Designing and Implementing a Dynamic Metadata-backed Filesystem

Ian McEwen
chair: Chris Perry
member: Jaime Dávila

April 26, 2012

Contents

Chapter 1

Rationale and Philosophy

1.1 Background

When I use my computer, it could be said I'm a power user of the filesystem. I use version control, symbolic links (usually called symlinks), I copy over networks using FTP and SCP, I've used and set up network filesystems, etc. Despite this proficiency, and despite intensive research (chapter 2), there are some things that I still can't do with a filesystem. Most importantly, perhaps, filesystems tend to be very cumbersome with any large collections – I've personally run into this problem with my collections for both music and videos, and I've never become a photography buff in large part because the idea of managing photographs, even with the wide spread of tools I am proficient in, scares me.

So: why? Because filesystems, with all their strengths, and with all the ways I know how to use them, don't effectively allow me to exploit one sort of data: metadata. Metadata is *data about data* – who's performing a song, what TV show a video is from, where a photo was taken. None of these things (nor countless other things) is part of the main data – they aren't part of the audio, video, or photo data (nor the filesystem metadata, which includes filesize, modification timestamps, access permissions and other such bookkeeping information (Wikipedia Contributors, 2012b)), but rather information *about* that data. What makes metadata really important is that it is how *people* think about data: when looking to watch some TV, no person goes looking for a specific MPEG sequence – instead, they'd look for a certain TV show, or movie, or actor.

Now, you may protest: "But I find my files now! I start all my filenames with the TV show name, an underscore, and then the episode number!" This is, of course, well and good, but not outstandingly flexible. Without giving it more information, a machine probably can't figure out what this system *means*. It's also easy to misencode metadata into the filename – perhaps by accidentally putting the episode number before the TV show name, where the data is perfectly correct but has in-

correct *structure*. This is because filesystem paths aren't really designed to be a place to store metadata – they've ended up as such, because they're supposed to be a way to find information and metadata is how we as humans find information.[1]

Clearly, though, there's room here for improvement. Here's what I want: to be able to use the same programs I do now and benefit from a better way of storing, editing, and retrieving metadata. But I don't want to write plugins or patch those programs to understand this new system, because I'd have to do it for every program I use, and that would be a huge task and a maintenance nightmare. Taking all of this into account, I come to this conclusion: whatever I use, I want it to *look and act like a filesystem*, because that's what my programs already understand.

As for "a better way of storing, editing, and retrieving metadata", I'd like something that aligns pretty closely with the way people think and flexible enough for ad-hoc usage, but computationally feasible to deal with too – I should be able to extract practically anything I want from this data without human processing (except to declare preferences, of course). Another set of people solving a different problem have a similar set of requirements – those working on creating the *Semantic Web* – extending the World Wide Web to include machine-readable semantic data, rather than the purely human-readable information it primarily has now. A similar approach is probably in order; the Semantic Web uses several core concepts to represent information in a way that's both computer-understandable and (fairly) human-understandable: identifiers, triples, and ontologies (and similar things to ontologies like vocabularies and taxonomies – I won't distinguish here, because it's not important for this usage). Triples are the core of the system – they represent a single fact or piece of information in terms of a *subject*, *object*, and *predicate*. You probably heard these terms in your English classes, and they're used much the same – a sentence "The girl went to the store" conceptually becomes a triple of "The girl" as subject, "went to" as predicate, and "the store" as object. The girl and the store have a relationship of one having gone to the other. So the computer can understand and tell them apart too, I can assign unique identifiers to subjects, predicates, and objects. Ontologies are shared repositories of pre-defined identifiers, usually primarily predicates and special subjects and objects for things like types (after all: "He is a person", a statement of type, is just a triple of "He", "is", and "a person"). Put together a whole bunch of triples, and you can describe anything, and since you can always define a new ontology it's easy to be infinitely extensible – and it's pretty close to how people think, too, if sometimes a bit stilted.

Let's consider: filesystems aren't quite what I want to use to *store* metadata, but since I use them for that now (and so does every program I use), I'd love to *access* metadata, and the data it describes, with a filesystem. Using the concept of the triple, I can imagine a better way to represent metadata in some sort of

[1] Of course, more-complex systems don't solve all the potential problems – if you mispell "Psych" as "Psycho", it doesn't matter where you store it, unless you have some sort of typo-correction system in place on top of it.

data store.

1.2 Scenario

There's a number of different ways to approach this problem. Therefore, I'll first specify in detail exactly how I'd like this system to work for a specific case: disambiguation of two bands with the same name.[2] There are at least two bands named Perfume in my music collection; one is from Japan, another from the United Kingdom. To find songs by only the Japanese band, I'd like to be able to execute the following series of commands and see the provided results. In the footnotes, I'll discuss the *meaning* of these commands and return values as part of this system.[3]

[2]This also chooses a problem domain – but I could as easily have chosen a similar problem for movies, research papers, photos, etc.

[3]This scenario was conceived in a sort of technical vacuum, assuming only basic filesystem commands exist, and assuming nothing about the underlying technical infrastructure – in chapter 3 I'll discuss some small changes that needed to be made in implementation.

```
user@host:~$ cd gmm-root [4]
user@host:~/gmm-root$ ls
Album    Artist    Disc Number    _files    Title    Track Number    Type[5]
user@host:~/gmm-root$ cd Album
user@host:~/gmm-root/Album$ ls
Artist    ASIN    Catalog Number    Country    Date    Format    Label
Language    Media    Original Date    Script    Status    Title    Total
Discs    Total Tracks    Track    Type[6]
user@host:~/gmm-root/Album$ cd Artist
user@host:~/gmm-root/Album/Artist$ ls
Album    Country    Credited Name    Name    Sort Name    Track    Type[7]
user@host:~/gmm-root/Album/Artist$ cd Name
user@host:~/gmm-root/Album/Artist/Name$ ls
3 Hürel    4minute    9ice    Bon Iver    Daft Punk    Every Little Thing
Jätkäjätkät    OSI    Perfume    The Pains of Being Pure at Heart    モーニング娘。[8]
user@host:~/gmm-root/Album/Artist/Name/$ cd Perfume
user@host:~/gmm-root/Album/Artist/Name//Perfume$ ls
Album    Artist    Disc Number    _files    Title    Track Number    Type[9]
user@host:~/gmm-root/Album/Artist/Name/Perfume$ cd Album/Artist/Country/JP[10]
user@host:~/gmm-root/Album/Artist/Name/Perfume/Album/Artist/Country/
JP$ cd _files[11]
```

[4]Or to whichever directory is the root directory for the system.

[5]These folders describe predicates linked directly to file objects. Here, I assume that the system only includes music files; in cases where this isn't the case, many predicates unrelated to music would appear, and the 'Type' predicate could prove much more useful.

[6]These folders describe predicates linked to Album objects, roughly; specifically, this list is of predicates which apply to objects linked to files by an Album predicate.

[7]These folders describe predicates linked to Artist objects (specifically, those Artists linked to Albums linked to files.

[8]These folders (and many more like them) represent actual values for the Name predicate. The system knows these are actual literals, and so choosing one conceptually 'finishes' a filter.

[9]Having completed a filter, the view should be much like the one from the root directory (with any predicates that are no longer applicable removed – for example, it's likely if there were videos in this same system, this time around a 'Director' predicate would not be shown).

[10]This command will add another filter, this time for the album-artist's country being set to 'JP'. One folder above this, there should only two folders: JP and GB – one for the Japanese band and one for the British band named Perfume. Useless folders shouldn't be shown.

[11]This tells the system I're done filtering – of course, a subsequent `cd ..` would undo this. This somewhat-inconvenient `_files` directory exists so that the number of files in directories are not overwhelmingly large – for example, if this were to include files alongside predicates in the root folder, in my music collection that folder would have more than ten thousand entries!

```
user@host:~/gmm-root/Album/Artist/Name/Perfume/Album/Artist/Country/
JP/_files$ ls
01_Perfume_GAME_1-01_ポリリズム.flac    […]
12_Perfume_GAME_1-12_Puppy love.flac
13_Perfume_JPN_1-01_The Opening.flac    […]
26_Perfume_JPN_1-14_スパイス.flac
27_Perfume_Perfume ～Complete Best～_1-01_パーフェクトスター・パーフェクトスタイル.flac
[…] 38_Perfume_Perfume ～Complete Best～_1-12_wonder2.flac[12]
```

To recap, I'm saying here that (relative to the base folder), the folder `./Album/Artist/Name/Perfume/Album/Artist/Country/JP/_files` shows songs from albums whose artists are named Perfume and are from Japan, listed in terms of a unique integer and a configured hopefully-unique filename. Thus, I'd find `./Album/Artist/Name/Perfume/Album/Artist/Country/JP/_files/16_Perfume_JPN_1_04_ナチュラルに恋して.flac`, open it in my music player, and rock out!

With what's already been discussed, then, how close am I to solving this problem? It seems like something like symbolic links might be useful here, since the same file could appear lots of different places. But, I certainly don't want to manage them manually – that would be way too many things to keep in-sync, especially since I want to be able to have multiple filters (like the two used above). So, it seems I need to dig a bit further to find a tool that fills this need.

[12] These are the files matched by the constructed query. They're usable files, with names in terms of a configured hopefully-unique filename, plus a prepended integer to ensure uniqueness. Here I've asserted that the unique names configured in the data-store follow the format Artist_Album_Disc_Track_Title.extension. Now I can use this collection of files as I see fit!

Chapter 2

Prior Art

Perhaps there already exists a tool to connect my desired interface (the filesystem) to a more-effective metadata system. Certainly, I'm not the only person who's thought of this sort of thing: in many ways, as I've already said, filesystems themselves were an attempt at solving this problem. More to the point, many other partial solutions to this sort of problem have been presented and implemented over the years. None is quite what *I* want, but their investigation will be informative.

To start, filesystems. Filesystems don't explicitly store metadata at all (except for their own bookkeeping metadata, as previously mentioned), but most people use filenames to store metadata. Filesystems operate on a purely hierarchical model,[1] which is often sufficient, but sometimes doesn't match human ways of organizing and understanding data. For example, sometimes I'd like to find a specific song by title (ignoring, or perhaps not even knowing, its album or artist), but other times I'd like to find a full album or everything by a given artist – a hierarchy has to put one or the other thing first, so it can't efficiently cater to all these ways of looking for music. Filesystems also require a lot of manual or third-party programmatic management: files don't move themselves, which makes the same thing always be in the same place. This is a great feature in general, but it's frustrating if it means I can't find something. (Wikipedia Contributors, 2012b)

It seems I could possibly solve my problem with a simple script that creates symbolic links – no need for anything more complicated than a data store and the good old filesystem! However, there's a major problem: recursion. Because I want to be able to apply arbitrarily many filters to my data, that script would have to figure out every possible combination ahead of time, which will take a long time unless I limit it to a certain depth

[1] For the most part, anyway – some advanced features like links allow for some slight non-hierarchical functionality

or structure my data specifically to avoid complication. I don't want these limitations!

Another conceptually faraway system that inspires without solving the problem is git-annex. Unlike the filesystem and most other tools in this section, git-annex is not designed to deal with the metadata problem; in fact, it's designed to solve a technical limitation of a *version control* system. It gets around a performance limitation of git by putting symbolic links into the version control system, which point to a separately-maintained tree of files.

A side-effect of this construction is that multiple places in the tree can point to the same data. This makes it easy to understand the same file along multiple interpretations – for example, my videos could be interpreted both by type (TV, Movie, etc.) and in terms of how I got them: I can maintain a directory tree in one place with folders for TV vs. Movies, and another tree elsewhere splitting by DVD vs. digital. By using symbolic links, I can see these same files in two places, one for each interpretation, without duplicating the actual data. Many such examples of multiple possible interpretations exist: photos in terms of their subject, provenance, or color balance; or research papers in terms of their authorship, publication, subject, relationships with personal projects, or relationships with other research papers. Supporting this sort of deduplication is quite in line with my personal understanding of files and their metadata. (Hess, 2011)

Unfortunately, git-annex provides no tools for metadata specifically – its improved suitability for this task is an accident of its design.

Moving approximately chronologically from filesystems, some of the first tools to help solve the metadata problem were probably basic music and video players and photo management applications. Standards like ID3 emerged to store embedded metadata, which music players took advantage of to display useful metadata in a consistent format to their users. (O'Neill, 2006) Some movie formats (for example Matroska (Matroska, 2012)) also support embedded metadata, which a few programs such as VLC can use. Some picture formats also have limited embedded metadata by way of EXIF, which some programs and services such as Flickr know how to use. (Wikipedia Contributors, 2012a) Some more recent programs also maintain databases of metadata for their own use, both to move beyond the constraints of embedded systems (such as the hardcoded list of genres in ID3v1 (O'Neill, 2006)) and to put metadata into the mix where no embedded format exists. These solutions are all very context-specific, though, concerning only music, only movies, only photos, etc., and are usually not designed to be terribly extensible or interoperable. Embedded metadata usually has a constrained format limiting extensibility (although it is very interoperable); program-specific databases are sometimes built very generally but are very rarely accessible outside of that program. Theoretically, a sufficiently general embedded scheme could solve my needs, but it would require coordinating the developers of literally thousands of different formats and getting them to agree, a task I would not assign to anyone less than superhuman.

It's worth noting a few standardization efforts (of a limited sort) that already happen. One such effort is by way of programs used for writing embedded metadata; usually such programs have a standard way of writing the same piece of data across several different formats. MusicBrainz Picard, a music-tagging program that connects to the open MusicBrainz database of music metadata as its primary data source, is a good example of such a program; in order to support major music formats it must have many ways of writing the same piece of information (e.g. licensing information to WCOP in ID3, LICENSE in Vorbis-derived formats, and somewhere else for AAC). (MusicBrainz Contributors, 2012) As a cost for this standardization, though, most such programs (Picard included) limit the flexibility and extensibility of even the most general embedded formats. By writing to embedded metadata, these programs are subject to the whims of other applications that may use that metadata in unusual ways, and to make up for this must often limit their users.

At this point, moving to more general applications, are tags. Popularized by tools on the web such as Flickr, blog software, etc., tags are simple associations between objects and chunks of text (ideally, descriptive ones). Tags are highly interoperable and very flexible (and very easy to use!) but they have the weakness of little semantic content – it's very hard to tell, without knowing beforehand, whether a tag "Rubinstein" refers to a performer or a composer. Tags are somewhat *aggressively* nonhierarchical – which doesn't mesh well with my understanding of the world around me.

Another attempt at solving this problem manifested in terms of so-called "collection managers" such as Alexandria and GCstar. These programs usually include some predetermined types of records (e.g. album, book, etc.) and perhaps an option for custom record types. Then, collections can be built of records of various sorts. The weaknesses of such a system are, once again, interoperability and flexibility. While more flexible than highly standardized embedded metadata, rigid record types aren't conducive to ad-hoc information that may only apply to a subset of items (constantly begging the question: "Is it worth adding a field to the record type to support this, or should I just ignore it?"); moreover default record types can't account for pieces of metadata that are important to only some individuals. Interoperability-wise, these systems rarely communicate outside their own walls, making reuse of this metadata very difficult even for programs and users that want to. While their goals of curating collections of anything led to some welcome changes like customizable record types, collection managers don't scratch my itch. (Jodar, 2010)

The Semantic Web effort's notion of triples, discussed earlier, can in some ways be seen as a way of extending the notion of tags to three things rather than two (plus a bit more flexibility about what each of the things can look like). The format developed by the W3C for expressing and conveying triples is RDF (standing for Resource Description Framework), which uses URIs (Uniform Resource Identifiers – URLs are

one kind of URI) as identifiers and (by default) XML as a format for serialization. RDF hasn't taken off for its intended purpose as a sort of decentralized knowledge-base on the World Wide Web (not yet, anyway), but a number of ontologies have been developed for use with it and as a data format it's highly expressive. RDF alone, of course, is just a data format: using it for metadata will take more logic around it. Nonetheless, RDF, being tailor-made for much the same data needs as mine, is almost certainly my choice for storing data.[2] Conveniently, most existing RDF stores (conventionally called *triple stores*) support a standardized query language for RDF called SPARQL. (W3C, 2008, 2004)

Another related tool is the *virtual file system*: an abstraction on top of a more concrete filesystem, designed to allow applications to access and manipulate different types of concrete filesystems or anything that can be made to look like a filesystem in a uniform way. Virtual file systems are nice because they create a filesystem-like interface without requiring implementers to actually worry about the details of storing files on disk. Of course, virtual file systems are a general tool, not a specific one – I can use them to implement the system described in my scenario, but alone they don't solve my problem. Unfortunately, nobody yet has made a virtual file system that interfaces with RDF (or any other structured data) to provide the interface I want, or one similar to it.

Some other applications even went as far as using RDF. The NEPOMUK project (for creating a "Social Semantic Desktop") and libferris are several programs that use RDF on the desktop. NEPOMUK aims to be integrated, and NEPOMUK-inspired implementations are part of both GNOME and KDE, the major Linux desktop environments. Sadly, NEPOMUK goes so far as to define its own representational language on top of RDF, limiting interoperability and flexibility (since it's that much harder to use existing ontologies with it). The existing implementations are also both focused very much on programmer use more than direct use, and we don't want to have to write plugins and patches in order to benefit from good metadata. Both implementations of NEPOMUK are also closely integrated with desktop environments I don't use, which while personal means that I'm not keen on installing everything I need to make them work. (GNOME Developers, 2011) (KDE Developers, 2011)

libferris is probably the closest anyone's gotten to what I want – it uses RDF, stores complex metadata, and acts like a filesystem. However, its focus is a bit different – it's not looking to construct filesystems from metadata at all. It adds much better metadata associated with files, but it still requires changing programs to be able to see this meta-

[2]A potentially valid complaint at this point, especially for those who know the Semantic Web and RDF already: "Wouldn't this be hopelessly confusing for a normal user?" Yes, perhaps. But I'm not designing for a normal user – I'm designing for me, and assuming (probably correctly) that there are others who have the same need and who would benefit from the same product. Ideally, this would build into a larger community effort – but if it doesn't, it should still be useful for me and the few people who want things the same way I want them.

[3]Or, at the command-line, using libferris's tools such as 'ferrisls' – but the combine-and-reuse philosophy

data.[3] libferris also makes it possible to mount a number of other things as fake filesystems, such as XML documents (including many office document formats), relational databases, and a number of other intriguing things. It does support mounting as virtual file systems queries against its metadata (or indexes upon its metadata), but it doesn't support hierarchical results, so any such search ends up being one giant directory full of the results. We're looking for something more persistent, and in much more manageable chunks, a prospect which requires hierarchy – most people wouldn't want twenty thousand music files to be in the exact same directory.[4] (Martin, 2010)

Though there's been much innovation on the topic, no existing tool truly fills my stated need. In order to have a filesystem-based access to files, structured hierarchically in terms of their associated metadata, I'll need to create it. Luckily, some tools will help: RDF and SPARQL are a perfect data store, and virtual file systems provide the abstraction I need to avoid reimplementing a whole filesystem.

---

of the shell makes this much less problematic, since you could just alias 'ls' to 'ferrisls'.

[4]In addition, libferris has a gigantic tree of dependencies, most of which aren't available outside of its author's Linux distribution of choice, which has regrettably prevented my ever successfully installing libferris on my own system.

Chapter 3

Implementation

3.1 Preliminaries

First, I need to choose among the options for virtual file systems. Among the best-known (and best-supported) virtual file system toolkits is the FUSE (Filesystem in Userspace) library. On my chosen platform of Linux, other virtual file system options are generally tied closely with desktop environments, as were (undesirably) NEPOMUK implementations (discussed earlier, p. 14). Therefore, FUSE is my virtual file system toolkit of choice. FUSE also has a Mac OS X equivalent, MacFUSE – but unfortunately it appears unmaintained at present. Should it be ressurected, the choice of FUSE might lend some additional platform-independence. As another bonus, FUSE's API has bindings in many commonly-used programming languages; this language flexibility eases my implementation task!

I also need to choose a data backend. As discussed earlier (p. 13), my chosen data model is RDF, and its data stores are called *triple stores*. There's no good reason for me to implement my own triple store – efficient storage of RDF is a complicated problem well beyond the scope of this project, and a number of existing projects provide very competent triple stores and support for the standard SPARQL query language. The backend code should be flexible enough to allow any SPARQL-compatible store to be used, but for proof-of-concept I'll choose just one. Out of convenience of installation on my system, I've chosen to use Garlik 4store.

Having chosen the "ends" of the system, I must choose a programming language to use in binding them together; a wide variety of languages have FUSE bindings, and nearly as many have libraries for using the SPARQL protocol. Additionally, for test data, I'll need something to generate RDF data – but this need not use the same toolchain as the main system. For the latter, I've chosen to use Python (and the excellent RDFlib library) due to my familiarity with the language; since this script will be run only once performance is not a concern. For the former, I've cho-

sen to use Haskell: it has bindings to both FUSE and a SPARQL protocol library; its functional structure plays very nicely with the recursive nature of the required queries as well as with the statelessness of filesystem calls; and to boot, I just plain *like* it.

Next, I'll make some basic assertions about the structure of the data – specifically, I need three predicates, one for pointing at the actual filesystem location of data, one for assigning nice-names to predicates, and one for assigning a hopefully-unique display name to a file. I'll call these, respectively, `gmm:fileContent`, `gmm:niceName`, and `gmm:filename`. The 'gmm' namespace prefix here will refer to `http://ianmcorvidae.net/gmm-ns/`, but this choice is quite arbitrary and a system beyond a prototype would undoubtedly put much more effort into this mini-ontology. The data will be structured to separate the file itself (identified by a `file://` URI) from a metadata object – hence the need for the `gmm:fileContent` predicate, linking the latter to the former. Otherwise, if two files were the same thing (say, in two different formats), I'd have to duplicate the metadata.

From here, I'll consider test data. Consistent with my scenario (p. 7), I'll define a small ontology for music with the namespace prefix 'ms', referring to `http://ianmcorvidae.net/ms-ns/`. I'll create data following three schemes: the scheme my scenario assumes, with a full graph and dedicated resources for artists, albums, and labels; but additionally two other schemes, which I'll call 'simple' and 'flat', to use as a basis for comparison (the scenario format I'll refer to as the 'deep' scheme). The 'simple' and 'flat' schemes are similar, both using only direct links from the file metadata object to literals.[1]. For more details on the creation of this test data, please see the code for `rdfgen.py` in the appendix.

Finally, I must add one bit to the scenario, for the sake of performance and ease of implementation: in order for the system to know where subsequent filters begin and end, and thus avoid having to consider every path part as potentially both a predicate name *and* as a literal value, I'll add an equals sign: between the last predicate name and the actual literal name, I'll add a folder called `_=` (following the same naming scheme as `_files`), by which the system can split up a pathname into filters and identify which folders are predicates and which are literal values.

3.2 Core Implementation

The meat of the project, of course, is the "Haskell part" – the FUSE filesystem and query system. I'll consider this portion of the project in terms of the FUSE interface – that is, in terms of implementation of specific filesystem calls. I'll begin with `readdir()`, the function that provides the listing of contents of a directory, since this is the most complicated of the implementations, and then touch on the other interesting implementations: `opendir()`, `readlink()`, and

[1]This is rather than dedicated additional objects – so, for example, the album-artist for a song would be linked by `ms:albumartist` rather than by a `ms:name` link to an artist linked to an album linked to the song

`stat()`.[2]

First of all, I'll discuss `readdir()`, which returns a list of filenames paired with `stat()`-style structs for each entry.[3] The primary consideration here is how to translate a pathname into a SPARQL query that returns directory entry filenames. To explore how to do this, let's consider the structure of a pathname within this system: a given pathname could be broken up into a number of independent filters. These filters have three possible forms: a complete filter, such as `Artist/Album/Name/_=/Perfume`; a half-complete filter[4], such as `Artist/Album/Name/_=`; or an incomplete filter, such as `Artist/Album` (or the empty filter). Each of these types is treated primarily the same, with some variations. Optionally, the end of a pathname might be `_files`, so long as all the filters before that are complete filters.

In addition to a number of filters, every query will include basic structure: a link to a file resource, a shared metadata object, and duplicate elimination. Therefore, the required queries include this basic structure plus recursively-built sections for each filter. HSparql, the Haskell SPARQL protocol library, provides a domain-specific language[5] for constructing queries that makes this very straightforward: run a function to set up variables and add the basic structure, then use a fold[6] to process each path and add to the query. We also have to do some extra processing in the case of the `_files` subfolder, but otherwise this is the basic structure of each query.

What remains, however, is handling each filter. This, too, is recursive: for each part of the path before `_=`, the program should expand the search to another level away from the metadata object (another "hop"), based on the predicate nice-name provided. This is conducive to another fold: provide a list of nice-names, start with the metadata object, and process one name with each iteration. At the lowest level, the function used by this inner fold, the program processes to distinguish the three cases mentioned earlier. In the case of an incomplete filter, the program should add the *structure* of another hop from the metadata object, but instead of declaring what the relevant predicate nice-name should be, designate that as a variable to be returned; by doing so, the query will return a list of possible predicate nice-names for this next hop. For a half-complete filter, the program should designate the last object returned by the fold to be returned, since this will be the literal value eventually reached by the "path" designated by the filter. For a complete filter, the pro-

[2]I've also implemented `statfs()`, but it's trivial and pretty boring. For details, see the code for gmmFuse.hs in the appendix.

[3]However, I use dummy structs rather than full `stat()` return values in this implementation, since it is a prototype.

[4]That is, complete up to the equals sign

[5]This DSL is based on the State monad, for the curious Haskell programmer

[6]A fold is a functional-programming tool which, given a start value, a list, and a function, runs the function with the start value and the first element of the list, and then recurses with the remainder of the list and the return value of the first call, eventually returning the return value of the final function call

gram should declare that the final hop's value should be equal to that which follows `_=` in the filter. A complete filter need not return anything, because it is always followed by an incomplete or half-complete filter (if only the empty filter) – but it'll actually return the unused 'name' variable that it was passed, in order to make implementing the fold easier.

Now I have a way to build up queries for directories. To modify this process in order to create a listing for the `_files` directory requires only adding one additional bit of basic structure, to extract the hopefully-unique filename from the RDF store, and then return this plus the file data location – there won't be any other returned values, because the `_files` directory can only be used after complete paths. I also add ordering by the stored filename and then the data location, so I can be assured results will always appear in the same order. In the code (after the query), I'll additionally prepend integer indices in order to ensure these listings are unique.

Finally, to implement `readdir()` itself, I add `_files` or `_=` as appropriate: the former when the last filter is complete, the latter when it's incomplete, and neither when it's half-complete.

Each of `opendir()`, `readlink()`, and `stat()` is fairly straightforward given the query infrastructure for `readdir()`. For `opendir()`, which returns a code (either `EOK` or `ENOENT`, the latter for "this doesn't exist"), the program can simply strip off the last pathname part (in order to find the parent folder), run the `readdir()` query on that path, and return `EOK` if it's in the directory listing for its parent. For `stat()`, which returns either an error code (`ENOENT` once again, generally) or a structure defining a file status such as permissions and size, the program can similarly check for the existence of the file or directory, constructing a stat object according to its existence and whether it is a symbolic link (i.e., is immediately within the `_files` directory) or a directory, doing a second query if it's a directory in order to determine the number of links to the directory.[7] Finally, for `readlink()`, which returns either the friendly `ENOENT` or a string with the target of the link, the `readdir()` query for the `_files` directory can check both the existence of a given link and provide the relevant link target to return. There are additionally a number of other FUSE functions that I could have implemented (such as `open()` and `read()`, for regular files), but they are unnecessary for this system.

The code for gmmFuse.hs is available in the appendix.

3.3 Additional Notes

During the process of implementation, I also ran into some sideline problems that are worth mentioning here.

Earlier, I mentioned that the statelessness of filesystem calls was a good fit for Haskell. Unfortunately, it's not a great fit for a database. A simple `ls` of a directory

[7] This is the `st_nlink` value included in the struct returned from `stat()`. It's displayed by `ls -l`, but most users probably aren't familiar with it. Generally, it's equal to 1 for a regular file, and 2 plus the number of subdirectories for a directory.

must: open the directory, read the directory, and stat all the files. By the implementations above, this results in N+2 SPARQL queries: one for each of the `open()`/`readdir()`, and one each per `stat()`. Most of them are the same: `stat()` calls the same query as the `readdir()` on the containing directory. Unfortunately, each of these queries can run for quite some time, meaning that even a very basic command can take a long time to return. But most calls are shared, so this seems like a good target for caching. It is – after I set up nginx as a caching reverse proxy on top of my SPARQL endpoint,[8] performance was quite reasonable – any given query only gets run once. Of course, caching becomes more complicated when data is updated or modified, as I'll discuss in section 5.1.3 below.

I've also added several incorrect but better-performing implementation variants of two filesystem calls, `stat()` and `opendir()`. `stat()` has two levels of optimized variants; its first-level optimization won't return proper `st_nlink` values (removing one of two queries), and the second-level optimization will additionally not check for existence of directories (removing the remaining query). `opendir()` has one optimized variant, which simply doesn't check that directories exist, therefore always returning `EOK` (and removing the single query this command makes). Limiting the number of queries is one of the best ways to improve performance of this system, as discussed below in chapter 4.

These version work fine in practice, but aren't technically correct, and could cause problems in some edge-case situations.[9] As I'll discuss in chapter 4, the completely-correct version is very slow, but even just the first-level `stat()` optimization improves performance to the point of reason. These functions are named, in gmmFuse.hs, with appended single-quotes (such as `gmmGetFileStat''`).

I also ran into a bug in 4store, my triplestore of choice. Certain queries (so far as I could tell, those where multiple bindings would be matched to the same resources – so, anything within this system where two filters had a shared prefix, including the case described in the scenario (p. 7)) would cause the SPARQL part of 4store, 4s-httpd, to throw a segmentation fault and return no results. Ultimately, while trying to debug the problem, I determined that running 4s-httpd inside valgrind, ironically a memory debugger, stopped the problem.

Finally, there's some handling for escaping slashes, which can appear in pathnames, but when they do can only be interpreted as path separators. I use a simple escape system, using the `#` character as an escape. `##` is interpreted as a single `#`, and `#%` is interpreted as a slash.

[8] SPARQL operates over HTTP, thus an HTTP reverse-proxy like nginx can provide this caching layer.

[9] For example, they won't result in correct `ls -l` listings, since this command shows the `st_nlink` values.

Chapter 4

Evaluation

4.1 Methodology

In order to evaluate this prototype, I'll take a tour around the system, describing both how it feels, as a user interacting with the system, and backing up these perceptions with numerical analysis.

Let's consider how to numerically analyze the performance of the system. Many tools are provided for this task: simple timing of commands at the command-line, log files for 4store and nginx, and some tools provided by Haskell for profiling time and space usage. I used a combination of all of these.

The 4store query log contains a time, the query, execution time, and number of rows returned. The nginx log I've configured to show time, HTTP request parameters (in which the query is embedded), return code, bytes returned, time, and upstream time (that is, the overall time in 4store), plus cache status (hits vs. misses being the values of interest).

First, I developed a test suite of commands to run, consisting of `ls` and `cd` commands as well as a few `metaflac` calls in order to test the `readlink()` implementation. I then ran each of these commands prefixed with `time`, to test the timing of these calls. I correlated each command with which lines it produces in both the 4store query log and the nginx log. Since this testing is for performance rather than correctness, I assumed that any successful return was correct. I additionally ran this test with the incorrect implementations noted in section 3.3, for comparison. Finally, I repeated everything with different collections of Haskell profiling flags, for time and space analysis of the system. Since it is possible to calculate the time saved by caching from the combination of the nginx and 4store query logs, I did not run at any point without the reverse proxy cache, though I cleared the cache between runs of the test suite.

To test different functionality, the test suite contains commands relating to a variety of potential use cases. First, the commands listed in the scenario (section 1.2), since this is my primary implementation tar-

get. In order to test `readlink()`, the suite has a call to `metaflac --export-tags-to /dev/null` on the first result at the end of the scenario. In order to test some other uses of the system, I add a few other paths and commands: `Album/Date/_=/2002*/`, plus (relative to that) `_files`, a `metaflac --export-tags-to /dev/null` on the first `_files` result, and `Album/Artist/Name/_=/`.[1] These paths test the effect of using shell globbing, a way of emulating logical OR (otherwise unsupported by this system).

This test suite, being very reliant upon the scenario, can't be run exactly on data following the 'simple' or 'flat' schemes. I'll comment briefly in each section on how these schemes could change the results.

4.2 Results

The first perception upon using the system is that everything seems slightly slow. This is backed up by the timings shown in table 4.1 – depending on the directory and the optimization level, either `ls` or `cd` had latency of 1-30 seconds (though generally less than 15). The `stat()` optimizations have the effect, mainly, of moving the latency to `ls` rather than `cd`; this is due to when the query for a given directory is run and then cached.

The various optimizations have only small effects, time-wise, except for the last few commands using globbing; this is understandable, since commands using globbing will generate many more filesystem calls and thus will see aggregate benefits from the optimizations. Especially notable are the last and second-to-last commands, where second-level `stat()` optimizations clearly shine through, improving performance by as much as 8 times. While using the system outside of testing, the cache benefits seen here could be amplified, as queries are reused more in exploratory usage (through reuse of folders during traversal of the filesystem, and niceties such as tab-completion in shells). Since these observations are independent of data structure, the 'simple' and 'flat' schemes do not see dramatically improved performance (except as noted below).

These timings also correlate with the timings of queries: the 1-12 second timing values are only a few milliseconds longer than the queries these commands generate; subsequent calls to the same query are cached and take negligible time.[2] The commands with globbing, at the end, result in several queries, thus their much longer time. Since the alternate data schemes are less complex overall, query times are reduced in these settings, improving performance slightly.

[1] Conceptually, the first is a filter for only songs from albums released in 2002; `_files` is the file results of that filter. The last path will list the album-artist names for albums released in 2002.

[2] The nginx log shows timings of 0.001 seconds for cache returns!

Command	Time (s) by optimization level			
	[none]	opendir	stat(1)	stat(2)
`>> cd gmm_deep/`	5.152	5.127	0.000	0.000
`>> ls`	0.002	0.002	5.222	5.111
`>> cd Album`	12.990	12.966	0.001	0.000
`>> ls`	0.004	0.003	12.973	12.920
`>> cd Artist`	7.823	7.752	0.003	0.000
`>> ls`	0.005	0.003	7.752	7.751
`>> cd Name`	1.143	1.141	0.004	0.000
`>> ls`	0.003	0.002	1.127	1.153
`>> cd _=`	1.619	1.629	0.004	0.000
`>> ls`	0.037	0.039	1.613	1.612
`>> cd Perfume`	0.364	0.280	0.034	0.000
`>> ls`	0.033	0.003	0.238	0.249
`>> cd Album/Artist/Country/_=/JP`	1.705	1.687	0.946	0.000
`>> ls`	0.004	0.003	0.324	0.641
`>> cd _files`	0.550	0.542	0.039	0.000
`>> ls`	0.019	0.019	0.440	0.457
`>> metaflac --export-tags-to /dev/null 01*.flac`	0.055	0.054	0.044	0.042
`>> ls Album/Date/_=/2002*`	10.110	8.166	8.004	5.738
`>> ls Album/Date/_=/2002*/_files`	9.098	8.929	6.901	4.538
`>> metaflac --export-tags-to /dev/null Album/Date/_=/2002*/_files/01*.flac`	4.039	3.876	3.177	0.500
`>> ls Album/Date/_=/2002*/Album/Artist/Name/_=`	30.139	29.459	24.073	5.757

Table 4.1: Timing by command and optimization flags

Though the cache benefits are underrepresented by this test methodology, the cache performance during these tests was fantastic, as table 4.2 shows. Cache hit rates were all above 50%; though optimizations reduced the hit rate, they did so by reducing the total number of queries, thus saving time; in fact, the lower cache hit rate of the second-level `stat()` optimizations showcases the effectiveness of that optimization. Not shown in the table is the overall cache performance (across all test runs), which due to the much larger number of queries generated with no optimizations or only `opendir()` optimizations is strongly biased toward those numbers. Alternate data schemes show no significant changes in this regard.

Optimization Level	Hits	Misses	Hit Rate
none	1801	146	92.5%
opendir	1758	146	92.3%
stat(1)	684	122	84.9%
stat(2)	90	76	54.2%

Table 4.2: Cache performance

This brings up another point: the `opendir()` optimizations, though they improved the timings somewhat, don't seem to be very effective. This is because this optimization reduces only cache hits, and not cache misses as the `stat()` optimizations do. Without caching, it would still not be as effective as the other optimizations, but would cause much more improvement than in these tests.

The best demonstration of the optimizations, however, is in the heap-profiling information generated with the standard Haskell tools. Figures 4.1, 4.2, and 4.3 show heap-allocated memory over time, split up by "cost center" – generally, by which function allocated the memory.[3] These graphs show exactly *why* these optimizations are so much faster: the reductions in how "spiky" the graphs are as subsequent optimizations are applied is indicative of the smaller number of required queries – by coloration, these spikes are all from calls to the `query` function from HSparql. The second-level `stat()` optimizations are especially striking, with a very small number of spikes; these correlate to the most complex of the required queries from the test suite: the `readdir()` queries from the scenario, and those at the very end using globbing. Alternate data schemes, with their reduced complexity, show lesser memory usage overall, but in similar patterns.

I see two lessons from this evaluation: that queries are the main bottleneck, and the

[3] `opendir()` optimizations are excluded because they are not substantially different from the correct implementation in terms of memory usage.

main source of queries is calls to `stat()`. That queries are the primary bottleneck is evident from the heap profiles – querying the SPARQL endpoint is the largest use of memory within the system, and command timings are closely correlated with query timings. That `stat()` is the biggest source of queries is evident from the differences in memory usage and timings among the various optimizations: optimizing `opendir()` had almost no effect, but both levels of optimization of `stat()` noticeably improved performance. Since these optimizations were based on reducing the number of queries, it's clear that `stat()` generates the bulk of the system's queries, and thus is the biggest performance bottleneck.

The data used to generate these tables and graphs should be available alongside this document; they are not included in this document itself due to their size.

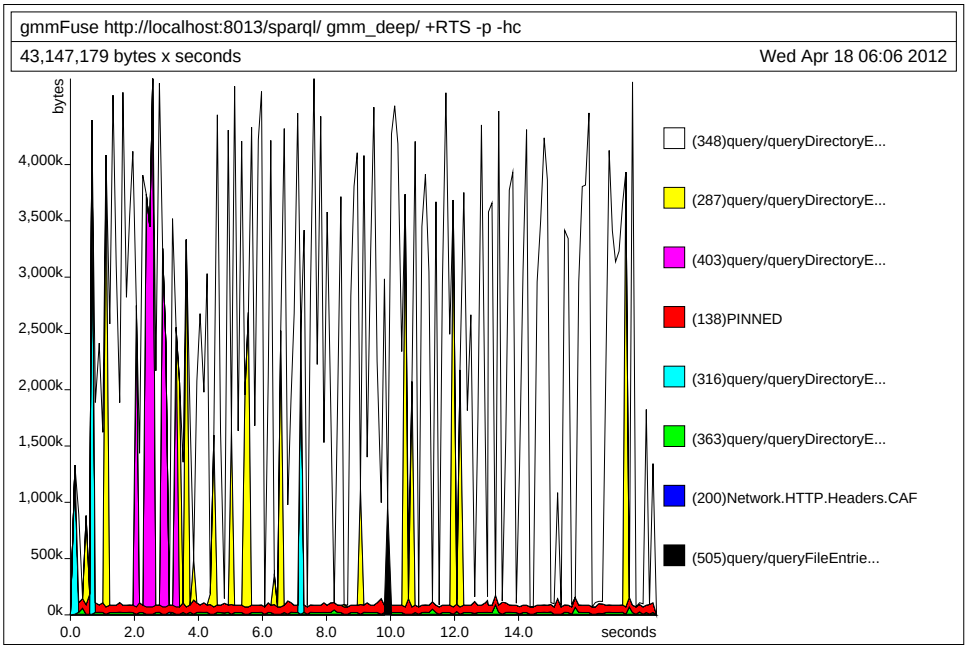

Figure 4.1: Heap usage by cost-center for the unoptimized implementation

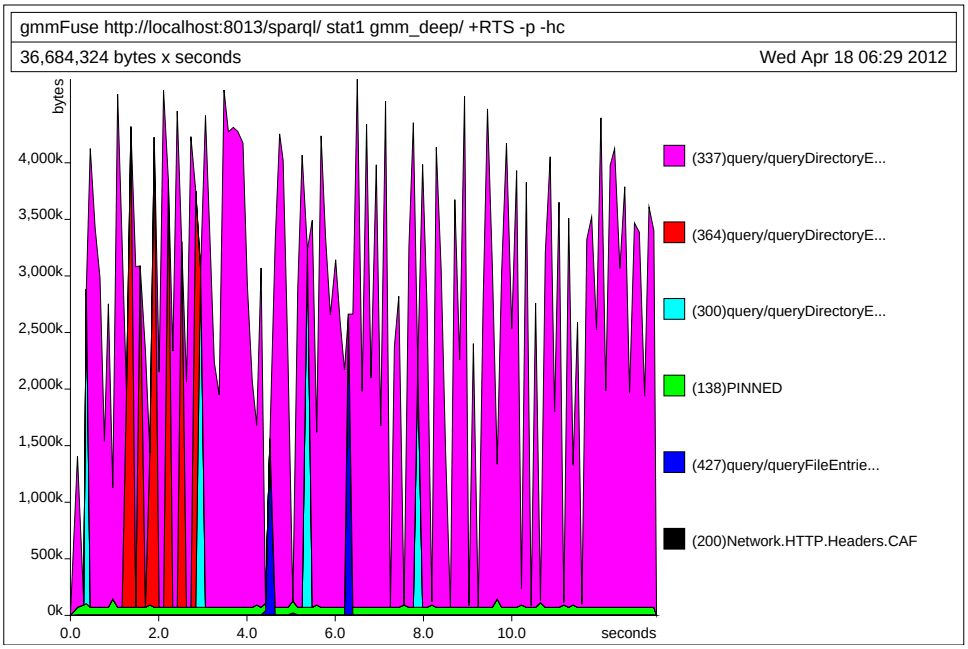

Figure 4.2: Heap usage by cost-center with first-level `stat()` optimizations

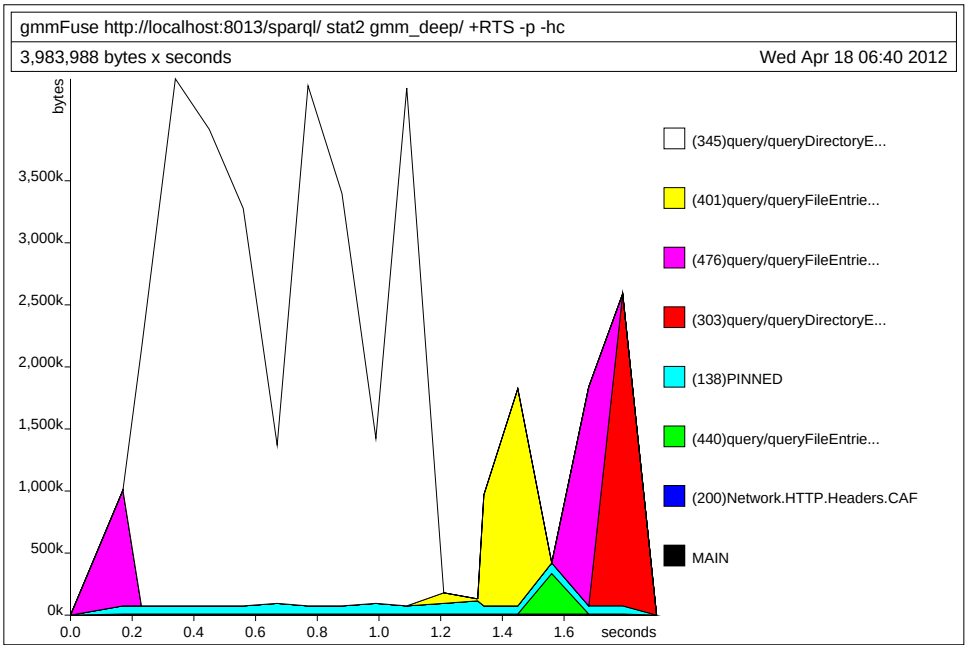

Figure 4.3: Heap usage by cost-center with second-level `stat()` optimizations

Chapter 5

Conclusions and Future Work

5.1 Future Work

This system, while a quite functional prototype, is by no means complete or without flaw. Here I'll present a variety of improvements and new features that would grow this system to fulfill more of its potential. Some of these changes move the system away from being simply a filesystem and toward a complete metadata system. I've called this a "generic metadata manager" before, and this dynamic filesystem project should be taken as a step along the way to this ultimate goal.

5.1.1 General improvements and performance

There are a variety of strategies that might improve upon the performance metrics from section 4. SPARQL querying is the clear bottleneck of the system (as the heap profile graphs demonstrated); therefore, these optimizations focus on reducing the number or complexity of queries.

Improving `stat()` As alluded-to in chapter 4, the `stat()` call is the largest performance bottleneck of the system; optimizing it had the best effects. This is twofold: it is called very frequently, and in the unoptimized, correct implementation, it also generated two queries per call. Collapsing this to one query is undoubtedly possible with some complication of the codebase. A potential disadvantage is the complexity of such a query, but some experimentation could bring good results. The implementation of `readdir()` also currently doesn't return accurate `stat()` structs as it should, so it is possible a more-complex unified query would see much more widespread use and thus benefit from caching.

Additionally, the existing incorrect `stat()` optimizations are harmless for most use, and could be recommended for most users of the system or turned on by default.

Better caching and materialization Though I'll discuss caching more in section

5.1.3, the caching this system uses is somewhat simplistic. It could probably benefit from caching data closer to the filesystem than it is, to avoid the overhead of SPARQL (and HTTP) and of parsing the responses from the query server. It could also automatically materialize more things in the triple store or an alternate data store, such as the number of subdirectories of a given query path (which could improve `stat()` performance on par with the first-level optimizations, but without the correctness detriments).

One potential caching-based reduction in query complexity seems especially low-hanging: caching of predicate nice-names. Unlike most other cache applications, invalidation is very easy for nice-names: invalidate whenever a triple with predicate `gmm:niceName` is inserted or updated. With this cached, the filesystem would be able to efficiently translate nice-names into predicate URIs and back, thus halving the number of triples needing declaration in the query chunk for a given filter. Some testing would nevertheless be valuable, to see if this query optimization actually results in quicker-running queries.

Sacrificing (further) correctness It's also possible I could sacrifice more correctness in the system in the interest of improving performance. For example, all the current `stat()` implementations still make queries for symbolic links, since they return an accurate size. It's possible that tools would have no trouble using a symbolic link with a misreported size; if so, sacrificing this bit of correctness could remove another query. There may be other parts of the system where similar processes could be applied.

5.1.2 Display and Functional Miscellany

Many potential improvements have to do with making the system, without huge changes to functionality, do a bit more or look a bit better.

Aesthetic pathnames The system could use some improvements to the appearance of pathnames. Removing `_=`, while possible, would result in very complex queries, since the program couldn't know, pre-query, where the divisions between filters lie. Some application of caching could perhaps prevent some of this complexity, but it would require careful engineering.

Ideally, the system would also support a system where slashes didn't need to be escaped – they'd just be directory separators and the system would deal with it. Much like `_=`, though, this brings with it increased query complexity, since the system couldn't easily distinguish predicate nice-names from literal values.

Finally, removing `_files` is much more difficult and would certainly increase query complexity in the same sorts of ways. This is also only debatably useful, since removing the `_files` folder would put symbolic-link results into the parent directory of their current location, making these directories very large (perhaps most notably, the root directory, which would always list every file plus

every applicable predicate).

Combiners and grouping The current system supports only simple conjunctive queries (joined by logical AND). For example, it's impossible to construct a path whose contents would be songs by bands named either "Perfume" or "aphasia". At a shell, it's possible to use globbing, as I did in the test cases in chapter 4, to simulate logical OR for some uses – but NOT and grouping within queries[1] are unsupported entirely, except by manual post-processing.

Therefore, the system could be improved by supporting grouping, OR, and NOT. This would presumably require some additional "system folders" like the current `_=` and `_files` folders, perhaps `_not`, `_or`, `_(`, and so forth. This would make the processing of filters and the creation of queries much harder, since it requires more ways of processing filters than the current three. Nevertheless, it might be a valuable addition, since it would allow for many forms of querying not currently supported – however, it would only be valuable if any user actually needs these functions, since otherwise it merely complicates the system.

Filename display Finally, as display and functionality goes, it would be very nice to improve the final display of files. The current system uses a hardcoded `gmm:filename` link to a literal string identifying the file; this requirement to enter in a filename makes the system seem, in some ways, not much different from a normal filesystem, where each file must be named. It should, however, be possible to generate some sort of identifying string from the metadata stored in the RDF store. There are a number of ways to go about this, ranging from a naïve idea of collecting a bunch of literal strings "near" the target and plopping them together, some sort of configuration, or some sort of complicated ontology creation that identifies so-called inverse-functional predicates (that is to say, that identifies identifiers).

The naïve string-collecting implementation would work something like this: from each file, collect every string literal connected to the file by no more than, say, 2 triples; apply a consistent ordering; and join them together with underscores. This would probably collect a lot of useful strings – for music, it would presumably collect the song title, the album name, the artist name, the track number, and such things; in other words, the things I was already putting into the `gmm:filename` value. It can also collect quite useless things, though, depending on the configuration, like the titles of adjacent tracks or the names of other albums released on the same label. It's also pretty inflexible, and in many ways very ugly.

Some sort of configuration could effect a better system – perhaps a configured list of "acceptable" identifying things. For music, this might be an artist name, an album name, a track name and number, and a disc number. This could be configured per type of file, and stored somewhere (perhaps the triple store or

[1] Such as: bands from Japan named either "Perfume" or "aphasia", conceptually something like `country = JP AND ( name = Perfume OR name = aphasia )`, grouping indicated by parenthesis.

another source). The system could then select only these strings and put them in an order configured by the user, but otherwise act no differently than the above system. The weaknesses here are that it requires a lot of configuration – not quite as much as the current system, since in this case a user would just be configuring by-type rather than by-file – but still a lot. It also requires users to understand a lot about the way the system works, which isn't always desirable (though in this case, it may not matter).

Finally, the system could try to be really smart. Many ontology definition languages, such as OWL (Web Ontology Language) have predicates that describe other predicates. One such classification of predicates is as an "inverse functional" property – that is to say, the inverse of that predicate is functional: for each input, there is a unique output. Ontologies describing people sometimes use email addresses for this – email addresses serve as an identifier for people. This system would need something even more complex, since ideally it would allow composite identifiers, such as those listed in the above configuration-oriented proposal. Nevertheless, this would move the task further into the realm of the ontology creator, and out of the hands of the user, which makes the overall system more automatic and more "smart."

One of the second or third options, a combination of them, or some even-more-complex system would be the best improvement to this feature. Other potentially-valuable inclusions in the filename system might include hierarchy and grouping within the `_files` folder, such as displaying songs grouped by the album they appear on, by their artist, and so forth; these are likely only possible with configuration, whether at an ontology level (option 3) or entered by the user (option 2).

With the addition of hierarchy, the earlier-discussed possibility of removing the `_files` directory seems less unreasonable, if we can assume the hierarchy reduces the complexity of the results; in fact, a sufficiently advanced configuration could even remove the need for many or most explicitly path-embedded queries, by providing a default view (or interpretation) of files (or several such views/interpretations).

5.1.3 Adding and updating metadata

This system largely assumes a static set of data – from `rdfgen.py` being a fairly large task, designed to fully replace the contents of an RDF store, to the caching setup, to the read-only nature of the filesystem, I've assumed that when people want to change data, they'll want to update everything all at once. This probably isn't true – as I get new albums, new movies, or add files related to some project I'm working on, I'd want to add these things to the system without having to completely recreate all my mostly-unchanged metadata.

Therefore, the system might be improved by some functionality concerning additions and updates of metadata. In fact, this missing functionality is the primary piece missing to transform this system into more of a true "generic metadata manager."

Read-write filesystem One idea for adding and updating data within the store is to work incrementally from what's already made: why not make it possible to add and move files within the filesystem in order to update their associated metadata? This is neatly in line with the principle of making the system use the established, time-tested interface of the filesystem; it's a clean and logical next step. Unfortunately, it's a pretty complicated task to define the behavior of such a system, and it's not clear without further research that it's even *possible* to encode as filesystem operations all the tools a user would desire.

The statelessness of filesystem calls could pose one problem: how does the system know that a file removal in one place and a subsequent creation somewhere else are related? How does moving a file previously outside the system appear when it suddenly "morphs" into a symbolic link? How does the system determine the canonical `gmm:fileContent` location of data, or does it create it itself?

Even if these problems can be solved, some things may simply not be possible – if I move a file to a place designating its artist-name should be set to "Perfume", how would the system know *which* Perfume to link? The 'simple' and 'flat' schemes have the potential to work nicely with this, since there's no question of intermediate objects – but they have their own weaknesses, such as limiting re-use of shared values, complicating data maintenance tasks.

In general, it seems that a read-write filesystem, while a logical next step, is deceptively difficult to create and to understand; a different interface might be more desirable for both developers and users.

Manual interfaces Another approach is to create a completely separate interface, designed to be a fully general RDF editor. Unsurprisingly, this is a project that's been taken on in other contexts where RDF is used, and some of those tools and interface breakthroughs could be transferred to this system. That being said, a fully general system must necessarily know little or nothing about specific domains, or must become very complex in order to understand a number of domains. General RDF-editing tools are valuable, especially in edge cases where a domain-specific tool simply won't do (ad-hoc and personal projects, for example, that the outside world need not know anything about); however, for many cases domain-specific tools, that do more work at the expense of generality, are desirable.

Automatic extraction Domain-specific tools can take a number of forms. One form is a fully automatic extraction tool; in fact, `rdfgen.py` is an example of such a tool. For the task of iteratively adding/updating metadata, `rdfgen.py` could be updated to know about the existing database and use that rather than working completely without prior input. Other similar tools could be written for other domains, using different data sources (perhaps IMDB for movies, embedded EXIF data for photos, or various citation databases for research papers).

Many, perhaps most, domains don't have the nice embedded metadata that music does, as mentioned earlier in chapter 2 (p. 12).

While sometimes convenient online databases of metadata and automatic identification tools exist, a fully-automatic system will always have the potential to make mistakes, especially with very similar things – which are, of course, the most important to correctly distinguish. For those applications where a fully-automatic system is possible, though, this is clearly one of the most user-friendly options for adding and updating metadata.

Semi-automatic extraction For those cases where good databases exist, and some tools exist to provide identification, a semi-automatic approach might be best. By semi-automatic, in this case, I mean a system where the system tries to match as best it can, but has the user manually "check off" on its choices – in effect, the user can act as a reviewer of the computer's work. On top of this, a semi-automatic system can allow some limited manual editing – say, when the online database is wrong about a name that the user happens to know, and they'd like to correct it.

It's also possible to create semi-automatic tools for extracting in a more-general case where the data source is simply the structure of the pathname. Some music-tagging programs already have features like this for updating embedded metadata, where given the details of file naming, the tool can extract strings to the proper place. This is a limited source, but extending this pattern to this system would have value, especially for users who already have large collections of data that they wish to migrate to this system.

As a final note, both automatic and semi-automatic extraction mitigate the previously-discussed problem of displaying filenames – as `rdfgen.py` demonstrates, an automatic or semi-automatic system can spare the user the task of manually entering `gmm:filename` links for every file.

Where this is not possible, manual interfaces seem the only choice. But where it is possible, semi-automatic interfaces seem to be a very good way to add and update metadata within a system. For music, such a system might be implemented as a plugin for MusicBrainz Picard; for movies, perhaps some integration with VLC or with common downloading and ripping tools; for research papers, perhaps integration with bibliograhic tools like Zotero: in many domains, tools already exist where this functionality could be added. Of course, this is a problem requiring a lot of work, and which moves somewhat away from the generic nature of the system we have now. The interface design is not always straightforward, and it's a topic not-deeply-explored in many domains.

Ultimately, a combination of these tactics is likely to be the most successful; where applicable, automatic and semi-automatic extraction are very user-friendly; in other realms manual editing is the only option.

Ramifications I mentioned in section 3.3 that adding and updating data makes caching such as my reverse proxy setup much more complicated. This, like so many caching problems, comes down to the question of invalidation: when something is added or updated, which things in the cache are no longer valid? In a system such as this, where a given triple can have effects spreading widely

through the system, it could be very hard to track down an exact answer to this question. What this probably means is that only two options remain: invalidating the entire cache on every addition or update, or ignoring the invalidation problem and relying on set limits. The former limits the value of caching – in a system updating near-constantly, it makes caching essentially impossible. The latter would mean a system where there's no guarantee as to exactly when changes will appear, which has the potential to be very frustrating or even frightening to a user. More-advanced caching could alleviate some of this problem, as discussed earlier.

In light of this, it seems possible that some patterns of adding and updating metadata bring with them performance penalties, since the system was unusably slow without caching. Sadly, exploring this rich problem is beyond the scope of my work as part of this project, both due to necessary design effort and these caching/performance ramifications.

5.1.4 Ontological concerns

Finally, there is one major open question of this system (and many other metadata system): how do people define and use ontologies? A poorly-defined ontology limits the usefulness of a system, or can have performance ramifications – the separate 'deep', 'simple', and 'flat' schemes, and their particular quirks, demonstrate this clearly: the 'simple' and 'flat' structures have data-maintenance troubles, while the 'deep' structure is prone to some difficult limitations to performance and extensibility (such as the difficulty of creating a read-write filesystem).

Despite its importance, the answer to this question is entangled in a complicated web of social and technical factors that's very hard to navigate. Other projects have a wide variety of approaches. NEPOMUK defines some ontologies as part of its standard, but includes a special ontology specifically for power-users, designed for defining personal ontologies (called PIMO). MusicBrainz expands its own ontology (of sorts) by way of a "style council" that discusses these issues. Commercial enterprises such as IMDB also create their own ontologies, of sorts, but their processes are hidden to us. Presumably they too use some sort of committee structure, or have dedicated professional ontologists of some sort. It is my opinion that NEPOMUK is in many ways closest to "right", in carefully defining common cases but allowing extensions; I don't necessarily agree with their ways of doing it, though.

In any case, a complete system based around this prototype would need to define its own way of handling this ontological question; this answer would be part of the self-definition of whatever community ostensibly came to exist around this system, and it would speak sharply to its values. This would be the ultimate future work of the project: to foster a community built on the ideals of this project, and grow that community into its fullest potential (in part by answering, or helping to answer, the other questions of this chapter).

5.2 Conclusions

In chapter 1, I said I wanted a better way of storing, editing, and retrieving metadata, and using it to find data, while using the established interface of the filesystem. By implementing to the scenario laid out in section 1.2, I've created a very successful prototype that allows me to use metadata from an RDF store to find data. This prototype has some performance issues and some missing functionality, but provides a proof-of-concept in a previously unexplored realm of conceptually embedding queries against a metadata store in filesystem paths. The process of developing and implementing to a scenario was highly valuable in reaching this point.

I unfortunately haven't created real interfaces for editing metadata, but I've found and integrated with a rich ecosystem of tools for this task by using RDF, and considered possible interfaces for this task in section 5.1.3. I also haven't deeply considered the issues of a community or broader adoption of such a tool, though I touched on these concerns in section 5.1.4.

Though I haven't reached my ultimate goal of a true "generic metadata manager," through the process outlined in this document I've created a prototype of a dynamic metadata-backed filesystem, evaluated it, and laid out future work on this topic. I am excited by the potential for further development of tools to fill this need.

Bibliography

GNOME Developers. Tracker. http://projects.gnome.org/tracker/, 2011. URL http://projects.gnome.org/tracker/.

Joey Hess. git-annex. http://git-annex.branchable.com/, October 2011. URL http://git-annex.branchable.com/.

Christian Jodar. GCstar, personal collections manager. http://www.gcstar.org/, August 2010. URL http://www.gcstar.org/.

KDE Developers. Nepomuk. http://nepomuk.kde.org/, 2011. URL http://nepomuk.kde.org/.

Ben Martin. www. http://www.libferris.com/, August 2010. URL http://www.libferris.com/.

Matroska. Tag specifications | matroska. http://www.matroska.org/technical/specs/tagging/index.html, 2012. URL http://www.matroska.org/technical/specs/tagging/index.html.

MusicBrainz Contributors. Musicbrainz picard/tags/mapping. http://wiki.musicbrainz.org/MusicBrainz_Picard/Tags/Mapping, February 2012. URL http://wiki.musicbrainz.org/MusicBrainz_Picard/Tags/Mapping.

Dan O'Neill. Id3v1. http://www.id3.org/ID3v1, October 2006. URL http://www.id3.org/ID3v1.

W3C. RDF primer, February 2004. URL http://www.w3.org/TR/rdf-primer/.

W3C. SPARQL query language for RDF, January 2008. URL http://www.w3.org/TR/rdf-sparql-query/#introduction.

Wikipedia Contributors. Exchangeable image file format. https://en.wikipedia.org/wiki/Exchangeable_image_file_format, 2012a. URL https://en.wikipedia.org/wiki/Exchangeable_image_file_format.

Wikipedia Contributors. File system. http://en.wikipedia.org/wiki/File_system, 2012b. URL http://en.wikipedia.org/wiki/File_system.

Appendix A

gmmFuse.hs

```haskell
module Main where

import qualified Data.ByteString.Char8 as B
import System.Environment
import Foreign.C.Error
import System.Posix.Types
import System.Posix.Files
import System.Posix.IO
import System.Locale

import System.Fuse
import System.FilePath

import Database.HSparql.Connection
import Database.HSparql.QueryGenerator
import Codec.Binary.UTF8.String

import Text.Printf

import Data.List hiding (union)

import Control.Monad

-- |main takes two (system) args: first should be a SPARQL
-- endpoint, the second a mountpoint
```

```
26 main :: IO ()
27 main = do
28     args <- getArgs
29     withArgs [last args] $
30         fuseMain (gmmFSOps (head args) (tail $ init args))
31                  defaultExceptionHandler
32
33 -- * FUSE Operations
34 -- |gmmFSOps is the top-level FUSE operations for the system;
35 -- it takes a SPARQL endpoint to use throughout
36 gmmFSOps :: EndPoint -> [String] -> FuseOperations ()
37 gmmFSOps ep perf = defaultFuseOps
38         { fuseGetFileStat        = fileStat ep
39         , fuseOpenDirectory      = openDirectory ep
40         , fuseReadDirectory      = gmmReadDirectory ep
41         , fuseGetFileSystemStats = gmmGetFileSystemStats ep
42         , fuseReadSymbolicLink   = gmmReadSymbolicLink ep
43         }
44         where fileStat = if "stat1" `elem` perf || "stat" `elem` perf
45                          then gmmGetFileStat'
46                          else if "stat2" `elem` perf
47                               then gmmGetFileStat''
48                               else gmmGetFileStat
49               openDirectory = if "opendir" `elem` perf
50                               then gmmOpenDirectory'
51                               else gmmOpenDirectory
52
53 -- |gmmGetFileStat, given a SPARQL endpoint and a path,
54 -- returns the relevant FileStat or eNOENT
55 gmmGetFileStat :: EndPoint -> FilePath -> IO (Either Errno FileStat)
56 gmmGetFileStat ep path
57     | isFilesDir = do
58         ctx <- getFuseContext
59         rootPaths <- {-# SCC "rootPaths._file" #-}
60                      queryFileEntries ep rootPath
61         return $ if tailPath `elem` fmap fst rootPaths
62                     then case lookup tailPath rootPaths of
63                               Just s -> Right $ symlinkStat ctx $
```

```
64                                          B.length $ B.pack $ drop 7 s
65                              Nothing -> Left eNOENT
66                      else Left eNOENT
67      | path == "/" || tailPath `elem` ["_=", "_files"] = do
68          ctx <- getFuseContext
69          myPaths <- {-# SCC "myPaths._special" #-}
70                     queryDirectoryEntries ep processedPath
71          return $ Right $ dirStat ctx (2 + length myPaths)
72      | otherwise                       = do
73          ctx <- getFuseContext
74          rootPaths <- {-# SCC "rootPaths._regular" #-}
75                       queryDirectoryEntries ep rootPath
76          myPaths <- {-# SCC "myPaths._regular" #-}
77                     queryDirectoryEntries ep processedPath
78          return $ if tailPath `elem` rootPaths
79                      then Right $ dirStat ctx (2 + length myPaths)
80                      else Left eNOENT
81      where processedPath = '/':dropWhileEnd isPathSeparator (tail path)
82            (rootPath, tailPath) = splitFileName processedPath
83            isFilesDir = "_files" == snd (splitFileName
84                                          (dropWhileEnd isPathSeparator rootPath))
85
86  -- |gmmGetFileStat' is a performant, but incorrect implementation of
87  -- 'gmmGetFileStat' (never checks for st_nlink values, otherwise the same)
88  gmmGetFileStat' :: EndPoint -> FilePath -> IO (Either Errno FileStat)
89  gmmGetFileStat' ep path
90      | isFilesDir = do
91          ctx <- getFuseContext
92          rootPaths <- {-# SCC "rootPaths._file" #-} queryFileEntries ep rootPath
93          return $ if tailPath `elem` fmap fst rootPaths
94                      then case lookup tailPath rootPaths of
95                               Just s -> Right $ symlinkStat ctx $
96                                                 B.length $ B.pack $ drop 7 s
97                               Nothing -> Left eNOENT
98                      else Left eNOENT
99      | path == "/" || tailPath `elem` ["_=", "_files"] = do
100         ctx <- getFuseContext
101         return $ Right $ dirStat ctx 2
```

```
102     | otherwise                     = do
103         ctx <- getFuseContext
104         rootPaths <- {-# SCC "rootPaths._regular" #-}
105                      queryDirectoryEntries ep rootPath
106         return $ if tailPath `elem` rootPaths
107                     then Right $ dirStat ctx 2
108                     else Left eNOENT
109     where processedPath = '/':dropWhileEnd isPathSeparator (tail path)
110           (rootPath, tailPath) = splitFileName processedPath
111           isFilesDir = "_files" == snd (splitFileName
112                                         (dropWhileEnd isPathSeparator rootPath))
113
114 -- |gmmGetFileStat'' is a performant, but incorrect, implementation of
115 -- 'gmmGetFileStat' (for non-symlinks, never returns eNOENT, and does
116 -- not return st_nlink values that mean anything)
117 gmmGetFileStat'' :: EndPoint -> FilePath -> IO (Either Errno FileStat)
118 gmmGetFileStat'' ep path
119     | isFilesDir = do
120         ctx <- getFuseContext
121         -- we still need to query here, since we care about sizes
122         rootPaths <- {-# SCC "rootPaths._file" #-}
123                      queryFileEntries ep rootPath
124         return $ if tailPath `elem` fmap fst rootPaths
125                     then case lookup tailPath rootPaths of
126                               Just s -> Right $ symlinkStat ctx $
127                                                 B.length $ B.pack $ drop 7 s
128                               Nothing -> Left eNOENT
129                     else Left eNOENT
130     | otherwise = do
131         ctx <- getFuseContext
132         return $ Right $ dirStat ctx 2
133     where (rootPath, tailPath) = splitFileName path
134           isFilesDir = "_files" == snd (splitFileName
135                                         (dropWhileEnd isPathSeparator rootPath))
136
137 -- |gmmOpenDirectory, given a SPARQL endpoint and a path,
138 -- returns eOK or eNOENT depending on the existence of a directory
139 gmmOpenDirectory :: EndPoint -> FilePath -> IO Errno
```

```haskell
gmmOpenDirectory ep path
    | path == "/" || tailPath `elem` ["_=", "_files"] = return eOK
    -- no subdirectories of _files, only symlinks
    | isFilesDir                                      = return eNOENT
    | otherwise                                       = do
        rootPaths <- {-# SCC "rootPaths" #-}
                     queryDirectoryEntries ep rootPath
        return $ if tailPath `elem` rootPaths
                    then eOK
                    else eNOENT
    where (rootPath, tailPath) = splitFileName path
          isFilesDir = "_files" == snd (splitFileName
                                        (dropWhileEnd isPathSeparator rootPath))

-- | gmmOpenDirectory' is a performant, but incorrect, implementation of
-- 'gmmOpenDirectory' (always returns eOK)
gmmOpenDirectory' :: EndPoint -> FilePath -> IO Errno
gmmOpenDirectory' _ _   = return eOK

-- |gmmReadDirectory, given a SPARQL endpoint and a path,
-- either returns eNOENT for a nonexistent directory
-- or returns a set of entries, presumably pulled from the RDF store
gmmReadDirectory ::
    EndPoint -> FilePath -> IO (Either Errno [(FilePath, FileStat)])
gmmReadDirectory ep path = do
    ctx <- getFuseContext
    dirEnts <- directoryEntries ep path ctx
    return $ case dirEnts of
                  Nothing  -> Left eNOENT
                  Just ent -> Right ent

gmmReadSymbolicLink :: EndPoint -> FilePath -> IO (Either Errno FilePath)
gmmReadSymbolicLink ep path
    | valid = do
        rootPaths <- {-# SCC "rootPaths" #-}
                     queryFileEntries ep rootPath
        return $ if tailPath `elem` fmap fst rootPaths
                    then case lookup tailPath rootPaths of
```

```
178                                 Just s -> Right $ drop 7 s
179                                 Nothing -> Left eNOENT
180                         else Left eNOENT
181     where (rootPath, tailPath) = splitFileName path
182           valid = "_files" == snd (splitFileName
183                                      (dropWhileEnd isPathSeparator rootPath))
184
185 -- |gmmGetFileSystemStats returns a FileSystemStats object.
186 -- I'm honestly not sure what it's supposed to return.
187 -- I should look into that.
188 gmmGetFileSystemStats ::
189     EndPoint -> FilePath -> IO (Either Errno FileSystemStats)
190 gmmGetFileSystemStats _ _ =
191     return $ Right FileSystemStats
192       { fsStatBlockSize = 512
193       , fsStatBlockCount = 1
194       , fsStatBlocksFree = 1
195       , fsStatBlocksAvailable = 1
196       , fsStatFileCount = 5
197       , fsStatFilesFree = 10
198       , fsStatMaxNameLength = 255
199       }
200
201 -- * 'gmmGetFileStat' utilities
202 -- |dirStat provides a FileStat for a directory,
203 -- given a FuseContext and a st_nlink number
204 dirStat :: Integral a => FuseContext -> a -> FileStat
205 dirStat ctx links = FileStat { statEntryType = Directory
206                          , statFileMode = foldr1 unionFileModes
207                                             [ ownerReadMode
208                                             , ownerExecuteMode
209                                             , groupReadMode
210                                             , groupExecuteMode
211                                             , otherReadMode
212                                             , otherExecuteMode
213                                             ]
214                          , statLinkCount = fromIntegral links
215                          , statFileOwner = fuseCtxUserID ctx
```

```
216                           , statFileGroup = fuseCtxGroupID ctx
217                           , statSpecialDeviceID = 0
218                           , statFileSize = 4096
219                           , statBlocks = 1
220                           , statAccessTime = 0
221                           , statModificationTime = 0
222                           , statStatusChangeTime = 0
223                           }
224
225 -- |symlinkStat provides a FileStat for a symlink,
226 -- given a FuseContext and a size number
227 symlinkStat :: Integral a => FuseContext -> a -> FileStat
228 symlinkStat ctx size = FileStat { statEntryType = SymbolicLink
229                                 , statFileMode = foldr1 unionFileModes
230                                                         [ ownerReadMode
231                                                         , groupReadMode
232                                                         , otherReadMode
233                                                         ]
234                                 , statLinkCount = 1
235                                 , statFileOwner = fuseCtxUserID ctx
236                                 , statFileGroup = fuseCtxGroupID ctx
237                                 , statSpecialDeviceID = 0
238                                 , statFileSize = fromIntegral size
239                                 , statBlocks = 1
240                                 , statAccessTime = 0
241                                 , statModificationTime = 0
242                                 , statStatusChangeTime = 0
243                                 }
244
245 -- * 'gmmReadDirectory' utilities
246 -- |directoryEntries returns a list of directory entries for
247 -- a given path; it calls out to helper functions to determine
248 -- what should be listed for a given path
249 directoryEntries ::
250     EndPoint -> FilePath -> FuseContext -> IO (Maybe [(FilePath, FileStat)])
251 directoryEntries ep path ctx
252     | last pathParts `elem` ["_files", "_files/"] = do
253         rootPaths <- {-# SCC "rootPaths" #-}
```

```
254                     queryFileEntries ep path
255         return $ Just $
256             defaultDirectoryEntries ctx
257             ++ zip (fmap fst rootPaths)
258                    (fmap (symlinkStat ctx . symlinkSize) rootPaths)
259     | otherwise = do
260         rootPaths <- {-# SCC "rootPaths" #-}
261                      queryDirectoryEntries ep path
262         return $ Just $ defaultDirectoryEntries ctx
263                         ++ zip rootPaths (repeat $ dirStat ctx 2)
264                         ++ case take 2 $ reverse parts of
265                                []        -> filesDirectoryEntry ctx
266                                [_, "_="] -> filesDirectoryEntry ctx
267                                ["_=", _] -> []
268                                _         -> eqDirectoryEntry ctx
269     where parts = last $ extractPaths path
270           pathParts = splitPath path
271           symlinkSize = B.length . B.pack . drop 7 . snd
272
273 -- |reusable default directory entries . and ..
274 defaultDirectoryEntries :: FuseContext -> [(FilePath, FileStat)]
275 defaultDirectoryEntries ctx = [(".", dirStat ctx 2), ("..", dirStat ctx 2)]
276
277 -- |reusable _files listing
278 filesDirectoryEntry :: FuseContext -> [(FilePath, FileStat)]
279 filesDirectoryEntry ctx = [("_files", dirStat ctx 2)]
280
281 -- |reusable _= listing
282 eqDirectoryEntry :: FuseContext -> [(FilePath, FileStat)]
283 eqDirectoryEntry ctx = [("_=", dirStat ctx 2)]
284
285 -- * SPARQL Querying
286 -- |queryDirectoryEntries runs a query to gather directory-entry results
287 queryDirectoryEntries :: EndPoint -> FilePath -> IO [String]
288 queryDirectoryEntries ep full@('/':path) = do
289     (Just results) <- {-# SCC "query" #-} query ep $
290                           directoryEntriesQuery $ processedPath
291     return $ fmap ((!! 0) . fmap (escapePathChunk . illiterate)) results
```

```
292     where processedPath = '/':dropWhileEnd isPathSeparator path
293
294 -- |queryFileEntries runs a query to gather directory-entry
295 -- results for the _files folder
296 queryFileEntries :: EndPoint -> FilePath -> IO [(String, String)]
297 queryFileEntries ep full@('/':path)
298     | last pathParts `elem` ["_files", "_files/"] = do
299         (Just results) <- {-# SCC "query" #-} query ep $
300                                 directoryEntriesQuery $ processedPath
301         return $ fmap packFilename $ zip (numbers results)
302                                          (fmap (fmap illiterate) results)
303     where pathParts = splitPath full
304           processedPath = '/':dropWhileEnd isPathSeparator path
305           printfDigits list = printf $
306                                   "%0" ++ show (numDigits $ length list) ++ "d"
307           numbers list = (fmap (printfDigits list) ([1..] :: [Integer]))
308           packFilename (n, [res, filename]) =
309               (n ++ '_':filename, encodeString res)
310
311 -- |count the number of digits in a number; for use in
312 -- 'queryFileEntries' for uniquifying entries
313 numDigits = length . map (`mod` 10) . reverse .
314                 takeWhile (> 0) . iterate (`div` 10)
315
316 -- |illiterate takes Literal (or URI) x and returns a utf-8-decoded x.
317 -- clever, I know :P
318 illiterate :: BindingValue -> String
319 illiterate (Literal x) = decodeString x
320 illiterate (URI x) = decodeString x
321
322 -- |SPARQL query to gather directory entries
323 -- (to be called by 'queryDirectoryEntries' and 'queryFileEntries')
324 directoryEntriesQuery :: FilePath -> Query [Variable]
325 directoryEntriesQuery path
326     | last pathParts == "_files" = do
327         (name, gmm, metaobj, file) <- baseQuery
328         name <- foldM (pathQuery gmm metaobj) name parts
329         filename <- var
```

```
330         triple file (gmm .:. "filename") filename
331         orderNext filename
332         orderNext file
333         return [file, filename]
334     | otherwise                  = do
335         (name, gmm, metaobj, file) <- baseQuery
336         name <- foldM (pathQuery gmm metaobj) name parts
337         return [name]
338     where parts = extractPaths path
339           pathParts = splitPath path
340
341 -- pathQuery :: Prefix -> Variable -> Variable -> GMMPath -> Query [Variable]
342 -- |SPARQL: combinable query chunk for one GMMPath
343 pathQuery gmm metaobj name path
344     | "_=" `notElem` path = do
345         lasthop <- foldM (predicateCase gmm) metaobj pathPredicates
346         predicateCase gmm lasthop name
347         filterExpr $ isLiteral name
348         return name
349     | "_=" == last path   = do
350         lasthop <- foldM (predicateCase gmm) metaobj pathPredicates
351         filterExpr $ isLiteral lasthop
352         return lasthop
353     | completePath path   = do
354         lasthop <- foldM (predicateCase gmm) metaobj (init pathPredicates)
355         pred <- var
356         predicateCaseVars gmm lasthop (last pathPredicates) (last path) pred
357         return name
358     where pathPredicates = takeWhile (/= "_=") path
359
360 -- |SPARQL query base: SELECT DISTINCT plus { start gmm:fileContent file. }
361 -- plus some variables to reuse
362 baseQuery = do
363     name <- var
364     metaobj <- var
365     file <- var
366
367     gmm <- prefix (iriRef "http://ianmcorvidae.net/gmm-ns/")
```

```
368
369     distinct
370     triple metaobj (gmm .:. "fileContent") file
371
372     return (name, gmm, metaobj, file)
373
374 -- |SPARQL: { one-hop predicate two-hops. predicate gmm:niceName name. }
375 predicateCase gmm hop name = do
376     pred <- var
377     hop1 <- var
378     predicateCaseVars gmm hop name hop1 pred
379     return hop1
380
381 -- |Same as 'predicateCase' except it doesn't set its
382 -- own vars, so it can be used in a 'union'.
383 predicateCaseVars gmm hop name hop1 pred = do
384     triple hop pred hop1
385     triple pred (gmm .:. "niceName") name
386
387 -- * File and GMMPath processing utility functions
388 -- | takes a filePath (rather, a chunk thereof) and escapes it
389 escapePathChunk :: FilePath -> FilePath
390 escapePathChunk path =
391     concat $ fmap escapeChar path
392
393 -- | take a single char and return a string that should appear
394 -- in an escaped string
395 escapeChar :: Char -> String
396 escapeChar ('/') = "#%"
397 escapeChar ('#') = "##"
398 escapeChar char  = char:[]
399
400 -- | inverse of 'escapePathChunk'; turn an escaped path-chunk
401 -- into an unescaped one
402 unescapePathChunk :: FilePath -> FilePath
403 unescapePathChunk [] = []
404 unescapePathChunk path
405     | head path == '#' =
```

```
406         unescapeChar (path !! 1) : unescapePathChunk (drop 2 path)
407     | otherwise        =
408         head path : unescapePathChunk (tail path)
409
410 -- | inverse of 'escapeChar', roughly, takes the character following
411 -- the escape character and returns the appropriate translation
412 unescapeChar :: Char -> Char
413 unescapeChar ('%') = '/'
414 unescapeChar ('#') = '#'
415 unescapeChar char  = char
416
417 -- | a GMMPath is a collection of strings (from FilePaths) that describe
418 -- a path to take within the RDF store; 'pathQuery' processes one into
419 -- SPARQL
420 type GMMPath = [FilePath]
421
422 -- | completePath tests if a GMMPath is a complete filter
423 completePath :: GMMPath -> Bool
424 completePath path@(_:_:_:_) =
425     "_=" == last (init path)
426 completePath _ = False
427
428 -- | take a FilePath and turn it into a list of GMMPaths
429 extractPaths :: FilePath -> [GMMPath]
430 extractPaths path
431     | paths == []               = [[]]
432     | completePath (last paths) = paths ++ extractPaths "/"
433     | otherwise                 = paths
434     where filterPart = unescapePathChunk . encodeString .
435                            dropWhileEnd isPathSeparator
436           paths = groupPaths $
437                       fmap filterPart
438                            (takeWhile (/= "_files") $ tail $ splitPath path)
439
440 -- | take a split FilePath and group it into GMMPaths
441 groupPaths :: [FilePath] -> [GMMPath]
442 groupPaths [] = []
443 groupPaths partList =
```

```
444       (fst partSpan ++ fst sndSplit) : groupPaths (snd sndSplit)
445       where partSpan = span (/= "_=") partList
446             sndSplit = splitAt 2 $ snd partSpan
```

Appendix B

rdfgen.py

```
#!/usr/bin/python2
#
# Processes a flac file into some RDF
# Usage: <script> [rdf filename] [processing type] flac-filename(s)
# processing types are simple, flat, and deep

from __future__ import print_function

import sys
from subprocess import check_output

from rdflib.graph import Graph
from rdflib.term import URIRef, Literal, BNode
from rdflib.namespace import Namespace, RDF

import musicbrainzngs as mb
mb.set_useragent("ianmcorvidae's rdfgen.py", "0.0.1", "http://ianmcorvidae.net")

MS = Namespace("http://ianmcorvidae.net/ms-ns/")
GMM = Namespace("http://ianmcorvidae.net/gmm-ns/")

simple_tags = ['artist', 'album', 'title']
flat_tags          = ['artist', 'album', 'title', 'albumartist',
                      'date', 'script', 'language', 'label',
                      'totaltracks',  'totaldiscs',  'discnumber',
```

```
                      'format', 'catalognumber', 'releasecountry',
                      'media', 'asin', 'releasestatus', 'originaldate',
                      'tracknumber', 'releasetype',
                      'musicbrainz_albumartistid', 'musicbrainz_albumid',
                      'musicbrainz_artistid', 'musicbrainz_trackid',
                      'lyricist', 'composer', 'mixer', 'performer']
name_map = {'releasecountry': 'country',
            'releasestatus': 'status',
            'releasetype': 'type'}
flat_tag_nicenames = ['Artist', 'Album', 'Title', 'Album Artist',
                      'Date', 'Script', 'Language', 'Label',
                      'Total Tracks', 'Total Discs', 'Disc Number',
                      'Format', 'Catalog Number', 'Country',
                      'Media', 'ASIN', 'Status', 'Original Date',
                      'Track Number', 'Type',
                      'MusicBrainz Album Artist ID', 'MusicBrainz Album ID',
                      'MusicBrainz Artist ID', 'MusicBrainz Track ID',
                      'Lyricist', 'Composer', 'Mixer', 'Performer']

def add_all(store, subject, predicate, tags, tag_name):
    for entry in tags[tag_name]:
        store.add((subject, predicate, Literal(entry)))

# processors
def process_common(store, filename, tags):
    def get_tag(tags, tag, after='_', default=''):
        try:
            return tags[tag][0] + after
        except:
            return default + after

    metaobj = BNode()
    store.add((metaobj, GMM['fileContent'], URIRef('file://' + filename)))
    store.add((metaobj, RDF.type, MS['Track']))

    fname = (get_tag(tags, 'artist') + get_tag(tags, 'album') +
             get_tag(tags, 'discnumber', '-', '1') +
             get_tag(tags, 'tracknumber').zfill(len(get_tag(tags, 'totaltracks'))))
```

```
64              get_tag(tags, 'title', '') + '.flac').replace('/', '_')
65      store.add((URIRef('file://' + filename), GMM['filename'], Literal(fname)))
66
67      store.add((RDF.type, GMM['niceName'], Literal('Type')))
68
69      return metaobj
70
71  def process_simple(store, filename, tags):
72      metaobj = process_common(store, filename, tags)
73
74      store.add((MS['artist'], GMM['niceName'], Literal('Artist')))
75      store.add((MS['album'], GMM['niceName'], Literal('Album')))
76      store.add((MS['title'], GMM['niceName'], Literal('Title')))
77
78      for each in simple_tags:
79          add_all(store, metaobj, MS[each], tags, each)
80
81  def process_flat(store, filename, tags):
82      metaobj = process_common(store, filename, tags)
83
84      for each in flat_tags:
85          nicename = Literal(flat_tag_nicenames[flat_tags.index(each)])
86          pred = MS[name_map.get(each, each)]
87          store.add((pred, GMM['niceName'], nicename))
88          try:
89              add_all(store, metaobj, pred, tags, each)
90          except KeyError:
91              pass
92
93  def process_deep(store, filename, tags):
94      metaobj = process_common(store, filename, tags)
95
96      for each in flat_tags:
97          nicename = Literal(flat_tag_nicenames[flat_tags.index(each)])
98          pred = MS[name_map.get(each, each)]
99          store.add((pred, GMM['niceName'], nicename))
100
101     store.add((MS['track'], GMM['niceName'], Literal('Track')))
```

```
102     store.add((MS['name'], GMM['niceName'], Literal('Name')))
103     store.add((MS['sortname'], GMM['niceName'], Literal('Sort Name')))
104     store.add((MS['credit'], GMM['niceName'], Literal('Credited Name')))
105 
106     # base object stuff
107     [add_all(store, metaobj, MS[item], tags, item) for item in
108             ['title', 'tracknumber', 'discnumber']]
109             #['lyricist', 'composer', 'mixer', 'performer']
110 
111     # album stuff
112     album = URIRef('http://musicbrainz.org/release/%s#_' %
113                    tags['musicbrainz_albumid'][0])
114     store.add((metaobj, MS['album'], album))
115     store.add((album, MS['track'], metaobj))
116     store.add((album, RDF.type, MS['Album']))
117 
118     [add_all(store, album, MS[name_map.get(item, item)], tags, item) for item in
119             ['date', 'originaldate', 'script', 'language',
120              'totaltracks', 'totaldiscs', 'format', 'catalognumber',
121              'releasecountry', 'media', 'asin',
122              'releasestatus', 'releasetype']]
123              #['label']
124 
125     add_all(store, album, MS['title'], tags, 'album')
126 
127     # artist stuff
128     artists = []
129     for each in tags['musicbrainz_artistid']:
130         artist = URIRef('http://musicbrainz.org/artist/%s#_' % each)
131         store.add((metaobj, MS['artist'], artist))
132         store.add((artist, RDF.type, MS['Artist']))
133         store.add((artist, MS['track'], metaobj))
134         artists.append(artist)
135 
136     albumartists = []
137     for each in tags['musicbrainz_albumartistid']:
138         albumartist = URIRef('http://musicbrainz.org/artist/%s#_' %
139                              tags['musicbrainz_albumartistid'][0])
```

```
140             store.add((album, MS['artist'], albumartist))
141             store.add((albumartist, RDF.type, MS['Artist']))
142             store.add((albumartist, MS['album'], album))
143             albumartists.append(albumartist)
144 
145         for artist in artists:
146             add_all(store, artist, MS['credit'], tags, 'artist')
147         for albumartist in albumartists:
148             add_all(store, albumartist, MS['credit'], tags, 'albumartist')
149 
150     def postprocess_deep(store, tags):
151         artists = dict([(item[-38:-2], item) for item in
152                             store.subjects(RDF.type, MS['Artist'])])
153         albums = dict([(item[-38:-2], item) for item in
154                             store.subjects(RDF.type, MS['Album'])])
155 
156         for mbid, uri in artists.iteritems():
157             try:
158                 print('API: artist %s' % mbid, file=sys.stderr)
159                 artist_data = mb.get_artist_by_id(mbid)
160                 name = artist_data['artist']['name']
161                 sortname = artist_data['artist']['sort-name']
162                 store.add((uri, MS['name'], Literal(name)))
163                 store.add((uri, MS['sortname'], Literal(sortname)))
164                 if 'country' in artist_data['artist'].keys():
165                     country = artist_data['artist']['country']
166                     store.add((uri, MS['country'], Literal(country)))
167                 if 'type' in artist_data['artist'].keys():
168                     a_type = artist_data['artist']['type']
169                     store.add((uri, MS['type'], Literal(a_type)))
170             except:
171                 continue
172 
173         for mbid, uri in albums.iteritems():
174             try:
175                 print('API: album %s' % mbid, file=sys.stderr)
176                 album_data = mb.get_release_by_id(mbid, ['labels'])
177                 for label_data in album_data['release'].get('label-info-list', []):
```

```
178                 label = URIRef('http://musicbrainz.org/label/%s#_' %
179                                label_data['label']['id'])
180                 store.add((uri, MS['label'], label))
181                 if label not in store.subjects(RDF.type, MS['Label']):
182                     print('    label %s' %
183                              label_data['label']['id'], file=sys.stderr)
184                     add_label(store, label, label_data)
185         except:
186             continue
187
188 def add_label(store, labeluri, label_data):
189     name = label_data['label']['name']
190     sortname = label_data['label']['sort-name']
191     store.add((labeluri, RDF.type, MS['Label']))
192     store.add((labeluri, MS['name'], Literal(name)))
193     store.add((labeluri, MS['sortname'], Literal(sortname)))
194
195 def setupGraph():
196     # setup graph
197     store = Graph()
198     store.bind('ms', MS)
199     store.bind('gmm', GMM)
200
201     return store
202
203 if __name__ == '__main__':
204     # options
205     rdf_filename = sys.argv[1]
206     processing_type = sys.argv[2]
207     filenames = sys.argv[3:]
208
209     if processing_type != 'all':
210         store = setupGraph()
211     else:
212         stores = [setupGraph() for scheme in ['simple', 'flat', 'deep']]
213
214     for filename in filenames:
215         print("Processing %s..." % filename, file=sys.stderr)
```

```
216         # pull out tags into a dict
217         def split_term(term):
218             parts = term.split('=')
219             return (parts[0], "=".join(parts[1:]).decode('utf-8'))
220 
221         metaflac = check_output(['metaflac', '--export-tags-to', '-', filename])
222         tags_raw = map(split_term, metaflac.split('\n'))
223         tags = {}
224         for tag in flat_tags:
225             lst = []
226             for item in tags_raw:
227                 if item[0] == tag:
228                     lst.append(item[1])
229             tags[tag] = lst
230 
231         # process
232         if processing_type == 'simple':
233             process_simple(store, filename, tags)
234         elif processing_type == 'flat':
235             process_flat(store, filename, tags)
236         elif processing_type == 'deep':
237             process_deep(store, filename, tags)
238         elif processing_type == 'all':
239             process_simple(stores[0], filename, tags)
240             process_flat(stores[1], filename, tags)
241             process_deep(stores[2], filename, tags)
242         else:
243             raise Exception('unimplemented')
244 
245 
246     if processing_type == 'deep' or processing_type == 'all':
247         print("Postprocessing for 'deep'...", file=sys.stderr)
248         try:
249             postprocess_deep(store, tags)
250         except:
251             postprocess_deep(stores[2], tags)
252 
253     # output
```

```
254     if rdf_filename[-3:] == 'xml':
255         ser_format = 'pretty-xml'
256     elif rdf_filename[-3:] == 'ntt':
257         ser_format = 'nt'
258     else:
259         ser_format="turtle"
260 
261     if rdf_filename[0] == '-' and processing_type != 'all':
262         print(store.serialize(format=ser_format))
263     elif processing_type != 'all':
264         store.serialize(rdf_filename, format=ser_format)
265     elif processing_type == 'all':
266         schemes = ['simple', 'flat', 'deep']
267         [stores[n].serialize(rdf_filename % schemes[n], format=ser_format) for
268                 n in range(0,3)]
```