%":
        ors=set(pathpieces[1].split("%"))
        ors.discard('')
        ors=set(getTagsByTxts(ors, session))
        if len(ors)>=1 and not ors.isdisjoint(opts):
            return list(ors.intersection(opts))[0]
    for o in opts:
        if o.name==pathpieces[1]: return o
        if "!"==pathpieces[1][0] and o.name==pathpieces[1][1:]: return o
    if typ == File and len(path.split('/'))>2 and \
            'ALLFILES' not in path.split('/'):
        obj = getFileByNameAndLogicalTags(objname.rsplit('@',2)[0],
                getTagsFromPath_logical(path,session),session)
        return obj
    return getFileFromPath(path,session)

def genEverything(session):
    stuff=set()
    q=session.query(File)
    stuff |= set(q.all())
    q=session.query(Tag)
    stuff |= set(q.all())
    #print("------stuff:",stuff)
    return stuff

def genAllTags(session):
    stuff=set(session.query(Tag).all())
    return stuff

def genAllFiles(session):
    stuff=set(session.query(File).all())
    return stuff
```

```
def genDisplayEverything(session):
    stuff=genEverything(session)
    return [genDisplayName(obj) for obj in stuff]

def genDisplayAllTags(session):
    stuff=genAllTags(session)
    return [genDisplayName(obj) for obj in stuff]

def genDisplayAllFiles(session):
    stuff=genAllFiles(session)
    return [genDisplayName(obj) for obj in stuff]

def getAttrByPath(path,session):
    obj=getObjByPath(path,session)
    if not obj: return None
    return getAttrByObj(obj)

def rmObj(obj,session):
    session.delete(obj)

def rmByPath(path,session):
    obj=getObjByPath(path,session)
    if not obj: return None
    rmObj(obj,session)

def addBlock(f,session):
    block=Datum()
    session.add(block)
    f.data.append(block)
    #block.parent_id=f.id
    #session.flush()
    return f

def delBlock(f,session):
    session.delete(f.data.pop())
```

```
        #session.flush()
        return f

#fuse stuff
class STUFFS(LoggingMixIn, Operations):
    def __init__(self):
        self.fd=0
        #self.session=Session()
        self.blocksize=4*1024

    def getattr(self, path, fh=None):
        #print("getattr:", path, fh)
        session=Session()
        attr=None
        if path.strip()=='/' or path.split('/')[-1]=='ALLFILES' \
            or (path.split('/')[-2]=='ALLFILES' and path.split('/')[-1]=='') \
            or (path.split("/")[-1][0]==path.split("/")[-1][-1]=="?"):
            attr= {'st_mode':(S_IFDIR | 0o777)
                , 'st_nlink':2
                , 'st_size':0
                , 'st_ctime':time()
                , 'st_mtime':time()
                , 'st_atime':time()
                , 'uid':0
                , 'gid':0
                }
        pieces=path.rsplit("/",1)
        if len(pieces)>0 and len(pieces[-1])>0 and pieces[-1][0]=='!':
            pieces[-1]=pieces[-1][1:]
            path='/'.join(pieces)
        if not attr:
            attr=getAttrByPath(path,session)
        #print("+++++++++")
        #print(attr)
        #print("+++++++++")
        if not attr:
            raise FuseOSError(ENOENT)
        return attr

    def mkdir(self,path,mode):
        session=Session()
        path=path.strip('/')
        path=path.split('/')
        txt=path[-1]
        mktag(txt, session, mode)
        session.commit()
        Session.remove()

    def readdir(self,path,fh=None):
        #print("readdir")
        session=Session()
        #if path=='/': return ['.','..']+genDisplayEverything(session)
        if path=='/': return ['.','..']+genDisplayAllTags(session)+['ALLFILES']
        if 'ALLFILES' == path.split('/')[-1] or \
            ('ALLFILES'==path.split('/')[-2] and ''==path.split('/')[-1]):
            return ['.','..']+genDisplayAllFiles(session)
        return ['.','..']+genSubDisplayLogical(path,session)

    def chmod(self, path, mode):
        session=Session()
        obj=getObjByPath(path,session)
        if not obj: return
        attrs=convertAttr(obj.attrs)
        attrs['st_mode'] |=mode
        obj.attrs=convertAttr(attrs)
        session.commit()
        Session.remove()
        return 0

    def chown(self, path,uid,gid):
        session=Session()
        obj=getObjByPath(path,session)
        if not obj: return
        attrs=convertAttr(obj.attrs)
        attrs['uid']=uid
        attrs['gid']=gid
        obj.attrs=convertAttr(attrs)
        session.add(obj)
        session.commit()
        Session.remove()

    def create(self,path,mode):
        #print("creat reached:",path,mode)
        session=Session()
        tpath, name = path.rsplit("/",1)
        tags=getTagsFromPath_logical(path,session)[0]
        mkfile(name,session,tags=tags)
        session.commit()
        Session.remove()
        self.fd +=1
        return self.fd

    def open(self,path,flags):
        #print("open reached:",path,flags)
        self.fd+=1
        return self.fd

    def read(self,path,size,offset,fh):
        #print("read")
        session=Session()
        f=getFileFromPath(path,session)
        if not f: return ""
        #print(":-:-:",f.data[offset:offset+size])
        #return f.data[offset:offset+size]
        data=bytes()
        blockoffs=offset//self.blocksize
        offset=offset%self.blocksize
        while size >0:
            #print(data)
            if blockoffs>=len(f.data): break
            data+=f.data[blockoffs].datum[offset:min(self.blocksize,
                size+offset)]
            size-=(self.blocksize-offset)
            blockoffs+=1
            offset=0
            #print("Loop!")
        #print(len(data))
        #print(data)
        #print(data.decode())
        return data

    def write(self,path,data,offset,fh):
        #print("write")
        #print(data)
        #print(type(data))
        session=Session()
        f=getFileFromPath(path,session)
        if not f: return
        #f.data=f.data[:offset]+data
        size=len(data)
        attrs=convertAttr(f.attrs)
        attrs['st_size']=offset+size
        f.attrs=convertAttr(attrs)
        blockoffs=offset//self.blocksize
        offset=offset%self.blocksize
        #print("offset:",offset)
        start=0
        while start<size:
            while blockoffs>=len(f.data):
                f=addBlock(f,session)
            f.data[blockoffs].datum=f.data[blockoffs].datum[:offset]+ \
                    data[start:start+min(size-start,self.blocksize-offset)]
            start+=min(size-start,self.blocksize-offset)
            offset=0
```

```python
            blockoffs+=1
            #print("loop!")
        session.commit()
        Session.remove()
        #print(size)
        return size

    def truncate(self, path, length, fh=None):
        #print("truncate")
        session=Session()
        f=getFileFromPath(path,session)
        if not f: return
        #f.data=f.data[:length]
        numblocks=(length+self.blocksize-1)//self.blocksize
        while numblocks>len(f.data):
            f=addBlock(f,session)
        while numblocks>len(f.data):
            f=delBlock(f,session)
        if numblocks>0:
            f.data[-1].datum=f.data[-1].datum[:length%self.blocksize]
        attrs=convertAttr(f.attrs)
        attrs['st_size']=length
        f.attrs=convertAttr(attrs)
        session.commit()
        Session.remove()

    def utimens(self, path, times=None):
        now=time()
        atime, mtime = times if times else (now,now)
        session=Session()
        f=getFileFromPath(path,session)
        if not f: return
        attrs=convertAttr(f.attrs)
        attrs['st_atime']=atime
        attrs['st_mtime']=mtime
        f.attrs=convertAttr(attrs)
        session.commit()
        Session.remove()

    def rmdir(self,path):
        session=Session()
        rmByPath(path,session)
        session.commit()
        Session.remove()

    def unlink(self, path):
        session=Session()
        rmByPath(path,session)
        session.commit()
        Session.remove()

    def rename(self, old, new):
        session=Session()
        #pieces=set(new.split('/')[:-1])
        '''
        npieces=set()
        for p in pieces:
            if len(p)<2:continue
            if p[0]=='!':
                npieces.add(p)
        pieces-=npieces
        npieces=list(npieces)
        for i in range(len(npieces)):
            npieces[i]=npieces[i][1:]
        new='/'+'/'.join(pieces)
        nnew='/'+'/'.join(npieces)
        tags=getTagsFromPath(new,session)
        ntags=getTagsFromPath(nnew,session)
        '''
        tags=getTagsFromPath_logical(new,session)
        f=getObjByPath(old,session)
        f.tags-=set(tags[1])
```

```
            f.tags|=set(tags[0])
            session.commit()
            Session.remove()

    def readlink(self, path):
        return self.read(path,float("inf"),0,None)

if __name__ == "__main__":
    if len(argv) < 2:
        print('usage: %s <mountpoint> [database]' % argv[0])
        exit(1)
    fuse = FUSE(STUFFS(), argv[1], foreground=True)
```

B

RAW IOZONE OUTPUT

Here follows the raw output of the IOzone benchmarks.

B.1 BTRFS

```
Iozone: Performance Test of File I/O
        Version $Revision: 3.420 $
Compiled for 64 bit mode.
Build: linux-AMD64

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
             Vangel Bojaxhi, Ben England, Vikentsi Lapa.

Run began: Fri Feb 21 20:48:02 2014

Auto Mode 2. This option is obsolete. Use -az -i0 -i1
Using maximum file size of 1024 kilobytes.
Command line used: iozone -A -g 1024K
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

              KB  reclen   write rewrite    read    reread
              64       4  311842  307554  2923952  3958892
              64       8  435202 1066042  2561267  3363612
              64      16  653436  492717  2662899  5283570
              64      32  524487  735831  4274062  5860307
              64      64  591520  821391  3057153  5735102
             128       4  447464  573699  2004703  2784517
             128       8  576161  780556  2985839  4934216
             128      16  677178  753179  2660335  3445772
             128      32  870405  836500  3199360  5122535
             128      64  800337  814915  3982553  5074121
             128     128  800337  976473  4596273  4934216
             256       4  742146  584570  1694118  3557725
             256       8 1143731  995306  1814348  3465855
             256      16 1012194 1027695  2783115  4332998
             256      32  900936 1070737  2247235  3654598
             256      64 1415041 1312954  3705040  5022044
             256     128 1147398 1967260  3654598  4197489
             256     256 1243036 1080434  4477549  4652146
             512       4  732478  681782  2560168  3504346
             512       8 1001715  803757  2811557  3487274
             512      16  911184 1145443  3104171  4610256
             512      32 1084694 1103647  2625909  3710197
             512      64 1055894 1497750  4195896  3934520
             512     128 1406520 1422357  4228947  4882800
             512     256 1042566 1783912  3736016  4131319
             512     512 1098566 1541840  3877684  2894940
            1024       4  821247  792019  2976816  3684119
            1024       8  977950 1057924  2882904  4146499
            1024      16 1155864 1136293  3955557  4674510
            1024      32 1030263 1477555  3556008  4083422
            1024      64 1497127 1602715  3112733  3481072
            1024     128 1247519 1359680  3532609  3414650
```

```
          1024     256 1148139 2016600  6244448  5536137
          1024     512 1193442 1651398  3618930  3110479
          1024    1024 1602118 1600327  3261656  3311958

iozone test complete.
```

B.2 btrfs-sync

```
Iozone: Performance Test of File I/O
        Version $Revision: 3.420 $
Compiled for 64 bit mode.
Build: linux-AMD64

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
             Vangel Bojaxhi, Ben England, Vikentsi Lapa.

Run began: Fri Feb 21 20:45:46 2014

Auto Mode 2. This option is obsolete. Use -az -i0 -i1
Include close in write timing
Include fsync in write timing
SYNC Mode.
Using maximum file size of 1024 kilobytes.
Command line used: iozone -Aceo -g 1024K
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
```

```
  KB  reclen   write rewrite    read    reread
  64       4      72      69  2203800  3791156
  64       8     155     151  3203069  4564786
  64      16     274     443  2203800  3022727
  64      32     721     718  3363612  5860307
  64      64     719    1922  2662899  4897948
 128       4      73      72  2132084  2784517
 128       8     185     185  2571150  3982553
 128      16     349     311  3124872  3867787
 128      32     523     640  2843510  4889281
 128      64    1439     886  2464907  4557257
 128     128    1441    2305  4444086  4267461
 256       4      70      68  2114473  2152625
 256       8     157     157  3123106  4350555
 256      16     319     329  3705040  4264168
 256      32     677     606  3756894  4264168
 256      64    1047    1441  4929815  6398720
 256     128    2304    1772  3717869  4422226
 256     256    2881    2882  3197509  3605511
 512       4      68      67  2416145  2187279
 512       8     144     132  3099691  4163357
 512      16     274     260  3068685  4061007
 512      32     606     561  2708713  4061007
 512      64    1071    1440  3556580  4375425
 512     128    1772    2560  3372274  3905895
 512     256    3072    3293  3610395  4000485
 512     512    5122    5768  4237291  4228947
1024       4      62      54  3010197  3436508
1024       8     134     130  4079544  3738636
1024      16     297     292  3567824  3659010
1024      32     639     631  3436508  3436508
1024      64    1097    1335  3594699  4014717
1024     128    2792    2248  3458646  4162573
1024     256    5115    4609  3355952  3712781
1024     512    6584    6151  3311958  3955557
1024    1024   13182   15390  2859868  3083680
```

```
iozone test complete.
```

B.3 stuffs

```
Iozone: Performance Test of File I/O
        Version $Revision: 3.420 $
Compiled for 64 bit mode.
Build: linux-AMD64

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
             Vangel Bojaxhi, Ben England, Vikentsi Lapa.

Run began: Fri Feb 21 20:42:38 2014

Auto Mode 2. This option is obsolete. Use -az -i0 -i1
Using maximum file size of 1024 kilobytes.
Command line used: iozone -A -g 1024K
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

              KB  reclen   write rewrite    read    reread
              64       4     346     367     3620     3605
              64       8     351     310     3497     3702
              64      16     355     370     7392     6909
              64      32     354     370  7100397  7100397
              64      64     312     370  3363612  5735102
             128       4     334     332     4312     4367
             128       8     335     333     4096     4310
             128      16     334     334     6608     6387
             128      32     323     334    12026    12178
             128      64     336     312  6406138  6406138
             128     128     338     336  5784891  6114306
             256       4     303     275     5392     5648
             256       8     305     274     5381     5595
             256      16     305     282     6960     7046
             256      32     295     283    10567     9988
             256      64     306     283    19046    20241
             256     128     299     275  6398720  5938650
             256     256     307     283  5117791  4907284
             512       4     253     215     5653     5534
             512       8     253     216     5632     4286
             512      16     257     214     6643     6465
             512      32     258     214     8138     8570
             512      64     255     217    10755    10977
             512     128     255     217    13701    15063
             512     256     254     216    17037    17013
             512     512     258     216  4660280  4522868
            1024       4     197     142     4597     4448
            1024       8     197     145     4216     4679
            1024      16     196     143     5150     5144
            1024      32     197     143     5853     5175
            1024      64     197     143     6566     6391
            1024     128     198     143     7159     6243
            1024     256     198     143     7848     7488
            1024     512     198     144    11871    11461
            1024    1024     197     144  3984917  4095102

iozone test complete.
```

B.4 STUFFS-SYNC

```
Iozone: Performance Test of File I/O
```

```
        Version $Revision: 3.420 $
Compiled for 64 bit mode.
Build: linux-AMD64

Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
             Al Slater, Scott Rhine, Mike Wisner, Ken Goss
             Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
             Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
             Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
             Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
             Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
             Vangel Bojaxhi, Ben England, Vikentsi Lapa.

Run began: Fri Feb 21 20:39:25 2014

Auto Mode 2. This option is obsolete. Use -az -i0 -i1
Include close in write timing
Include fsync in write timing
SYNC Mode.
Using maximum file size of 1024 kilobytes.
Command line used: iozone -Aceo -g 1024K
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.

             KB  reclen   write rewrite    read    reread
             64       4     351     369     3422     3607
             64       8     351     368     3496     3519
             64      16     353     368     6684     6888
             64      32     355     370   333540   316623
             64      64     354     370   426897   453587
            128       4     333     332     4235     4256
            128       8     334     333     4035     4199
            128      16     333     311     6436     6375
```

```
             128      32     337     336   12798   11966
             128      64     337     335  814915  836500
             128     128     337     335  795593  776042
             256       4     303     275    5238    5555
             256       8     305     282    5431    5535
             256      16     305     276    6983    7094
             256      32     305     282    9967   10537
             256      64     304     275   19078   18932
             256     128     295     282   54076   55580
             256     256     304     275 1354356 1298662
             512       4     254     213    4354    5459
             512       8     257     210    5472    5410
             512      16     254     214    6537    6384
             512      32     255     214    7943    8272
             512      64     256     214   10736   10786
             512     128     259     214   13737   14412
             512     256     257     211   16873   16452
             512     512     258     212 1729323 1776533
            1024       4     196     141    4508    3817
            1024       8     196     144    4186    4516
            1024      16     195     141    5228    5092
            1024      32     196     143    5797    4402
            1024      64     196     143    6511    6368
            1024     128     196     142    7190    7864
            1024     256     196     144    7895    7652
            1024     512     196     144   11518   11330
            1024    1024     197     143 2316837 2260740

iozone test complete.
```

BIBLIOGRAPHY

Armbrust, Michael, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin (2010). "A view of cloud computing." en. In: *Communications of the ACM* 53.4, p. 50. DOI: 10.1145/1721654.1721672.

Baron, Joseph G. and Robert Schneider (2010). "Storage Options in the AWS Cloud." In: *Amazon White Papers* (cit. on p. 29).

Bhagwat, Deepavali and Neoklis Polyzotis (2005). "Searching a file system using inferred semantic links." In: ACM Press, p. 85. DOI: 10.1145/1083356.1083372.

Bloehdorn, Stephan, Olaf Görlitz, Simon Schenk, and Max Völkel (2006). "Tagfs-tag semantics for hierarchical file systems." In: *Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06)*, pp. 6–8.

Cachin, Christian, Idit Keidar, and Alexander Shraer (2009). "Trusting the cloud." en. In: *ACM SIGACT News* 40.2, p. 81. DOI: 10.1145/1556154.1556173 (cit. on p. 29).

Capra, Robert G. and Manuel A. Pérez-Quiñones (2003). "Re-finding found things: An exploratory study of how users re-find information." In: *arXiv preprint cs/0310011*.

Center for History and New Media. *Zotero Quick Start Guide*.

Douceur, John R. and William J. Bolosky (1999). "A large-scale study of file-system contents." In: *ACM SIGMETRICS Performance Evaluation Review* 27.1, pp. 59–70. DOI: 10.1145/301464.301480.

Dourish, Paul, W. Keith Edwards, Anthony LaMarca, and Michael Salisbury (1999). "Presto: an experimental architecture for fluid interactive document spaces." In: *ACM Transactions on Computer-Human Interaction* 6.2, pp. 133–161. DOI: 10.1145/319091.319099 (cit. on p. 30).

Eric Thomas Freeman (1997). "The Lifestreams Software Architecture." Phd. Yale University.

Faubel, Sebastian and Christian Kuschel (2008). "Towards Semantic File System Interfaces." In: *International Semantic Web Conference (Posters & Demos)* (cit. on p. 60).

Garcia-Molina, Hector (2009). *Database systems: the complete book*. 2nd ed. Upper Saddle River, N.J: Pearson Prentice Hall (cit. on p. 14).

Giampaolo, Dominic (1999). *Practical file system design with the BE file system*. San Francisco: Morgan Kaufmann Publishers (cit. on pp. 9, 13, 28).

Gifford, David K., Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole (1991). "Semantic file systems." In: *13TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*. ACM Press, pp. 16–25. DOI: 10.1145/121132.121138 (cit. on p. 30).

Golder, S. A. (2006). "Usage patterns of collaborative tagging systems." In: *Journal of Information Science* 32.2, pp. 198–208. DOI: 10.1177/0165551506062337.

Gray, Jim (1981). "The transaction concept: Virtues and limitations." In: *VLDB*. Vol. 81, pp. 144–154 (cit. on pp. 21, 23).

Hans Reiser (2001). *The Naming System Venture* (cit. on p. 30).

Harper, Richard, Siân Lindley, Eno Thereska, Richard Banks, Philip Gosset, Gavin Smyth, William Odom, and Eryn Whitworth (2013). "What is a file?" In: ACM Press, p. 1125. DOI: 10.1145/2441776.2441903 (cit. on p. 9).

Homer, Michael (2014). "An updated directory structure for Unix." In: (cit. on p. 26).

James W. O'Toole and David K. Gifford (1992). "Names should mean What, not Where." In: (cit. on p. 16).

Jesse Phillips (2006). *TagFS – Nascent* (cit. on p. 30).

Jody Foo (2003). "DocPlayer: Design Insights from Applying the Non-HierarchicalMedia-Player model to Document Management." English. Independent thesis Advanced level (degree of Magister). Linköping University, Department of Computer and Information Science (cit. on p. 27).

Jones, William, Ammy Jiranida Phuwanartnurak, Rajdeep Gill, and Harry Bruce (2005). "Don't take my folders away!: organizing personal information to get things done." In: ACM Press, p. 1505. DOI: 10.1145/1056808.1056952.

Katcher, Jeffrey (1997). *Postmark: A new file system benchmark*. Tech. rep. Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html.

Korn, David G. and Eduardo Krell (1990). "A new dimension for the Unix® file system." en. In: *Software: Practice and Experience* 20.S1, S19–S34. DOI: 10.1002/spe.4380201304.

Korth, Henry F. (1991). *Database system concepts*. 2nd ed. McGraw-Hill computer science series. New York: McGraw-Hill (cit. on p. 14).

Martin, Ben (2004). "Formal Concept Analysis and Semantic File Systems." In: *Concept Lattices*. Ed. by Peter Eklund. Lecture Notes in Computer Science 2961. Springer Berlin Heidelberg, pp. 88–95.

Mason, Chris (2007). "The btrfs filesystem." In: *The Orcale cooperation*.

Miller, George A. (1956). "The magical number seven, plus or minus two: some limits on our capacity for processing information." In: *Psychological review* 63.2, p. 81 (cit. on p. 19).

Muhammad, Hisham, André Detsch, and São Leopoldo-RS-Brasil (2002). "AN ALTERNATIVE FOR THE UNIX DIRECTORY STRUCTURE£." In: *Proceedings of the III WSL-Workshop em Software Livre, Porto Alegre* (cit. on p. 26).

Nielsen, Jakob (1994). *Usability engineering*. English. San Francisco, Calif.: Morgan Kaufmann Publishers.

Oleksik, Gerard, Max L. Wilson, Craig Tashman, Eduarda Mendes Rodrigues, Gabriella Kazai, Gavin Smyth, Natasa Milic-Frayling, and Rachel Jones (2009). "Lightweight tagging expands information and activity management practices." In: ACM Press, p. 279. DOI: 10.1145/1518701.1518746.

Olson, Michael A., Keith Bostic, and Margo I. Seltzer (1999). "Berkeley DB." In: *USENIX Annual Technical Conference, FREENIX Track*, pp. 183–191 (cit. on p. 61).

Owens, Michael and Grant Allen (2006). *The definitive guide to SQLite*. Vol. 1. Springer.

Padioleau, Yoann and Oliver Ridoux (2003). "A Logic File System." In: *USENIX Association Proceedings of the General Track: 2003 USENIX Annual Technical Conference*. San Antonio, Texas, USA: USENIX Association, pp. 99–112.

Padioleau, Yoann and Olivier Ridoux (2005). "A Parts-of-File File System." In: *USENIX Annual Technical Conference, General Track*, pp. 359–362.

Padioleau, Yoann, Benjamin Sigonneau, and Olivier Ridoux (2006). "LISFS: a logical information system as a file system." In: Shanghai, China: ACM Press, p. 803. DOI: 10.1145/1134285.1134418 (cit. on p. 30).

Pike, Rob, Dave Presotto, Ken Thompson, and Howard Trickey (1990). "Plan 9 from bell labs." In: *Proceedings of the summer 1990 UKUUG Conference*, pp. 1–9 (cit. on p. 14).

Pike, Rob, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom (1992). "The use of name spaces in Plan 9." In: *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*. ACM, pp. 1–5 (cit. on p. 14).

Ramakrishnan, Raghu (2003). *Database management systems*. 3rd ed. Boston: McGraw-Hill (cit. on p. 8).

Ritchie, Dennis M. and Ken Thompson (1974). "The UNIX time-sharing system." In: *Communications of the ACM* 17.7, pp. 365–375. DOI: 10.1145/361011.361061.

Rodeh, Ohad, Josef Bacik, and Chris Mason (2013). "Btrfs: The linux b-tree filesystem." In: *ACM Transactions on Storage (TOS)* 9.3, p. 9.

Sauermann, Leo, Gunnar Aastrand Grimnes, Malte Kiesel, Christiaan Fluit, Heiko Maus, Dominik Heim, Danish Nadeem, Benjamin Horak, and Andreas Dengel (2006). "Semantic Desktop 2.0: The Gnowsis Experience." In: *The Semantic Web - ISWC 2006*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora M. Aroyo. Vol. 4273. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 887–900 (cit. on p. 60).

Saxbeck Larsen, Frans (2011). "Tag-based Data Organisation." PhD thesis. Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark.

Schandl, Bernhard and Bernhard Haslhofer (2009). "The Sile Model – A Semantic File System Infrastructure for the Desktop." In: *The Semantic Web: Research and Applications*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Lora Aroyo, Paolo Traverso, Fabio Ciravegna, Philipp Cimiano, Tom Heath, Eero Hyvönen, Riichiro Mizoguchi, Eyal Oren, Marta Sabou, and Elena Simperl. Vol. 5554. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 51–65 (cit. on p. 30).

Sears, Russell, Catharine Van Ingen, and Jim Gray (2006). *To BLOB or not to BLOB: Large object storage in a database or a filesystem?* Technical Report MSR-TR-2006-45. Microsoft Reseach.

Seltzer, Margo I. and Nicholas Murphy (2009). "Hierarchical File Systems Are Dead." In: *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*. Monte Verita, Switzerland (cit. on pp. 18, 30).

Shawn Wildermuth (2004). *A Developer's Perspective on WinFS: Part 1*.

Smith, Keith A. and Margo I. Seltzer (1997). “File system aging—increasing the relevance of file system benchmarks.” In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 25, pp. 203–213.

Spillane, Richard P., Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright (2009). “Enabling Transactional File Access via Lightweight Kernel Extensions.” In: *FAST*. Vol. 9, pp. 29–42.

Subramanian, Sriram, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jeffrey F. Naughton (2010). “Impact of Disk Corruption on Open-Source DBMS.” In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 509–520 (cit. on p. 27).

Sweeney, Adam (1993). “xFS Transaction Mechanism.” In: *Silicon Graphics, Oct* 7, p. 18 (cit. on pp. 24, 29).

Tamma, Krishna Pradeep and Shreepadma Venugopalan (2014). “Failure Analysis of SGI XFS File System.” In: (cit. on p. 23).

Traeger, Avishay, Erez Zadok, Nikolai Joukov, and Charles P. Wright (2008). “A nine year study of file system and storage benchmarking.” In: *ACM Transactions on Storage (TOS)* 4.2, p. 5.

Traiger, Irving L., Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay (1982). “Transactions and consistency in distributed database systems.” In: *ACM Transactions on Database Systems (TODS)* 7.3, pp. 323–342 (cit. on p. 21).

Venu Vasudevan and Paul Pazandak (1997). *Semantic File Systems*.

Vokorokos, Liberios, Anton Baláz, Branislav Madoš, and Ján Radušovsk (2013). “Cloud File System.” In: *Intelligent Engineering Systems (INES), 2013 IEEE 17th International Conference on*. IEEE, pp. 311–315 (cit. on p. 29).

Wright, Charles P., Richard Spillane, Gopalan Sivathanu, and Erez Zadok (2007). “Extending ACID semantics to the file system.” In: *ACM Transactions on Storage* 3.2, 4–es. DOI: 10.1145/1242520.1242521 (cit. on pp. 30, 61, 76).

You, Jin, Di Mu, Hongda Ma, and Boon Thau Loo (2013). “PennFS: A File System on Relational Database.” In: (cit. on p. 30).

Zhang, Yupu, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau (2014). “ViewBox: integrating local file systems with cloud storage services.” In: *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, pp. 119–132 (cit. on p. 29).