Metagit: Decentralised Metadata Management With Git

Mark Scott[a,*], Steven J. Johnston[a], Simon J. Cox[a]

[a]*Faculty of Engineering and the Environment, University of Southampton, SO16 7QF, United Kingdom*

Abstract

Version control systems are used widely for tracking edits to data files, especially when working in teams with simultaneous editors. They track who made edits and when, and provide tools for comparing changes made and for resolving conflicts when the same file is edited by two people. Attaching metadata to files which stays with the data and tracking changes can be difficult if the file format does not support it. In this paper we present a new and innovative architecture for recording key-value metadata for objects in a revision control system, specifically Git. The utility and maturity of version control system tools make them a good candidate for a metadata store. We take advantage of Git's data store to permit the values to be blobs, opening up other possibilities such as defining thumbnails for files and folders. Propagation rules are presented, detailing when metadata follows a file after modifications to the repository. A prototype version of a tool is presented and the usefulness of the architecture is demonstrated with a number of examples. Adapting the approach to other systems such as Mercurial and Subversion is also discussed.

Keywords: Metadata, Data sharing, Digital libraries, Knowledge management applications

1. Introduction

With the huge amount of data being produced by many disciplines the need for accurate and up-to-date metadata is well documented [1, 2, 3]. It is therefore important to manage it using appropriate methods. This paper shows how systems such as Git can be used to provide this metadata management.

Metadata is data used to describe other data, for example the owner and creation date of a file. It is useful for categorising data to help with its organisation and management.

Metadata is sometimes stored inside a data file to provide pertinent information about its data (embedded metadata [4]). For example, Microsoft Word documents store the author's name, company and creation/modification date of a document [5]. Portable Document Format (PDF) files [6] use two types of metadata: a key-value dictionary and an XMP compliant metadata stream stored as XML [7].

Not all file formats support metadata, and modification of the metadata in those that do changes the data file itself making it difficult to identify whether it was the data or metadata that changed.

*Corresponding author
Email address: Mark.Scott@soton.ac.uk (Mark Scott)

Archival repositories expect metadata to accompany the item being deposited. Dublin Core [8] is one of the more commonly used metadata standards but, depending on the discipline, others exist. In a repository, metadata about the files that have been uploaded is stored separately from the data file (associated metadata [4]). This makes it easier to track changes to metadata and data as they are independent, but metadata is lost when the file is downloaded.

A reliable method is needed to create metadata for a single file or a dataset (collection of files) which can be kept with the data when sharing it with other users or uploading to a central repository. Any changes to the metadata should be tracked to give a historical record of the metadata.

Version control systems provide features for tracking changes made to data files. Each time a change is made to a file, it can be committed into the repository with a description (commit message). The history of a file is then retained for future reference. Among the many version control systems that exist, the most popular are CVS, Subversion, Git and Mercurial [9]. CVS and Subversion use a centralised architecture, where a commit is made to a central repository for others to see immediately. Git and Mercurial are *distributed revision control systems* (DRCSs), which use a distributed model where commits happen locally. The commits can then be pushed to any other copy of the repository. One copy is usually nominated as a central place that others will push to and pull from.

There are many features that version control systems provide: the historical record of an individual file, the ability to compare different versions of a file, being able to revert files to a previous state, and robust tools for the exchange of information between users of a repository and for resolving conflicts when the same file is edited by different people.

Many of these features have been targeted at the modification and exchange of data files, but metadata goes through regular modification too, and these features could be usefully applied to the tracking of metadata.

In this paper, we demonstrate this innovative approach by creating such a system within Git. Git provides the ability to track changes and has powerful sharing capabilities, allowing changes to metadata to be exchanged with a central repository and other users. Branches make it possible to separate data and metadata so metadata commits can be made independently of data commits, and using a DRCS means that modifications are at first made on the user's local system.

The paper has the following structure. In Section 2 we discuss related work and then in Section 3 we present the metadata storage architecture and propagation rules (necessary when data is modified). A tool, which we have called Metagit, that implements the architecture is also described. Section 4 gives examples of its use and Section 5 shows some alternative models that were investigated. Section 6 provides a discussion of the architecture. The paper then concludes and suggests some future work.

No new data was created during this study. The source code for the proof of concept tools is available from the University of Southampton at `http://dx.doi.org/10.5258/SOTON/393614` [10].

2. Related Work

Current approaches to metadata storage fall into two categories: metadata held centrally in a repository and metadata stored locally. The metadata can be embedded (stored as part of the data), associated (stored in separate files held together with the data), or third-party (metadata held in a separate repository not controlled by the organisation hosting the data) [4].

When storing data locally, a user will commonly keep it in a file system – a tree-based data storage mechanism for unstructured content (data with no specified structure). A number of ways exist to create local associated metadata in modern file systems including file attributes, multiple data streams per file and file systems based on relational databases [11]. There is a tight relationship between the data, metadata and the file system which means that metadata can be lost if the data is transferred to a file system that does not support the same features.

Key-value stores are useful for defining metadata. Databases can hold key-value data [12], often using an RDF triple store [13] and queried with SPARQL [14], a language used for querying RDF graphs. NoSQL stores [15, 16] such as MongoDB [17] are also useful for this type of data. These solutions can be used locally, or used to support a centralised solution.

Local associated metadata can even be created by storing it in separate data files, such as in a spreadsheet.

Data can also be stored on a server, accessed using a network file system. Any features employed to support metadata, such as the multiple data streams used with local file systems must be supported by the client, the network protocol being used, and the server's local file system. This can make it very easy to lose the metadata if the user does not check for support before copying. Some work has been done to support metadata in repositories and file servers with the SemDAV project [11] which extends the WebDAV protocol, supported by a large number of existing operating systems, to allow metadata to be added and manipulated on data files. The SemDAV server communicates data and metadata to clients using a SemDAV Browser or API, but backwards compatibility to WebDAV clients still using the standard WebDAV protocol is also provided for access to the data only. To users of the SemDAV system, data and metadata would be kept together but, behind the scenes, they would be separated into a file system and database. This is a platform independant solution, but moves data storage to a repository.

Key-value stores are commonly used in cloud computing scenarios to deal with huge quantities of data; Amazon's SimpleDB and DynamoDB [18, 19], Microsoft's Azure Tables [20] and Google's AppEngine Datastore [21] are cloud database services that permit scaling across data centers. These are centralised solutions by nature. Enterprise metadata solutions like SAS Metadata Server also use a centralised approach. Exploiting RDF and SPARQL in cloud environments has been explored [22].

The 'Semantic Web' describes the World Wide Web Consortium's (W3C) vision of using the internet to share structured data via standard formats and protocols. Controlled vocabularies, accurate recording of

relationships between datasets – and the meaning of those relationships – and a common framework that can be understood by computers has the potential to allow data exchange across disciplines and institutions. Current philosophy encourages data stores to be semantically enabled using RDF which can then be explored. Noteworthy semantic tools include the Haystack Semantic Web Browser [23] and the SEMEX Semantic Media Explorer [24]. Fortunata [25] is a wiki-based framework that simplifies the creation of semantic-enabled web applications which can often involve a lot of effort and complexity to set up and manage. Other tools are explored by Groza et al. [26].

Centralised data repositories do exist [27], usually discipline specific; Data Dryad [28] is a data archival repository for data files associated with published articles in the sciences or medicine, and figshare [29] provides a similar service. DataCite [30] enables organisations to register research datasets and assign persistent identifiers.

There are a number of disadvantages with the approaches discussed:

- **Portability:** When leveraging file system features, metadata can be lost when copying.
- **Centralised or localised:** Data does not easily transfer between a centralised and a localised approach. Centralised designs make it difficult for users to manage their own metadata whilst a localised architecture means it cannot easily be shared.
- **Lack of data provenance:** Showing the history of changes and who made them can be a required feature for a data repository. This can be more difficult to control when metadata is stored in a localised solution.

A hybrid, decentralised approach where metadata can be held in a repository, but users can edit their own metadata locally without a repository seems the next logical step in metadata management. This decentralised approach has been used in database management systems: collaborative data sharing systems (CDSSs) allow for separate database instances to be maintained with modifications made locally and then exchanged with another CDSS instance [31, 32].

3. Design and implementation

In this section, we present a novel architecture for the storage of key-value metadata using Git. This allows users to edit metadata in a decentralised way, only pushing to a central Git repository when required.

The metadata will be held in its own branch in order to leave the user's data untouched. Metadata is addressed using the combination of object path and commit ID; multiple named metadata streams are permitted.

Metadata needs to be carried forward with a file to later commits when its contents have not been modified, but should not carry back to earlier commits. More detail on these metadata propagation rules is discussed in Section 3.3.

3.1. Introduction to the Git features used

This architecture utilises Git blobs, trees, references and commits for the storage of metadata for data in a Git repository. Git blobs are used to hold the metadata and, due to the flexibility of Git, these blobs can be anything from a thumbnail, to a JSON or XML document, to a simple string containing a metadata value. Blobs have no name in Git, just an object ID (SHA-1 hash), so trees allow the blob to be found using a name: trees are objects that contain a list of blob IDs along with a name for those blobs. They can also contain other tree IDs giving names to the nested tree objects, and this nesting creates a path to a blob.

Commits are used to record the state of a tree at a particular point in time (with a commit message and the committer). References, which point to a particular commit, are then used to create a branch which effectively gives a name to a particular commit. When Git saves a new commit, the previous commit is linked to the new one to give a history of changes; the reference is also updated to point to the latest commit.

Git creates and manages these objects in its data store, held in a directory called `.git`. The `.git/refs` directory is used to store references to commit object IDs: a reference file containing the SHA-1 hash of the commit can be given to Git instead of the ID. References in the `.git/refs/heads` directory create named branches that Git tools will display, and references elsewhere under `.git/refs` such as `.git/refs/metadata` would not be displayed as a branch but can still be looked up by specifying their path explicitly. This is useful to hide the metadata-related references from tools being used to browse data.

3.2. Git tree structure

In this section, we discuss the storage of metadata using a single branch in `.git/refs/heads/metadata`. Metadata is stored as a blob in Git with the following path structure:

```
metadata:[path]/(metadata)/[stream name]/[commit ID]
```

In this path, `metadata:` tells Git to look on the `metadata` branch, `[path]` represents the data object's path in the data branch that is specified with `[commit ID]`. `[path]` and `[commit ID]` uniquely identify the object to which the metadata refers. Other models for addressing and storing metadata are explored in Section 5.

The `(metadata)` part of the path is replaced with a string that has a low likelihood of colliding with the user's data. Using 'metadata' would not be sensible as the user may also have a file or folder with that name, breaking the tool. Hashes and UUIDs present useful ways of avoiding conflicts but, whatever name is chosen, it needs to be easy to reproduce. The proof of concept tool used a UUID to distinguish it

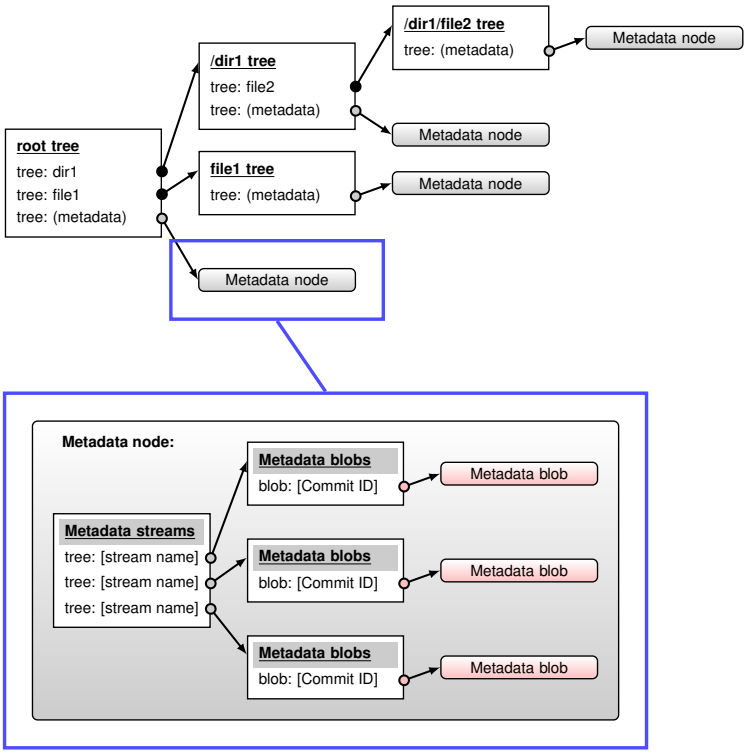

Figure 1: Organisation of an example metadata repository showing files and folders and their associated metadata. The inset shows the structure of a metadata node.

visually from the hashes used for commit IDs and, because it is perhaps more appropriate than a hash as the intended use of a UUID is to stand as an identifier for information.

UUIDs can be random, or alternatively generated using a fixed string and namespace to allow them to be reproducible. The UUID we chose was generated using 'metadata' and the X.500 namespace, producing the following string:

```
92df1d6a-b6da-5ddb-9055-44349d03203e
```

`[stream name]` permits multiple metadata blobs per path. For example, one metadata stream could be a blob containing a text value, another a JSON document with key-value data and one metadata stream could contain a thumbnail for the file. Examples of all three will be shown in Section 4. The stream name-value combination provides the key-value capability of the store.

We put the metadata stream name before the commit ID to keep different versions of the same metadata stream together; putting the stream name after the commit ID would make it less obvious what streams exist in the repository for a particular path.

Figure 1 summarises the organisation of an example metadata repository showing files and folders and

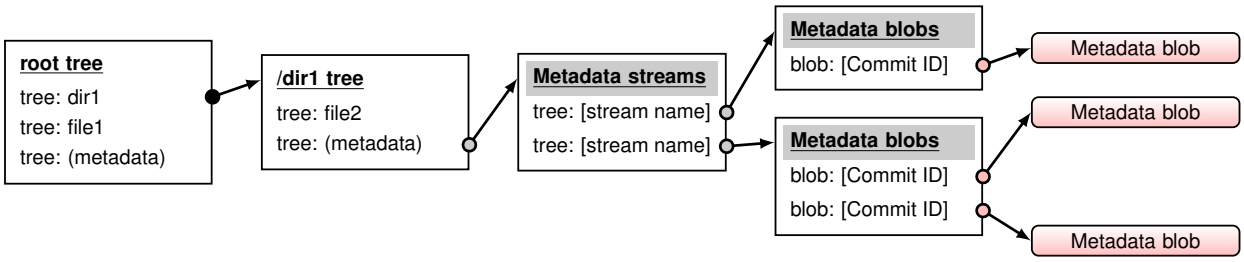

Figure 2: Structure of a metadata node for a single directory at path `/dir1`

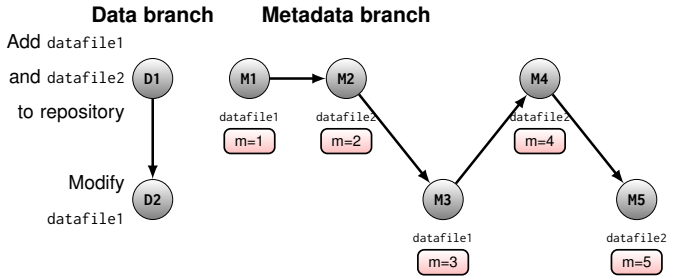

Figure 3: Example of metadata being added to files in different commits.

their associated metadata. Each file and folder in the figure has a `(metadata)` entry pointing to a metadata node – see inset. A metadata node contains a Git tree with a list of stream names pointing to Git trees with commit IDs which then point to the particular metadata blob for that commit.

Figure 2 gives an example of metadata for a single directory, showing two metadata streams with one of the metadata streams containing metadata blobs for two commits.

3.3. Metadata propagation

The structure defined above only ties metadata to a single data commit. In order to build a more effective tool, a set of propagation rules is required in order for metadata from earlier commits to be viewed. This ensures metadata is not lost each time a data modification is made.

For example, Figure 3 shows a repository with the files `datafile1` and `datafile2` and associated metadata stored with the commit `D1`. In reality, Git would use a SHA-1 hash for the ID. Updating `datafile1` creates a new commit with a new ID, e.g. `D2`. Metadata for `datafile1` or `datafile2` stored under the commit ID `D1` will no longer be accessible using the new ID of `D2`, and just retrieving `D1`'s metadata may not be appropriate because `datafile1` has now been modified.

It would be a reasonable expectation that `datafile2` – which was unchanged – should keep its metadata: metadata at earlier commits should carry forward on unaltered files. It should also be possible to modify the metadata on `datafile2` on commit `D1` where it was added, or to override the earlier definition on commit `D2`.

Metadata added at later commits should not carry backwards to previous commits. One reason for this is because branching makes this impractical, unpredictable and confusing. If the metadata were to carry backwards to a file created before the branch, it must also carry forward from that earlier point to where the file exists on the other branches because of the previously defined behaviour. This may not be what the user intended, e.g. if the user was adding metadata on a production branch but was wanting different metadata on a debug branch. If the 'forwards' propagation rule was ignored in this instance and metadata only affected the earlier commit but not the other branches, confusion might arise as to why metadata was visible on one branch and on its earlier commit but not on the other branch. If both branches had their own metadata it would be difficult to resolve which set of metadata would apply to the earlier commit. Therefore, if metadata is to be changed for a file on an earlier commit, the user is expected to manually edit the metadata on the earlier commit. This will then behave more naturally to the user who will know that metadata will propagate to all branches defined after that commit.

This establishes the following metadata propagation rules:

- Metadata should reside with a single commit
- Metadata should be capable of propagating to later data commits
- Metadata should not affect previous versions of a file
- Adding metadata to an earlier data commit should not override metadata already defined on a later commit
- Metadata should not automatically carry forward on a file when its contents have been modified
- Branching does not affect metadata propagation and metadata should propagate to all branches equally
- Merging of branches should stop metadata from propagating because it is unclear which branch's metadata will apply to each file

3.4. Tool behaviour and syntax

A command line tool was built storing metadata using the model introduced in Section 3.2 and following the propagation rules explained in Section 3.3. It included the following metadata functions:

- `get`
- `set`
- `delete`
- `copy`

- and a number of exploratory functions:
  - `ls`: lists all files in a repository and whether metadata is defined
  - `log`: lists commits for a path and the streams defined
  - `list`: lists all versions of a specified metadata stream defined on a file

The tool is executed by typing `m` followed by the function name and any parameters. `m` is 'Metagit' or 'metadata' abbreviated for convenience.

Each function of the tool needs the following information:

- A path to a file or directory
- The commit ID of the data – for locating the matching metadata
- Whether searching back to previous commits is permitted – for when metadata has not been defined for the path at the specified commit
- The metadata stream

To simplify the syntax, all of these except whether searching back is permitted can have default values: the commit can default to `HEAD` (the current revision of the working directory), the path can default to the current directory, and a default stream (`metadata`) can be used if not specified. When a user has not specified a commit but the file itself has not been committed, the tool refuses to continue for a `get` and a `set` because it is not possible to manipulate metadata on unsaved data.

The tool uses a binary flag, `s`, to specify whether searching back is permitted: the user can specify `s+` to enable searching and `s-` to disable searching. The effect of these options is shown in Table 1. The eventual commit that will be used for metadata can be affected so radically by this parameter that it was decided to make it a required parameter.

Combining all components into a single parameter separated by the colon makes commands easier to read by reducing the number of switches that would be required for all these parameters, particularly for the `copy` function which will require six pieces of information. No colon means that we assume the user has specified a path, one colon means that we assume the user has specified a revision and a path, and two colons means the user has specified all three parts of the path.

This permits the following `m get` examples for retrieving metadata:

- Use revision `3a8f9af`, no searching back:

  ```
  m get s-3a8f9af:file1:stream2
  ```

Table 1: The effect of `s+` and `s-` on the behaviour of the tool

Operation	**Metadata**	`s+`	`s-`
`get`	Exists	Return metadata	Return metadata
`get`	Does not exist	Look for metadata in parents	Return nothing
`set` [1]	Exists	Set metadata in specified commit/HEAD	Set metadata in specified commit/HEAD
`set` [1]	Does not exist	Look in parents for earlier commit with metadata and set, or create new metadata at original commit	Set metadata in specified commit/HEAD
`list` [2]	n/a	All metadata is listed for path. The revision is only used by the tool to lookup the object for any defined metadata and report whether it is the same file as the specified revision.	
`delete`	Exists	Deletes metadata	Deletes metadata from specified commit/HEAD if it exists
`delete`	Does not exist	Deletes metadata from first commit in history that matches path	Does nothing
`copy`	Source must exist	Finds source metadata and destination commit using same rules as `get` and `set`	Copies metadata from specified version to specified version

[1] If commit is not specified, the path specified in working directory must have been committed

[2] If commit is not specified, the tool used the contents of the working directory to compare against defined metadata

- Use revision `3a8f9af` or start searching back from it:

```
m get s+3a8f9af:file1:stream2
```

- No stream – assume default stream (don't search back):

```
m get s-HEAD:file1
```

- No revision – assume HEAD (search back):

```
m get s+:file1:stream2
```

- Just a path – assume HEAD and default stream (don't search back):

```
m get s-:file1
```

- No path – assume HEAD and working directory (search back):

```
m get s+
```

- Just a revision – assume working directory (don't search back):

```
m get s-3a8f9af:
```

4. Example Usage

This section gives some examples of the tool in use with examples showing a key-value pair feature possible with the tool, a developer using the key-value pair functionality to set metadata on files in multiple branches, a user searching for files with metadata and a user adding thumbnails to their files.

4.1. Key-value pair metadata

The storage of metadata as blobs within Git with multiple 'streams' for each file provides a flexible base and a simple method of storing key-value metadata: the stream name is the key and the stream's blob contains the value.

For example, `s-:file1:authorsurname` and `s-:file1:address` can have their blobs set to data containing 'Smith' and 'London' as follows:

```
printf Smith | m set s-:file1:authorsurname -
printf London | m set s-:file1:address -
```

These commands pipe the output of the `printf` command into the `m` tool which has been instructed to take its input from standard input with the `-` option (otherwise it would expect a file name). The key-value pairs `authorsurname=Smith` and `address=London` are set on `file1` at the `HEAD` commit.

Blobs can be other types of file, not just text data, so the `thumbnail` stream of `file1` can be set to a PNG image file as follows:

```
m set s-file1:thumbnail /path/to/thumbnail.png
```

4.1.1. Key-value pair metadata in JSON

Encoding metadata as JSON and using a metadata stream to store the JSON blob is possible and demonstrates the framework's flexibility. It would give access to the limited data types supported by JSON and, because each edit results in a new commit to the metadata branch, changes to metadata entries can be compared with Git diff tools because the JSON is stored as a text file (which would be more difficult if key-value metadata was encoded using a binary file format). JSON is also well supported so this would export to other systems.

The command-line tool discussed so far is divided into two components: a library that can store or retrieve data from the metadata branch and a front-end component for parsing the parameters passed and calling the appropriate library code. The front-end component can parse key-value pairs and generate JSON blobs which are then stored as a metadata stream using the library.

For example, the following commands:

```
m setvalue s-:file1:metadata authorsurname=Smith
m setvalue s-:file1:metadata address=London
```

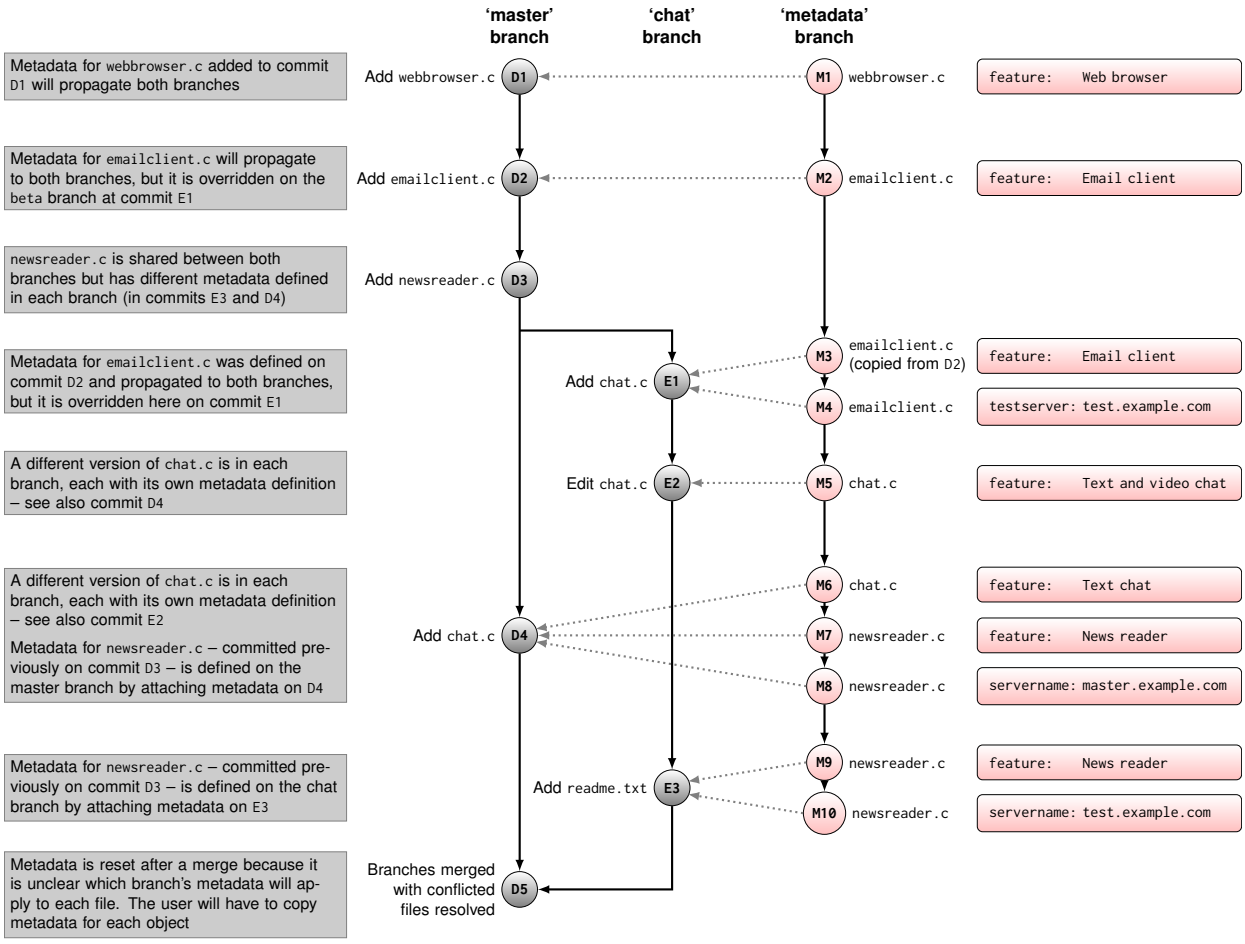

Figure 4: Commits in a repository along with commits to a metadata branch

generate the JSON blob below, attached to the `metadata` stream of `file1` on the `HEAD` revision:

```
{"authorsurname": "Smith", "address": "London"}
```

4.2. Developer with multiple branches

Revision control systems such as Git use branches to allow different versions of a file or files to be edited in parallel – useful in software development when trying to introduce new features whilst still fixing bugs in previous versions. For example, a developer can maintain two branches in a Git repository, one with a set of code used in production and one with a version of the code intended for ongoing development. This example is illustrated in Figure 4.

The developer starts by initialising a Git repository and adding three files to it:

```
git init
```

```
vi webbrowser.c
git add webbrowser.c
git commit webbrowser.c -m "Add webbrowser.c"

vi emailclient.c
git add emailclient.c
git commit emailclient.c -m "Add emailclient.c"

vi newsreader.c
git add newsreader.c
git commit newsreader.c -m "Add newsreader.c"
```

Then metadata can be added to the `webbrowser.c` and `emailclient.c` files, using the JSON blob key-value approach:

```
m setvalue s+:webbrowser.c feature="Web browser"
m setvalue s+:emailclient.c feature="Email client"
```

The `s+` allows searching back for the commit where the file was added (the first commit for `webbrowser.c` and the second commit for `emailclient.c`).

At this point, the developer decides to branch the repository and commits `chat.c` to the repository and makes an edit to `chat.c`:

```
git branch chat
git checkout chat

vi chat.c
git add chat.c
git commit chat.c -m "Add chat.c"

vi chat.c
git add chat.c
git commit chat.c -m "Edit chat.c"
```

The developer decides that the text chat feature of the new `chat.c` file is important enough to release into production but the video chat still requires some work, so switches back to the master branch and adds another file named `chat.c` with the text chatting code, and then switches to the chat branch to make a

comment about what is still to be done regarding video chat in a `readme.txt` file:

```
git checkout master

vi chat.c
git add chat.c
git commit chat.c -m "Add chat.c"

git checkout chat

vi readme.txt
git add readme.txt
git commit readme.txt -m "Add readme.txt"
```

Branching can give rise to various situations with respect to metadata, as illustrated in Figure 4:

- **The same file exists in both branches with shared metadata.** `webbrowser.c` is an example of this.
- **Overriding shared metadata to make it distinct on one of the branches.** The same file exists in both branches with shared metadata, but then the shared metadata is overridden to make it distinct on one of the branches. The user may copy the existing metadata first before modifying or may just create new metadata.
- **Different files with the same path exist in different branches.**
- **The same file exists in both branches with distinct metadata.**
- **Merging of branches.**

4.2.1. The same file exists in both branches with shared metadata.

`webbrowser.c` is an example of this. It was committed and its metadata added before the branch. Data commits on both branches will retrieve the same metadata (unless overridden).

4.2.2. Overriding shared metadata to make it distinct on one of the branches

For example, new metadata is required on the `emailclient.c` file on the chat branch, which already has metadata on it set before the branch was made.

Optionally, the old metadata can be copied which is useful if only one parameter is to be overridden but the rest will remain unedited:

```
git checkout chat
m copy s+HEAD:emailclient.c:metadata s-HEAD^^:emailclient.c:metadata
```

The `s+` used for the source lets the tool search back for the original metadata and the `s-` used for the destination stops metadata being added on earlier commits. `HEAD^^` is Git syntax to use the commit two commits earlier than the `HEAD` commit. Instead of `HEAD^^`, the developer could instead look up the exact revision using Git tools (e.g. `git reflog`).

Then the metadata on `emailclient.c` can be overridden on the chat branch:

```
m setvalue s+:emailclient.c testserver=test.example.com
```

The copy created a new metadata blob on the `HEAD` which the `s+` option will automatically find.

4.2.3. Different files with the same path exist in different branches

Metadata can be added to the `chat.c` file on the chat branch:

```
git checkout chat
m setvalue s+:chat.c feature="Text and video chat"
```

`s+` can be used to find the commit where the file was added (`HEAD^` on the chat branch). Then the developer adds branch-specific metadata to `chat.c` on the master branch:

```
git checkout master
m setvalue s+:chat.c feature="Text chat"
```

`s+` can be used for `chat.c` because it was just created.

4.2.4. The same file exists in both branches with distinct metadata

The `newsreader.c` file was created before the branch but currently has no metadata.

Metadata can be added to the commits on the master branch which will not be visible on the chat branch:

```
git checkout master
m setvalue s-:newsreader.c feature="News reader"
m setvalue s-:newsreader.c servername="master.example.com"
```

The `s-` stops metadata being added on earlier commits, so this will be attached to the `HEAD` commit because no commit has been specified. No search back is required because we want the metadata distinct to this branch. The developer can define different metadata on the chat branch:

```
git checkout chat
m setvalue s-:newsreader.c feature="News reader"
m setvalue s-:newsreader.c servername="test.example.com"
```

4.2.5. Merging of branches

When working with branches, it is common behaviour for branches to be merged, e.g. when beta features are to be moved into production. This is done with the `merge` command:

```
git checkout master
git merge chat
```

This combines the files in both branches, taking the latest version of any files, but files that have been modified in both branches will produce conflicts that have to be resolved manually (such as `chat.c` in the example):

```
# Resolve conflict
vi chat.c
git add chat.c
git commit -m "Merged chat branch"
```

Merging creates a problem with metadata propagation because it is unclear which branch's metadata will apply to each file. For example, it is not possible to automatically determine the metadata for the file `chat.c` because it had different definitions on each branch. Due to this uncertainty, metadata is not brought forward after a merge.

It can still be listed or retrieved for the commits before the merge and can be copied manually. The following command creates new metadata on a merged file by looking backwards on the chat branch for an existing `metadata` stream (`s+` was specified) and copying it to the head of the merged master branch:

```
m copy s+chat:chat.c:metadata s-master:chat.c:metadata
```

4.3. Searching for files with metadata

Metadata stored against individual files and folders can be retrieved using the `m` tool with the `get` command. This becomes more powerful when combined with other tools. For example, the exit status of

the tool is 0 on success and >0 on error so a script can be used to walk a directory tree and print out the names of files that contain metadata:

```
#!/bin/bash
for f in $(find repo_path/papers -name \*.txt)
do
  authorname=$(m get s+HEAD:$f:authorname)
  if (($?==0)); then
    echo $f, $authorname
  fi
done > textfileswithauthors.csv
```

The bash script above uses the Unix `find` command to search for files with the `txt` extension and retrieve the `authorname` stream. If the stream exists, the name of the file and the metadata stream are printed to standard out, separated by a comma which is redirected to a file.

4.4. Thumbnails

The examples so far have only shown the storage of small amounts of textual metadata but Git also supports other file types. A metadata stream could be created to give a file or folder a thumbnail.

```
convert -thumbnail 100 /pathtoimages/chat-architecture.png - | m set s+:
   chat.c:thumbnail -
```

This command uses the `convert` command from ImageMagick to create a 100 pixel wide thumbnail from a full sized image. The result is written to standard out which is piped into the `m` tool to write it to the `thumbnail` stream of the `feature1.c` file.

When combined with the `find` command, thumbnails can be created for all images in a dataset:

```
#!/bin/bash
for f in $(find repo_path/images -name \*.png)
do
  convert -thumbnail 100 $f - | m set s+:$f:thumbnail -
done
```

The tool's ability to accept metadata from standard in means that it can be chained together in a pipeline and accept data from interesting sources. For example, an image from the internet can be retrieved using the `wget` tool (in this case a woodpecker from Wikimedia), and piped into the `convert` command to create a thumbnail, which can then be piped to the tool to save it as a metadata stream, as shown in the following

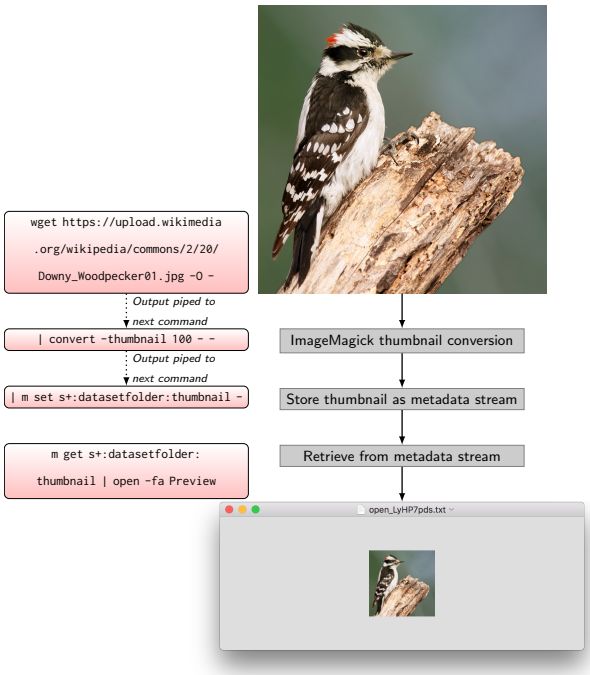

Figure 5: Retrieving image from internet and storing its thumbnail as a metadata stream

command and illustrated in Figure 5. The `-` option is used in the place of filenames to indicate the pipeline should be used rather than reading or writing to a file:

```
wget https://upload.wikimedia.org/wikipedia/commons/2/20/
    Downy_Woodpecker01.jpg -O - \
 | convert -thumbnail 100 - - \
 | m set s+:datasetfolder:thumbnail -
```

Retrieving the thumbnail with the `m` tool can be done with the `get` instruction, redirecting the output to a file or piping it to another application; in this example it is piped to the Preview application on a Mac to directly display it without saving:

```
m get s+:datasetfolder:thumbnail | open -fa Preview
```

5. Suitable Git models

So far, we have only shown metadata storage using a single branch, addressed with the object path and its commit ID. A number of alternative models are now discussed. Terminology used in this section was introduced in Section 3.1.

One approach to recording metadata is to mirror the data branch exactly in the metadata branch and locate metadata using the path of the file: on the metadata branch the requested path would contain the blob's metadata rather than the blob itself.

While this would give metadata history for each file there would be no link between a version of a file and its metadata – without some link to the version of the file, different versions of the same file will always have the same metadata (which is the approach taken by Subversion). One solution to this is to use the data object's ID (a SHA-1 hash), rather than its path, to locate a file's metadata, but Git repositories can have multiple branches so there is no guarantee that metadata for a file with a particular ID is for that branch's file. Using the commit ID as well as the object ID might solve that problem but identical files in a commit share object IDs so would therefore share each other's metadata.

Using the commit ID and the object's path as the key to the metadata keeps metadata with a particular version of a file, and using the path rather than the object ID ensures bit-for-bit identical files at different paths can have differing metadata.

We investigated six models for adding metadata to a data branch which all use the data commit's ID and the item's path as the metadata key. The six models are based upon the following three designs, each of which has two variations: the object path or the data commit ID as the first part of the key. These six models are summarised in Table 2.

1. **Use Git objects to store the commit ID/path key:**

   Use a single branch in `.git/refs/heads` with the path of the item and the commit ID being stored in Git;

2. **Create a reference file in `.git/refs/metadata` for each path/commit ID key combination:**

   Use the `.git/refs/metadata` directory to recreate the data branch's folder structure along with the commit ID which will lead to a reference file pointing to where metadata can be found;

3. **Use a hybrid solution:**

   Use `.git/refs/metadata` to store the first part of the key and Git objects to store the second part of the key.

Of these, the three models that use `.git/refs/metadata` to store the path part of the key (models 3, 4 and 5) create branches that track metadata per object which is shown in Figure 6. The others define metadata in one big tree in Git, either by commit as illustrated in Figure 7, or repository as shown in Figure 8.

All three figures have a data branch containing commits `D1`–`D4` and define metadata on `dir1/file1` and `dir1`. The latest metadata from commit `D1` propagates to commit `D2` because it has no metadata of its own

Table 2: Designs for using Git for metadata storage.

	Reference file (on disk)	Git tree structure	Number of metadata branches	Parent commit saved	Type
1*	`refs/heads/metadata`	`[path]/[commit ID]`	Per repository	No	Repository
2	`refs/heads/metadata`	`[commit ID]/[path]`	Per repository	No	Repository
3	`refs/metadata/[path]/[commit ID]`	—	Per commit/object combination	Yes	Object
4	`refs/metadata/[commit ID]/[path]`	—	Per commit/object combination	Yes	Object
5	`refs/metadata/[path]`	`[commit ID]`	Per object	No	Object
6	`refs/metadata/[commit ID]`	`[path]`	Per commit	If no propagation†	Commit

* Recommended model because it is easy to push (only one branch), more performant than 6 and metadata propagation for a single object is easy to delete.

† If using propagation rules, the creation of an override starts a new set of history with no relationship to the previous commit. If not using propagation rules, recording the parent commit makes sense.

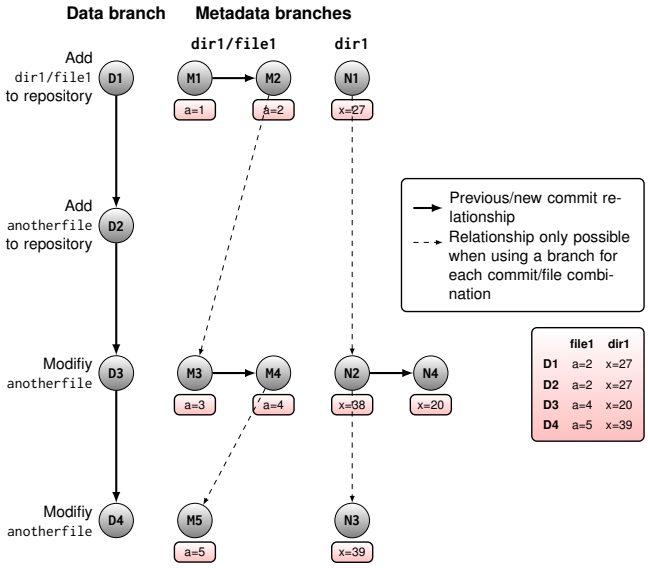

Figure 6: Metadata defined with metadata per object. The dotted line illustrates a link to a parent commit which is not possible in all scenarios.

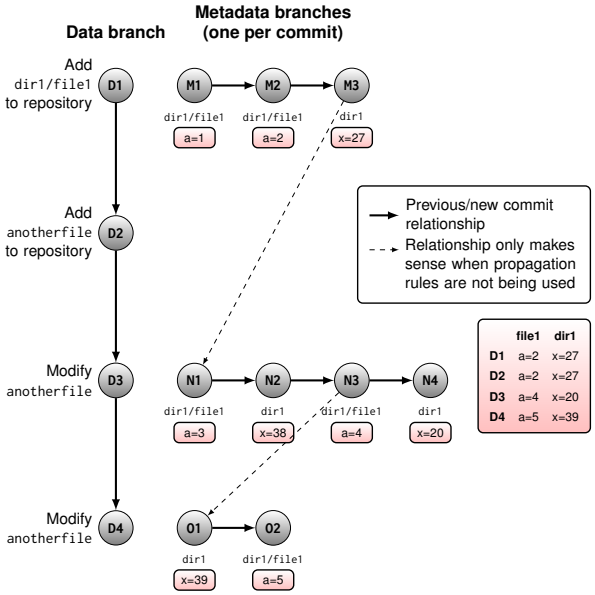

Figure 7: Metadata defined with a metadata branch for each commit. The dotted line illustrates a link to a parent commit which is not always possible.

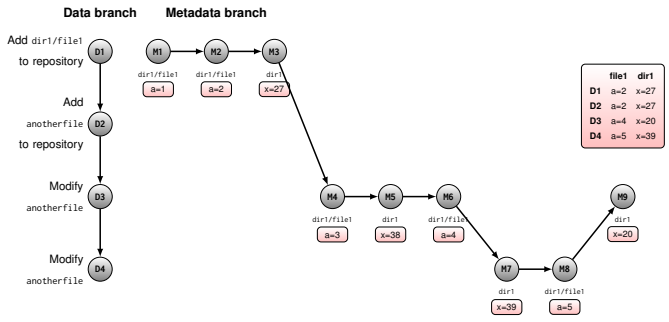

Figure 8: Metadata defined in one big tree for all objects in the dataset.

defined. `D3` and `D4` define their own metadata for `dir/file1` and `dir1`, overriding the metadata on earlier commits. Solid lines indicate the history of commits for that branch. Dotted lines are used to indicate whether the model can record the previous metadata before it was overridden at a later data commit. It can be seen that the value of `dir1` at `D4` was `x=38` before it was changed to `x=39`, and the metadata for `dir1` was changed to `x=20` at `D3` after it was overridden at `D4`.

5.1. Model selection

Consideration must be made as to whether to use the path or the commit ID as the first part of the key. Looking up a path inside Git involves the processing of a Git tree at each level of the path, which is less performant than the same directory structure stored in a file system outside of Git. This performance penalty becomes most apparent when the commit ID is used as the first part of the key: propagation rules mean that every commit in the data's history needs to be checked for the existence of metadata until metadata is found or the first commit is reached.

When the path forms the first part of the key, the path only needs to be looked up once, giving a list of commit IDs containing metadata. This reduces the number of operations required which might prove significant for very long paths. This makes models 2 and 6 less attractive when propagation is required.

The number of Git references created in the repository could also be a factor. Processing of a single Git tree does take time, no matter how small. This would be noticeable if files were nested very deeply or metadata was being retrieved for a large amount of files. Therefore, models 3 and 4 would have a huge advantage in performance because no Git trees would have to be processed to find the metadata for an object. Instead, the directory structure on the file system is used to locate metadata, creating one branch for each commit/object combination. While keeping history for every object's metadata self-contained might seem attractive, it could be very difficult to manage, particularly when pulling and pushing the repository and having to deal with conflicts. Model 5 creates one branch for every object which would suffer from a similar problem.

Therefore, model 1 offers a good compromise between speed of lookup and the number of branches. As it only requires a single metadata branch that can be browsed easily with existing tools it makes pulling, pushing and conflict resolution easier to manage. The head of the branch will have the latest versions of metadata defined for any data commits and, because it stores the path first, Git only needs to traverse the Git trees once to find metadata.

5.2. Metadata history

Models 1, 2 and 5 do not permit the recording of the parent commit when overriding a previous metadata definition on a later commit (to represent the dotted line in Figures 6 and 7).

For example, when metadata is first added to an object in commit `D3`, models 3, 4 and 6 are able to lookup the previous time metadata was added to the object in commit `D1` and record a link to retain this history. It is done by setting the parent commit ID of the new metadata commit to the ID of the last commit of metadata for that data branch. Even if metadata is then changed or the metadata and its history is deleted altogether for commit `D1`, the history for commit `D3` will still exist. Models 1, 2 and 5 do not create a separate branch per data commit and therefore cannot record this parent so will restart a new set of history for each data commit.

The significance of choosing a model without this support may not be too serious. If we want to modify the metadata of a particular version of a file accessible to all data branches then the metadata should really be modified on the data commit where that version of the file was created, and then the history will be complete. Overriding the metadata at a later data commit is only useful when a separate data branch exists. The history is kept intact because the override of metadata is creating a new set of metadata just for that branch so history should start afresh. Therefore, when metadata is edited the default behaviour should be to modify the commit where the file was first added.

This is a difference in semantics, one *extends* by manipulating metadata on a commit with existing metadata and the other *overrides* by creating new metadata at a particular commit hiding existing metadata created at an earlier commit.

When metadata commits must show an exact commit history for a particular object even for overrides then the only way is to create a separate branch for each data commit (or every data commit/object combination). The only models to support this are 3, 4 and 6.

6. Discussion

In this paper we have shown an architecture for the storage of metadata in Git. This is implemented as a tool and is described using examples. One key advantage of this model is the ability to use multiple metadata streams containing any binary data. Metadata streams can be used for simple text values to create a basic key-value store and, since any blob can be used as a metadata stream, it was also shown how JSON can provide an alternative key-value store. This is particularly useful as JSON is well supported by existing tools and gives more portability if metadata export was required. Other metadata types demonstrated were thumbnails. Storing large files in Git increases the size of the repository and can make it slow to transfer repositories so creation of metadata streams containing large blobs requires careful consideration.

This section discusses the architecture and the tool.

6.1. Metadata editing

The use of a single branch in Git means that a user can edit the key-value store through the file system. This allows the architecture to work well with existing tooling. The `git checkout metadata` command

checks out all of the metadata blobs as files which the user can then edit with their preferred tool or by copying a new file over the blob. The modified metadata blob can then be committed with `git commit` and then the repository can be switched back to a data branch with e.g. `git checkout master`.

This helps with resolving any metadata errors and makes it very easy to delete large amounts of metadata in a repository or modify metadata without using the Metagit tool at all (provided the correct folder structure is followed). The `m` tool still works well for reading metadata whilst the metadata branch is checked out, but should currently not be used for editing metadata with `m set` as the working directory is not updated by the tool after the modification.

Due to the ability to push and pull between repositories, two copies of the same repository can exist on a file system, one with the `metadata` branch checked out and one with a data branch checked out. Metadata can be edited separately using the method described in this section and pushed to the other copy of the repository.

6.2. Adapting the model to other systems

We chose Git for this proof of concept tool. Its flexible data store and its popularity made it a good candidate. Other suitable revision control systems exist such as Subversion and Mercurial.

Mercurial has many of the building blocks to build a similar system: in particular, named branches, changesets and manifests. The API allows a *change context* object to be retrieved by branch name or revision number which contains files. A separate branch could be maintained containing the metadata. The named branch would use changesets and manifests that contained the metadata for each file or folder – the same structure used for Git earlier in this paper could also be used.

Although Subversion is not decentralised, it already supports arbitrary metadata against files and directories using properties. These properties use a key-value approach and the value can be any data including binary data. Each metadata edit results in a new commit to the repository so metadata edits are versioned. This means that Subversion already has an approach in place to achieve many of the things we have presented. The differences between Subversion's model and the one presented in this paper are:

- In Subversion, metadata remains with a data file even after its contents are modified. This also means that different versions of a file cannot have different sets of metadata.
- Changes to metadata are done as a commit to the main repository, rather than to a separate branch.

To address the first point, the Git tree structure presented in Section 3.2 could be created in a directory somewhere in the repository, e.g. `branches/metadata`, and then the tool could be adjusted to retrieve its metadata from that folder instead of from a Git repository. This approach would also work for Git and Mercurial. Metadata would end up mixed up with the data files and modifications to the metadata would

require a commit to the data branch rather than a separate metadata branch, but it would be a flexible solution which could even allow metadata to be manipulated on a dataset when no revision control system was being used.

6.3. Metadata integrity

When a user updates an item of metadata a new commit is created recording what changes were made. This provides a record of the history of metadata in the repository, but in Git it is possible to rewrite history. This may not be desirable when a record of metadata history is critical. Git configuration settings such as `receive.denyNonFastForwards` and `receive.denyDeletes` can be used to stop history being rewritten, and the approach taken by GitLab and GitHub systems permit branches to be protected so only trusted users can manipulate those. Users who want to send the changes they have made to the repository need to do this with a *pull request* so the changes can be checked before being integrated. These measures help to protect the data and history in central repository.

Metadata is particularly relevant to digital libraries and being able to show data provenance would improve the trustworthiness of the metadata. Git permits commits to be signed using a GNU Privacy Guard (GPG) private key, providing some guarantee of who made the commit. Anyone changing someone else's metadata would have to obtain and unlock the private key first. This feature was not implemented in the proof of concept tool because the pygit2 API did not support this function – other APIs did, e.g. GitPython.

6.4. Syntax

Specifying the `s+` or `s-` every time, to indicate whether to search back for earlier metadata, adds a level of complication that may deter users, but automatically assuming `s+` or `s-` may not be the correct approach either. Behaving as `s-` all of the time would simplify the tool's syntax, but it would increase the number of steps required to modify metadata because the user would be expected to look up the commit's ID each time using the `list`, `ls` and `log` functions.

Consider the commits to the repository shown in Figure 9. When metadata is added to a file, the user needs to know which commit to use for the metadata. Similarly, when retrieving metadata, the user needs to know which commit's metadata is being returned.

Specifying the commit each time makes the tool difficult to use because the user must first look up the commit before they can add metadata. It would ensure the metadata is applied to the correct commit but makes the tool less user friendly.

Always assuming the current commit is dangerous for setting metadata because metadata from earlier commits would be overridden and metadata would not be visible on earlier commits with the same file.

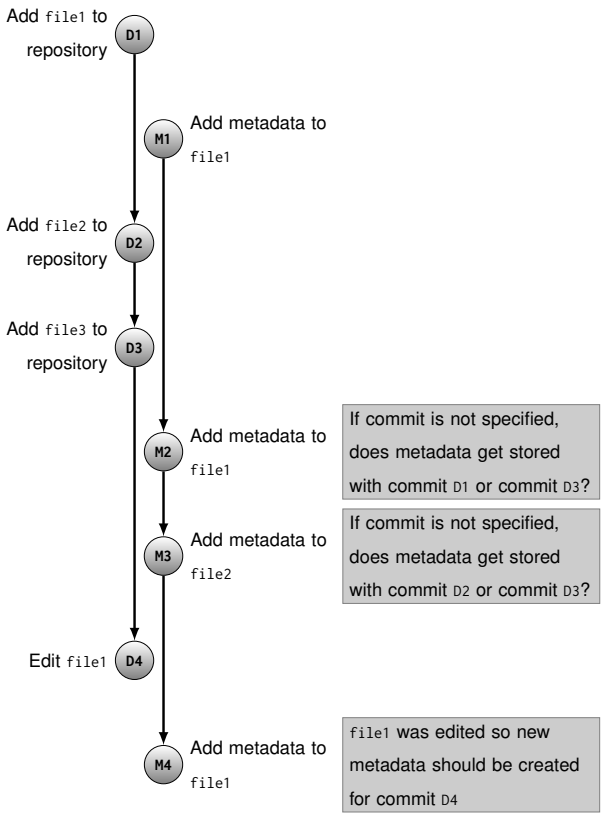

Figure 9: Repository with data (`D`) and metadata (`M`) commits.

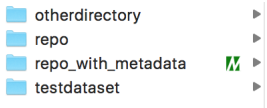

Figure 10: Folder listing with a badge on folders with metadata

Automatically searching back for metadata following the propagation rules above might be the most natural fit for a `get`. For a `set`, this approach might not be what the user intended. For instance, if a file has no metadata defined, the user might want to create metadata from this commit forward (creating a metadata override) and a user just specifying `set` does not describe the user's intention to create the metadata earlier or at the current commit.

A graphical interface (or more powerful command line tool) might be the only way to handle this safely.

6.5. Graphical tools

Some investigation was done to represent Metagit metadata in a graphical interface. This was attempted on a Mac running OS X 10.11.4 as a Finder Sync app extension.

Figures 10 and 11 show how the extension draws an icon or badge on items where metadata is detected.

Figure 11: File listing with a badge indicating files with metadata

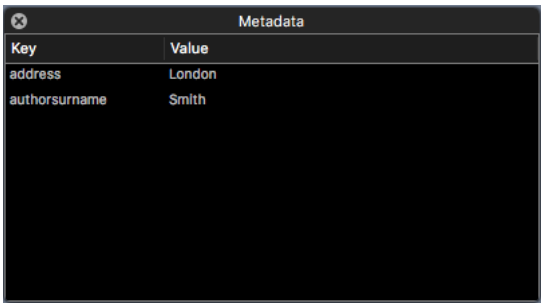

Figure 12: Window showing the key-value pair metadata present on a file. The user requests this window by right-clicking on a file with metadata and selecting the `Show associated metadata` command which has been added to the file's contextual menu by the Finder extension.

A window is displayed by right-clicking on a file or folder which has metadata defined and choosing to display the associated metadata from a pop-up menu; Figure 12 shows the key-value pair metadata created in Section 4.1.

This example demonstrates how metadata would be represented to a user who would not want to use the command line or API. The tool would have to be expanded to allow metadata editing and browsing of metadata versions.

6.6. Merging of branches

We considered merges out of scope for this work. However, one of the intrinsic components of Git is branching and merging, and therefore not carrying metadata forward after a merge is not ideal when working with others in a central repository because merges would happen frequently.

Addition of a post-merge `fix` command to the `m` tool would allow it, in some circumstances, to find and copy the metadata automatically. In other situations the tool might be able to intelligently suggest metadata to be copied. Copying metadata is a manual process so another possibility is to add a new `s++` option to accompany the `s+` and `s-` options. This would force a search for metadata defined before a merge, but careful thought would be required as to which of the two branches would be used, and what to do when other merges are detected further back in the commit history. The following rules are proposed:

- **When the same file exists in both branches with shared metadata:** In this instance, we could trust the shared metadata as valid.

- **When overridden shared metadata exists, distinct on one of the branches:** In this instance there are two distinct metadata definitions for the same file so the user will have to resolve the conflict or indicate to `s++` which branch to follow.

- **When different files with the same path exist in different branches:** This time, if the file after the merge is the same as in one of the branches, the metadata can be trusted from that branch. If a new merged file was created then new metadata would be needed.

- **When the same file exists in both branches with distinct metadata:** This means the user will have to resolve the conflict or indicate to `s++` which branch to follow.

Detection of these situations would involve walking back through the history until a match is found, increasing metadata retrieval time. This is why it is proposed that these rules need to be manually activated, e.g. with `s++`.

These complexities require further exploration to fully understand how to deal with merges.

6.7. Using the tool for data curation

The scope of this paper was to demonstrate the management of metadata and tracking of edits in a decentralised way though a distributed revision control system, and is aimed early in a dataset's life cycle. For this tool to cover the requirements of data curators and link to later stages in the life cycle there are a number of concerns to be discussed.

Institutional or disciplinary repositories are generally used for published and archived data, as discussed in Section 2, so the progress of data from a tool like Metagit into the repository requires consideration. Data curators need to ensure the reliability, integrity and authenticity of digital objects, which this tool can assist with and was covered in Section 6.3.

Standard metadata formats are expected, such as Dublin Core. The ability to store any type of blob as metadata provides flexibility for the system to store the standard metadata formats, provided they can be reproduced in a data file; for example, this could be a MARCXML file, Dublin Core stored as RDF/XML, or JSON representing some other metadata format. Dublin Core includes fifteen elements: contributor, coverage, creator, date, description, format, identifier, language, publisher, relation, rights, source, subject, title and type. Each of these elements could be stored as a separate metadata stream, enabling each element's edits to be tracked. Alternatively, they could be stored as an RDF/XML blob against a single metadata stream in the same way as we demonstrated with JSON in Section 4.2. A specific tool for editing Dublin Core metadata would be required to achieve the best results, so the correct elements are edited and metadata validation can occur. The date, format and contributor elements can be generated automatically from the data commit information (but not creator because the committer might not be the main creator).

Metadata already defined on datasets may prove useful when data is published to a repository, although support would be required from the chosen system, such as DataDryad. Treloar et al. [33] argue that most of the curation task is done by curation professionals once the data is published and crosses the *Curation Boundary*, but points out that 'curation relies on provenance metadata that should have been captured during the research process'. A tool like Metagit helps to collect this metadata for the publication repository to ingest.

6.8. Using the tool for day-to-day metadata management

The biggest scope for improvement in this tool is the ways in which users interact with the metadata. Through the provision of a simple Create, Read, Update and Delete (CRUD) API which follows the metadata propagation rules, as well as extra functions such as `log`, `ls` and `list`, it is possible to achieve complex query and update scenarios. The implementation of query tools was left up to each specific usage of the architecture.

Currently, any tools have to rely on just an exit code to indicate whether metadata exists. Surfacing more information in the API would enhance functionality and remove the reliance on just an exit code. This could include:

- Size of metadata blob
- Type of metadata blob (this could be done by utilising a Python library like `pymagic` to gain Unix `file`-like functionality from Python)
- Commit information, e.g. commit time and date
- Number of metadata streams for an individual blob
- Metadata history for a key
- The length of time it takes to perform an operation, e.g. when searching for metadata and obeying propagation rules

Making this information available in the API would involve increasing the richness of the data structures passed back to the calling function. Passing this information to other Linux tools can be done by printing the information to the screen to be parsed with other Linux tools such as `awk` and `grep`.

Performing some sort of metadata aggregation, such as Sum, Count, Max, Min, Median, Mode and Average, might be desirable with the tool. Examples include:

- Summing all of the `x` streams across all files – this can be done using a Bash script and relying on the exit code from the tool as demonstrated in Section 4.3

- Finding the minimum of all of the streams of one particular file. Again, this can be done using the basic functionality provided by the tool. The output from the `m list` and `m log` commands can be parsed to identify metadata streams to iterate over. With the new functionality outlined above, this could be improved by checking the metadata type before proceeding.

A query language, whilst beyond the scope of this work, would bring all of these features together. It would assist with – and simplify – complex requests. Any operations would need to take into consideration the data type of the metadata blob to ensure the type is as expected.

SQL is the language that is most commonly used for querying relational databases, and many systems adopt a language similar to this for querying data (e.g. SPARQL [14], HiveQL for Hadoop [34], App-fsql [35], FSQL [36]). When a dataset becomes particularly large, or has a large number of metadata streams against a particular file, filtering and querying the dataset would be easier with a query language than using the tool in the way that we have demonstrated. The example in Section 4.3 would be much easier with a query like this:

```
SELECT filename, authorname -- Properties
FROM HEAD                   -- Commit
WHERE filename LIKE "%.csv" -- Filter
ENABLE PROPAGATION          -- Search enabled
```

This is not intented to define any syntax for the query language, but to demonstrate the utility of this functionality.

7. Conclusions

This paper demonstrated the potential of a decentralised key-value store used for recording metadata against a file. Users can edit their own metadata locally and, through the use of Git's tools, can then push it along with the data to a central repository. An architecture was presented and metadata propagation rules were defined for when files were modified. Storage of key-value metadata in Git was demonstrated with a proof of concept tool called Metagit (or `m`) and a number of examples: the key-value capability in use, the use of a JSON blob for more flexible key-value storage, how branching and merging affects metadata, the use of the tool with other commands such as Unix `find`, and the use of a metadata stream to store a thumbnail of an image against a file or folder.

The architecture of such a tool can affect the complexity, capability and performance of such a tool, and a number of other architectures were presented and discussed.

We have shown that a tool that can annotate data with metadata in a decentralised way without affecting the data branches is possible and the examples demonstrate its value and potential. The use of Git means

that a record of metadata modifications is kept and there is potential to use GPG to sign a blob or commit to improve metadata provenance.

As the metadata propagation rules are complex, graphical tools would help with browsing and editing metadata; a read-only tool was demonstrated for Mac OS X.

8. Future Work

In the future, we would like to look at expanding the interface tools, particularly onto other platforms such as Microsoft Windows. Due to the complexities of metadata propagation, the presentation of metadata and the data commits to which they are attached and how they affect later commits will greatly improve the ability to explore metadata. Displaying this in an informative and simple way will also help with the editing of metadata.

Surfacing more information about the dataset in the API and investigating how to integrate query languages with the tool is essential to enrich the tool's functionality.

Investigation of alternative approaches – such as forking Git itself in order to store metadata in the same way as Subversion does – might lead to interesting outcomes. The architecture presented is a very Git-specific approach so adapting the tool to store the metadata as blobs in a normal directory would enable any file system or revision control system to track metadata and allow more flexibility. The trade-off is that metadata and data will then be mixed together which this architecture tried to avoid.

Investigating the signing capabilities of Git is sensible to improve the tool. GPG can be used to sign commits, or signing a blob before it is committed to the repository provides some separation from Git should metadata be exported or the model adapted to other versioning systems.

Rewriting the tool in C (to match the Git code itself) might offer a performance benefit over the current Python version. Any performance improvements that can be made would help when using the tool to process metadata on large numbers of files.

Finally, branch merging rules should be investigated, and other approaches to the tool's syntax should be more fully explored.

9. Acknowledgements

We gratefully thank the University of Southampton for ongoing support, and Nana Abankwa, Mihaela Apetroaie-Cristea, Andrew Poulter and Lasse Wollatz for the helpful discussions whilst developing this architecture.

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

[1] Sandy, H.M., Dykas, F.. High-quality metadata and repository staffing: Perceptions of United States-based OpenDOAR participants. Cataloging & Classification Quarterly 2016;54(2):101–116. doi:10.1080/01639374.2015.1116480.

[2] Hodge, G.M.. Best practices for digital archiving: An information life cycle approach. D-Lib Magazine 2009;6(1). doi:10.1045/january2000-hodge.

[3] Tani, A., Candela, L., Castelli, D.. Dealing with metadata quality: The legacy of digital library efforts. Information Processing & Management 2013;49(6):1194–1205. doi:10.1016/j.ipm.2013.05.003.

[4] Duval, E., Hodgins, W., Sutton, S., Weibel, S.L.. Metadata principles and practicalities. D-Lib Magazine 2002;8(4). doi:10.1045/april2002-weibel.

[5] Microsoft, . How to minimize metadata in Word 2003. Knowledge Base; 2006. URL: http://support.microsoft.com/kb/825576.

[6] Adobe Systems Incorporated, . Document management – Portable Document Format – Part 1: PDF 1.7. 1 ed.; Adobe Systems Incorporated; 2008.

[7] Adobe Systems Incorporated, . XMP specification. 2005.

[8] Dublin Core Metadata Initiative, . Dublin Core Metadata Element Set, version 1.1: Reference description. 2012. URL: http://dublincore.org/documents/dces/.

[9] O'Sullivan, B.. Making sense of revision-control systems. Communications of the ACM 2009;52(9):56–62. doi:10.1145/1562164.1562183.

[10] Scott, M., Johnston, S.J., Cox, S.J.. Metagit GitHub repository. Data Set; 2016. URL: http://eprints.soton.ac.uk/393614. doi:10.5258/SOTON/393614; zipped and archived to institutional repository (will be populated pending publication).

[11] Schandl, B., King, R.. The SemDAV project: Metadata management for unstructured content. In: Proceedings of the 1st International Workshop on Contextualized Attention Metadata: Collecting, Managing and Exploiting of Rich Usage Information. CAMA '06; New York, NY, USA: ACM. ISBN 1-59593-524-X; 2006, p. 27–32. doi:10.1145/1183604.1183612.

[12] Ruiz, E.J., Hristidis, V., Ipeirotis, P.G.. Facilitating document annotation using content and querying value. IEEE Transactions on Knowledge and Data Engineering 2014;26(2):336–349. doi:10.1109/TKDE.2012.224.

[13] Lassila, O., Swick, R.R.. Resource Description Framework (RDF) model and syntax specification. W3C Recommendation; World Wide Web Consortium; 1999. URL: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[14] SPARQL query language for RDF. W3C Recommendation; 2008. URL: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[15] Rys, M.. Scalable SQL. Communications of the ACM 2011;54(6). doi:10.1145/1953122.1953141.

[16] Meijer, E., Bierman, G.. A co-relational model of data for large shared data banks. Communications of the ACM 2011;54(4):49–58. doi:10.1145/1924421.1924436.

[17] Chodorow, K., Dirolf, M.. MongoDB: The definitive guide. O'Reilly Media; 2010. ISBN 978-1-4493-8156-1.

[18] Amazon, . Amazon SimpleDB documentation. 2013. URL: http://aws.amazon.com/documentation/simpledb/; accessed 2016-03-23.

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., et al. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review 2007;41(6):205–220. doi:10.1145/1323293.1294281.

[20] Fultz, J.. Forecast: Cloudy - SQL Azure and Microsoft Azure Table Storage. MSDN Magazine 2010;URL: https://msdn.microsoft.com/en-gb/magazine/gg309178.aspx; accessed 2016-03-23.

[21] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems 2008;26(2):4:1–4:26. doi:10.1145/1365815.1365816.

[22] Punnoose, R., Crainiceanu, A., Rapp, D.. SPARQL in the cloud using Rya. Information Systems 2015;48:181–195. doi:10.1016/j.is.2013.07.001.

[23] Karger, D., Bakshi, K., Huynh, D., Quan, D., Sinha, V.. Haystack: A general-purpose information management tool for end users based on semistructured data. In: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005). 2005,.

[24] Dong, X., Halevy, A.. A platform for personal information management and integration. In: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005). 2005,.

[25] Rico, M., Camacho, D., Óscar Corcho, . A contribution-based framework for the creation of semantically-enabled web applications. Information Sciences 2010;180(10):1850–1864. doi:10.1016/j.ins.2009.07.004; special Issue on Intelligent Distributed Information Systems.

[26] Groza, T., Handschuh, S., Möller, K., Grimnes, G., Sauermann, L., Minack, E., et al. The NEPOMUK project – on the way to the social semantic desktop. In: Proceedings of I-SEMANTICS 2007. 2007,.

[27] Scott, M., Boardman, R.P., Reed, P.A., Cox, S.J.. Managing heterogeneous datasets. Information Systems 2014;44:34–53. doi:10.1016/j.is.2014.03.004.

[28] Dryad, . Dryad digital depository website. 2016. URL: http://datadryad.org/; accessed 2016-04-05.

[29] Kraker, P., Lex, E., Gorraiz, J., Gumpenberger, C., Peters, I.. Research data explored II: the anatomy and reception of figshare. In: Proceedings of the 20th International Conference on Science and Technology Indicators (STI 2015). 2015,.

[30] Brase, J.. DataCite – a global registration agency for research data. In: Proceedings of the 4th International Conference on Cooperation and Promotion of Information Resources in Science and Technology. IEEE; 2009, p. 257–261. doi:10.1109/COINFO.2009.66.

[31] Khazalah, F., Malik, Z., Rezgui, A.. Automated conflict resolution in collaborative data sharing systems using community feedbacks. Information Sciences 2015;298:407–424. doi:10.1016/j.ins.2014.11.029.

[32] Ives, Z.G., Green, T.J., Karvounarakis, G., Taylor, N.E., Tannen, V., Talukdar, P.P., et al. The ORCHESTRA collaborative data sharing system. SIGMOD Rec 2008;37(3):26–32. doi:10.1145/1462571.1462577.

[33] Treloar, A., Groenewegen, D., Harboe-Ree, C.. The data curation continuum: Managing data objects in institutional repositories. D-Lib Magazine 2007;13(9/10). doi:10.1045/september2007-treloar.

[34] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., et al. Hive: A warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2009;2(2):1626–1629. doi:10.14778/1687553.1687609.

[35] Perlancar, . fsql – perform SQL queries against files in CSV/TSV/LTSV/JSON/YAML formats. 2016. URL: https://metacpan.org/release/App-fsql.

[36] Gahag, . FSQL – File System Query Language. 2015. URL: https://github.com/gahag/FSQL.