Evolving the Software Heritage data model

The Octobus team for Software Heritage

The current data model for the Software Heritage Archive is already successfully holding up what is most likely one of the largest monorepo in existence. Combining the shared experience of the people at Octobus working on version control systems for more than a decade and on Software Heritage itself for a few years, we will try to define a proposal for improving the Software Heritage data model.

First, we will look at how the current format fails to record some of the information that various systems carry, then we will comment on a high-level approach on how to adjust the schema to take this variety of information into account. Finally we will get into more technical details of the changes we recommend.

Table of Contents

1 Existing Limitations

The current data schema of the Software Heritage archive is based off of the Git data schema. Doing so had the advantage of providing a high quality schema *for free* and also greatly simplified the ingestion of Git repositories, a significant part of the archive.

However it also means the data and meta data we will be able to store will be mostly limited to what Git is able to store, leading to some data loss. It also means that some data might be preserved, but at the expense of unpractical and costly conversions.

1.1 Examples from the Mercurial Data Schema

While building the Mercurial loader we noticed multiple information that we had to drop while storing a Mercurial repository inside the Software Heritage Archive.

1.1.1 Branch Information

The first one is about Branch information. In Mercurial, branches are not just a pointer: they are an important information present on each revision, and from that information,the current branch head is computed (and unlike Git, a branch can have multiple heads). In addition, a branch head can be “open” or “closed”, so the open branch heads and open + closed branches can be a different set of revisions. All of this branch information cannot be directly represented in the current archive format. It can be approximated by finding a way to translate the Mercurial data to a different model, but this is cumbersome, imperfect and requires the Archive reader to be aware about the formula of transformation.

1.1.2 Copy and Rename Information

Mercurial tracks copy and rename information for files. This means that there is explicit data in the history that keeps this information. This also applied to copied files ; actually, a renamed file is just a copied file that is also deleted.

This rich information is used by many commands in these tools to provide additional information about the state or provenance of data in the repository, which makes going back in history much easier, and allows for a deeper understanding of how code-bases evolve over time and how tools are being used by developers.

There is nowhere to store this information in the current archive schema, so we had to drop it while building the current Mercurial importer.
Dropping this information also means we are giving up on any way to validate that the Mercurial identifier is actually correct, because we no longer have the full information to rebuild the Mercurial data schema.

1.1.3 File History

The identifier of a file revision in Mercurial depends of the history of modification of that file. In practice this means that each file-revision is tracking its parents (in addition of potential copy information). This effectively builds a Merkle graph of the history of the file, the same way that both Mercurial and Git build a Merkle graph of commits. Note that this different way of building the identifier means that different identifiers can point to the same actual binary content (that gets the same identifier in the SWH archive).

This extra information make much simpler to perform some history visualization operation, like… listing the history of a file/directory (obviously) or file annotation.

As this history information is currently thrown away this makes it hard to rebuild and re-validate the archived Mercurial content for the archives. We could imagine replaying all change from the roots of the repository to rebuild that file history, but some other issues will get in the way: more on that later.

If the repository uses the “tree-manifest” variants, this history will also be easily available at the directory level.

1.1.4 Subrepos

Mercurial has a sub-repository concept, it allows to add and track the content of revisions from other Repositories, possibly from non-Mercurial repositories.

Git has similar concepts, and different ways of doing it and we currently record it the Git way. Either by having “revision” directory entries, or by having dedicated config files in a directory.

This seems fairly minor, but this means we can get two revisions with the same *abstract* content that ends up with different SwhID, making them harder to compare. We will talk more about that later.

1.1.5 Phases

Phases is a unique concept in Mercurial that tracks which part of the history is considered “immutable” (public phase) and which one still open to mutation (draft phase).

Right now, we have nothing to store this information so it gets discarded on import. We would need to record the heads of the public phases at the time of each import to preserve this.

1.1.6 Changeset Evolution

Mercurial (and possibly Git in the future, as there are works currently in process to add a similar feature) has the concept of “Changeset Evolution”. In short, it is a set of features to gracefully handle distributed history rewriting operations, enabling users and automation to make sense of the moving target of exchanging drafts.

This means having a means of storing “obsolescence markers” which record each history rewriting operation and link obsolete revisions to their successors.

Storing such data would provide invaluable information about the development process of a code-base.

1.1.7 Side Data

Mercurial gained the ability to store “side-data” alongside a content (changeset/manifest/file-revision). These “side-data” are not part of the content identifier and could be harmlessly discarded. However, they often contain valuable pre-computations used by other algorithms (for example copy tracing), and having them around could help the Archive to present more data to the users.

1.1.8 Censored Content (and Corrupted Content)

Mercurial as a concept of *censored* revisions that allows for removing the data of a specific file revision while preserving the history around it. The trace of that file revision and its identifier remains, but the actual data content is no longer accessible.

This is a significant challenge for importing data in the Software Heritage Archive. Without the content, it is impossible to compute a suitable SwhID for that content, cascading to all the content containing it.

We will have to find a suitable strategy to represent those.

A similar challenge will arise for corrupted repositories. If you archive some old repository, it is possible to encounter missing or corrupted data that we won’t be able to restore. We end up in the same situation, having no way of computing a SWHID.

1.1.9 Authors field

While Git’s author field is quite constrained, Mercurial’s author field is free form. This means we could get multiple authors (usually are comma-separated author fields). Being able to explicitly parse and store multiple authors for a revision would be better.

1.2 Examples from the Bazaar Schema

1.2.1 Empty Directories

Bazaar permits storing empty directories: it may be needed to keep that behavior for certain build systems or even historical data.

1.2.2 Files and Directories Renames

Like Mercurial, Bazaar is able to explicitly track rename. Unlike Mercurial it explicitly tracks directories and is able to store rename information for directories too.

1.2.3 Ghost Revisions

Bazaar has the concept of a “ghost revision”: a revision that is here for provenance information without having actual contents, used for linking to other VCS or external sources.

So the ghost revision will create the same kind of challenge we got with Mercurial sub-repo or with Mercurial censored revision, depending the actual type of ghost revision.

1.2.4 Authors field

Bazaar has explicit support for multiple authors.

1.3 Examples from the Subversion Schema

1.3.1 Rename Tracking

Subversion has explicit copy and rename tracking, so that is another VCS that currently cannot record this information when importing into the archive.

1.3.2 External Content

Again, subversion has a way to include content from external sources, including other repositories or CVS.

And again, it will have its own version to do and express that.

1.3.3 Subversion Properties

Subversion has a powerful mechanism to add *properties* to files and directory, it is used for many different things : https://svnbook.red-bean.com/en/1.7/svn.ref.properties.html

Some of these usages are quite common to other VCS, like tracking executable status, or setting up ignore patterns, and that is already an interesting deviation from how Git's or Mercurial are managing ignore patterns.

Some other usages are more original, like tracking the *mime-type* or *expected end of line types.* These are important information that we want to be able to record.

1.4 Examples from the Darcs/Pijul family

1.4.1 Patches as First Class Citizen

Pijul (like Darcs) handles patches instead of snapshots of the full directory state, independent patches commute and the version identifiers (hashes) are independent from the order in which patches are applied. While a concept of snapshots still exists and could be stored in the current Archive, the main part of the development information live in the patches themselves, and being able to store them would be nice.

In addition there is various metadata about theses patches, some might explicitly depends on others patches.

In Pijul, patches can be group in "channel" in a way that is a bit similar to how Mercurial branching is working.

1.4.2 Storing Conflict Information

Some VCS like Pijul, Darcs or Jujutsu have first-class support for storing conflict information (it is also a proposal in Mercurial). For example, if you rebase a revision and it results in a conflict, the conflict will be recorded in the rebased revision and the rebase operation will succeed. You can then resolve the conflict whenever you want or have time to deal with the conflict. You can even hand-off the conflict resolution to someone else with this tooling.

We currently don't have any way of storing this kind of information the archive.

1.5 Examples from Tarballs, Packages and Disk Images

The files directly accessible through the file system and their usual counterparts tarballs can contain a lot more information than what a VCS usually tracks. Some of it might be considered more on the *build artifact* category, but it is still interesting to think about what we can do to preserve them more here.

1.5.1 Permission, ACLs, Timestamps and User Ownership

On disk files have a much wider range of permissions than what VCS usually track. In addition to the usual user/group/all permission, it might also bear ACLs, timestamps, and important ownership information.

All this might be preserved in a tarball, and there might be some case where it matters. It seems useful to think a new schema that can hold this kind of information if needed.

1.5.2 Resource Forks

In the same fashion as what is done with Subversion's properties, some file systems (eg HFS+) support extra data stored in a single file using *resource-forks*. They may contain important information and some tarball formats can preserve them. We will need a way to store them.

1.5.3 Hardlinks

File file system and tarballs can contain hardlinks: files that are not only identical, but exactly the same. Presenting this explicitly might be useful.

1.6 Other Considerations

1.6.1 External Identifiers

Various VCS tend to have their own identifier for revisions, and various contents. Since we "downgrade" everything to the internal archive schema, we cannot efficiently query the archive about knowledge of the these identifier to detect already uploaded content.

We do have such mapping on the revision object, but having it on more might be valuable.

1.6.2 Incorrect and Corrupted Content

In addition to the plain "corrupted" data we mentioned earlier, there might be data in existing repository that does not really match the expected schema.

Usually those are created by older versions of the tools and might be harmless. However, they will result in a mismatch between what the SWH Archive data schema expects, and the data we need to store to keep and accurate view of the repository. Being able to mark these cases and maybe even record the details of an inconsistency is going to be useful eventually.

We can find multiple examples of this in some Mercurial repositories: some revisions have two identical parents. This should not happen, but it did happen and this will impact the hash of this revision and all revisions above it. In the same way, some people might have introduced inconsistencies in their file history graph: the content is correct, but the history to get to it is not.

2 Recommendation: a two-layered approach

Preserving the history of software is almost as important as preserving any given snapshot of it. For example, stripping away things like the branch data, file history or the copy information from a Mercurial repository reduces our ability to understand and re-use software in the long-term.

Our experience with working on and with the Software Heritage Archive has highlighted the multiple roles that the Archive fulfills and the constraints it has to live with. In particular, two complementary usages emerge:

- The strict preservation of content and development processes of software over the ages,
- The ability to relate and compare similar content used in different forms and locations.

Both of these usages are important but they require competing information. Preserving the history requires to store as much detail as possible while comparing requires translating content into a simplified common base.

In addition, the first part of this document highlights the variety of models that exists and the futility of trying to build a single model that would be able to encompass all of them.

To face this challenge, we are proposing to split the data schema into two main layers: a *core* layer and a *semantic* layer.

The core layer will hold a simplified and unified version of the data that will make comparison simpler. It will also take care of the de-duplication of content that is necessary to make the Software Heritage Archive viable.

The Semantic layer will take care of recording data from each model as accurately as possible, to fulfill the preserving mission of the Software Heritage Archive.

2.1 The Core Layer

The core layer is responsible for storing comparable content in a de-duplicated way.

Since revisions and releases have different implementations and semantics across VCS and packaging systems, we strongly feel that they should be addressed in a separate layer than the storage of pure content, the *semantic* layer.

The *core* layer focuses on storing the very common denominator of what we are trying to archive: snapshots, directories, symbolic links, executable bit, files, all without any other metadata like filename permissions, etc. The exact list of attributes to preserve will have to be determined when actually building the model.

This stripped down version should make it simple for content coming from subtly different sources to be compared without the variations inherent to their original source interfering. Two identical contents coming from Mercurial and Git should comparable without the rename information from Mercurial interfering.

To push things further, if we compare directory content coming from Windows and from a Unix system. Some properties might be lost, notably the executable bit, so we should build a model that allows a simple comparison without these differences interfering.

Another important element that hampers comparison is the use of sub-repositories and external content. From the VCS's point of view, a source tree might contain its own directories and files and some pointers to other resources containing a directory and tree structure. It you compare this to an on-disk directory, the pointer does not exist and the other directories tree simply exist in place. So having a version of the content where the external resources are resolved will be necessary to be able to fully relate the two contents.

All of this will be hard to solve with a single structure, so the *core* logic might benefit from storing multiple "equivalent" versions of the same content using the common variants.

Keeping the objects we store simple will also help the de-duplication effort of the core layer, especially for files. The majority of the storage consumed by the Software Heritage Archive is used by file contents and directory trees, ensuring that they are only stored "once" is important. We should not lose this property that already exists in the current schema.

2.2 The Semantic Layer

The semantic layer will store the metadata and properties specific to each family of source (one for Mercurial, one for Git, one for tarballs, etc.), giving powerful abilities to the archive to better verify, rebuild, compare and sustain software in all of its forms. Introducing the semantic layer will increase the number of objects the archive needs to keep track of, but it will hold information in a more flexible and ultimately more correct fashion.

This layer can be seen as containing most small metadata, possibly in a many small objects, while the bulk of the data will be stored in the core layer. For example, this layer would preserve the rich file-level history available in Mercurial, and that file history would point to elements of the core layer preserving the actual contents of the file.

Aside from adding more metadata and general information, there could be a more subtle interaction between the two layers in the ability to "normalize" content. A release tarball of a Mercurial repository will not have the .hgignore file, but will be identical in most other aspects, thus the semantic layer could produce and link to multiple "normalized" versions of the same directories in the *core* layer. That will help comparison of similar content and build a more complete picture of the software's history.

3 Specific Recommendations

3.1 Strong and efficient hashing

The SWH object storage already allows for multiple hash algorithms to counter collision attacks and enable a seamless transition to stronger cryptographic algorithms when they appear.

However, SWH IDs (the identifiers for something present in the archive) and a lot of the tooling are still using SHA-1 which is now quite weak, slow to compute and not parallelizable. Switching to a more modern cryptographic hashing function like BLAKE3 for all primary hashing would yield massive performance improvements and make the tooling more resistant to attacks in the future. One way of enabling a transition out of BLAKE3 and any other algorithm that may become too weak or otherwise obsolete in a space-efficient way is to use the first bit of the hash to indicate the version of the hash, enabling a transition of the hashing function up to seven layers deep.

In the same way, the compression/decompression algorithms used by the object storage (bz2, lzma, gzip, zlib) leave something to be desired compared to zstd, both in terms of efficiency of compression and speed of compression/decompression. This is less of a model issue and more of a related strong suggestion.

3.2 Using a Binary Format

Regardless of the hash algorithm we choose, another important question is *what* do we feed to the hash function. The conceptual model stored in the Software Heritage Archive must have an official encoding into bytes that we will hash.

The current SwhID is mostly based on the data encoding of the Git DVCS, at least for the Content and the Directory objects. We believe this could be significantly improved.

The data encoding for Content are mostly fine, but using plain hashing of the content, without the `“<length> “` prefix seems simpler and better.

The data encoding for Directory is a wider topic. A directory does not have “canonical” contents as its representation has to be encoded using a custom format. The currently used Git encoding is not ideal for multiple reasons:

- the `“<length> “` prefix makes it hard to start hashing before the full data is generated,
- The permission field is both too wide and too theoretically constrained. In practice, the Software Heritage Model allows quite arbitrary data in there. This is far from ideal: first it makes comparison harder as subtle permission differences might result in content being seen as different in a context where it should not. Second, but not least, it makes the encoded version of this field variable size. This forces you to encode all directory entries to compute the

final length. A cumbersome and inefficient process.

- All directory names end with `/`. For Git developers themselves this is mostly useless and considered an old mistake. This also makes the encoding process more cumbersome as that extra character has to be added on the fly.
- The encoding is mostly text-based, using a couple of specific marks to delimit fields. This might be a bit fragile in some cases: for example, some file systems allow for `\0` in filenames. If we were to archive legacy data from this kind of system, we might run into problems. In addition this is less practical when reading data from that format, as this make much harder to access an arbitrary entry.
  That point is less important for the Software Heritage Archive where the encoding will most likely only be used to compute the ID. However it might still be useful to have an efficient encoding, and direct access to the encoded blob might also be useful in some cases.

Our recommendations for a new directory encoding are:

- Use a strict binary encoding, the size of any variable length field should be explicitly stored and used to retrieve the variable length field.
- Use a limited number of fields with a small set of possible values (especially for permissions). If the binary encoding allows for unknown or inconsistent data, you can expect them to happen, even in the restricted Software Heritage environment. So the less room there is for *invalid data*, the better.
- It is often better to gather all the fixed-size data in an initial “index” at the start of the encoded value. This allows for simple arbitrary access to items without having to parse the full binary blob as the size of an entry for any item is known in advance. Combined with the sorting of entries that must be done to ensure an unique identifier, this allows for very efficient search and access to any data in the blob.
  All the variable length data is written after the index. The index contains the necessary information to find the rest.

3.3 SwhID versioning

Currently a SwhID based on sha1_git is in used, this document is advocating for the use of another hash function and another encoding of the data. This kind of change will likely happen again in the future. Our recommendation is to include a “version” as a prefix of the new hash we use: this means turning a 32 bytes result of the hash function into something one byte longer, adding a prefix with the SwhID version number (or discarding the final byte of the hash to keep it the same size).

An alternative would be to explicitly list the format version in all SWHIDs: `v1:f3cd2d6eeef9`, `v2:6193e846cb65`

3.4 Interesting New Information to Store

In the first section of this document, we have listed many different kinds of information that the current model cannot preserve.

As explained before, our recommendation is not to try to build a single piece of model that would be able to record everything in the same structure, but to have multiple specialized layers for each family of storage. To help future work, here is a summary of all the different new types of information that might be useful to record in the archive eventually:

- support for multiple authors,
- rename/copy information for files and directories,
- tracking history at the file and directory level,
- tracking of custom attributes (resource-fork, Subversion properties, Git attributes),
- full storage of file and directories permissions, ACL, timestamp and ownership (when applicable),
- “Mercurial”-style branch information (or Pijul Channel),
- explicit tracking of subrepositories and other external content (there is currently some support in the archive, but it is very Git-centric)
- storing patch-based VCS (and associated metadata like patches dependencies),
- preserving conflict information,
- keeping track of the “Changeset Evolution” history,
- preserving hardlink information,
- proper handling of missing/censored/ghost/corrupted content.
- An official way to record data and encoding inconsistency in the ingested data.

At least two of these points, the one about sub-repositories and the one about missing content will require some thinking at the level of the *core* layer, not just the *semantic* layer.

4 Final Words

Throughout the years of working with Software Heritage, we have seen the different evolutions that their model has gone through and feel confident in their ability to improve upon the already solid foundations that are in place, yet a larger evolution is needed to ensure the Software Heritage Archive can accurately preserve the development history of the software it archives.

However, the Software Heritage team has been mostly focusing on other challenges in the past couple of years, and a full overhaul of the storage model is not planned yet. Other tasks, like migrating the main storage from PostgreSQL to Casandra are being completed beforehand.

As a result, after discussing it with the Software Heritage team, we focused this document on the census of data and meta-data we are currently losing and on an analysis of the issues of the current model. The result should be a solid help to build the new model when the time to do so comes.