Engram; a fast and space-efficient version control system in Rust

Michael Bednarek
michael@bednarek.cloud
Bucknell University

Edward Talmage
elt006@bucknell.edu
Bucknell University

December 8, 2025

Abstract

Engram is a fast and space-efficient version control system for portable file backups, inspired by git and rsnapshot. [1] It creates snapshots of directories and stores them in a compressed and portable delta-based format. Engram can be used in a cron job to automatically backup files, and the backup can be stored remotely with a tool like rclone. [2] It will only update internal files when necessary, so modification times can be factored when using rsync-like tools. Engram does not encrypt snapshots, so this should be handled externally if desired. Unlike other backup tools, Engram allows for deletion of any number of previous revisions, as it stores instructions on how to create previous snapshots from the current state. Engram is also heavily optimized for performance, and is capable of processing files at multiple GB/s on modern hardware.

1 Introduction

Engram's core function is to store snapshots of a specified directory's state, and provide the necessary functionality to return to one of these snapshots. Each patch file contains all the necessary instructions and data required to return to the adjacent previous snapshot.

Engram stores the most recent state of the user's directory, along with patch files and necessary metadata in a directory called a *repository*. See Listing 1 for the structure of this directory. The current state of the directory is stored under the base directory in the repository. The data in this directory is compared to the user's actual directory every time the update command is run to generate a new patch file. A table of file content hashes is also stored, containing both fuzzy and strong hashes used to efficiently track changes of files.

```
repo/
├── .lock          <-- [in-progress lockfile]
├── base/
│   ├── file1.txt  <-- [current files]
│   └── file2.txt
├── hashes         <-- [hash table]
└── patches/
    ├── 1736634596 <-- [patch files]
    └── 1736634679
```

Listing 1: Engram repository directory file structure

2 How Engram Works

2.1 Patchfiles

Engram stores instructions, known as *entries*, on how to return to a previous snapshot of a directory in patch files. It is important to note that the entries generated may not precisely match the actual actions of the user. Each of these files contains a list of entries, along with the necessary data to reconstruct specific files for certain types of entries. Each entry corresponds to a specific operation to apply to either a file or a directory, and also references the source path of that file or directory, and as appropriate, a destination path additionally as shown in Table 1. Some entries also require additional data to be stored, known as *objects*. An object represents either the contents of a file in the case of *Add* or *Update* entries, or a patch to apply to a file in the case of *Delta*, *DeltaCopy*, and *DeltaMove* entries.

Command	**Function**	**Desintation Field?**	**Associated Object?**
DirDelete	Deletes all files in a directory	No	No
DirMove	Moves all files in a directory to a common destination	Yes	No
DirCopy	Copies all files in a directory to a common destination	Yes	No
Delete	Deletes a single file	No	No
Move	Moves a single file to a destination	Yes	No
Copy	Copies a single file to a destination	Yes	No
DeltaMove	Moves and modifies a single file to a destination	Yes	Yes
DeltaCopy	Copies and modifies a single file to a destination	Yes	Yes
Delta	Modifies a single file	No	Yes
Update	Replaces the contents of a single file	No	Yes
Add	Adds a new file	No	Yes

Table 1: List of all entry operations types as stored in patch files

2.2 Entry Generation

Entries are initially generated through a multi-step process beginning with comparison of underlying file paths. File paths from both the new and old state are classified into one of three types; *only_in_old*, *only_in_new*, and *same*. Each of these paths are then grouped with a corresponding pair of fuzzy and strong hashes.

First, new and old hash pairs with the same file paths are compared. These comparisons result in one of two resulting entries, either a *Delta* or an *Update*, depending on the hash pair distance being above or below the global distance threshold respectively.

Then, hash pairs corresponding to files exclusive to the old state are compared with hash pairs corresponding to all files in the current state. These hash distances are computed and assigned a ranking depending on both the distance and the origin of the file path as shown in Table 2.

	From same	From only_in_new
DISTANCE = 0	0	1
DISTANCE < THRESH	2	3
DISTANCE >= THRESH	4	5

Table 2: Hash distance pair rank values for use in entry generation

These pairs of file paths are sorted by their rank in ascending order using a counting sort. A greedy process is then performed to convert these pairs into entries. Pairs are processed in order, and the first pair seen is converted into an entry based on its hash distance and origin. See Table 3 for the corresponding entries generated. Pairs that result in a move operation block any further pairs that reference the same file from the current state.

	From same	From only_in_new
DISTANCE = 0	Copy	Move*
DISTANCE < THRESH	DeltaCopy*	DeltaMove*
DISTANCE >= THRESH	Add	Add

Table 3: Pair entry assignments, * indicates a blocking operation

Any files from *only_in_new* are then assigned a *Delete* entry if not previously processed. If no new or common files exist, all old files are assigned an *Add* entry, Likewise, if no files exclusive to old exist, all new files are assigned a *Delete* entry. Pseudocode is provided in Listing 2 for this initial entry generation process.

```
1. Hash files in new state (using fuzzy hashing)
2. Categorize filenames: only_in_new, only_in_old, same (in both)
3. Compare files in same (with names in both new and old):
   - Compute hash distance between old and new versions
   - Generate Delta (small change) or Update (large change) entries
4. Match old files to new/same files and generate entries:
   - For each old file, find closest hash match in new/same files
   - If match is from same: generate Copy or DeltaCopy
   - If match is from only_in_new: generate Move or DeltaMove
   - If no good match exists: generate Add + Delete
5. Generate Delete entries for unmatched new files
6. Return all entries and new hashtable
```

Listing 2: Pseudocode of initial entry generation process

2.3 Directory Compression

To reduce the resulting size of patch files, some entries may be compressed into directory operations under certain conditions. This stage will detect entries that share a common parent, and group them together into one directory-based entry. Three types of directory entries exist; *DirDelete*, *DirCopy*, and *DirMove*.

The circumstances required for each type of entry varies. *DirDelete* entries require that all files in the respective directory have been assigned a Delete entry, whereas *DirCopy* and *DirMove* require that only 50% or more of files have been copied or moved to a common destination.

The process of grouping initial entries into directory entries for *DirCopy* and *DirMove* entries begins by extracting the sources and destinations of all *Copy* and *Move* operations respectively

into pairs. For each pair, source and destination path suffixes are removed and the remaining paths are stored, up until the longest common suffix. These resulting pairs of path prefixes are counted and then sorted in order of highest frequency followed by lowest depth. For *Move* operations, later source paths that are a subset, superset, or identical to other source paths in this list of pairs are removed. These pairs are then filtered to only include pairs in which the corresponding frequency count is greater than 50% of the total current file count (recursive) in that source directory. This process is shown in Listing 3. All remaining pairs are then assigned a *DirCopy* or *DirMove* operation, and the original list of entries is filtered to exclude any *Copy* or *Move* operations that were used. In the event that a *DirCopy* or *DirMove* includes files that were not initially referenced by a *Copy* or *Move* entry, respective *Delete* and *Move* entries are added to compensate.

```
1. Extract all (src, dst) pairs from move/copy operations
2. For each pair, find longest common path suffix and remove it
3. Count frequency of each reduced (src, dst) pair
4. Sort pairs by: frequency (high -> low), then depth (shallow -> deep)
5. Deduplicate by source prefix (skip for copies; allow multiple destinations)
6. Keep only pairs where:
   - Frequency > 50% of files in source directory
   - Frequency > 1
7. Convert qualifying pairs to DirMove/DirCopy entries
```

Listing 3: Pseudocode of directory entry path generation process

2.4 Hashing Distance

Engram typically stores two types of hashes for all stored files, a strong hash and a fuzzy hash. Strong hashes are recalculated for every snapshot, but fuzzy hashes are only recalculated if the strong hashes do not match. However, the fuzzy hashes require a minimum filesize, so engram will omit them if this requirement is not met. These hashes are used to detect differences between new and old files, and to roughly estimate how different they are. If the strong hashes match, the distance between hash pairs is considered zero. Otherwise, the distance between fuzzy hashes is used, with a minimum distance set to one. If either one of the fuzzy hashes in a pair is missing, the distance is set to the maximum value.

2.5 Entry Application

Engram applies patch files in a way that allows for both the original data directory and the repository to not be modified until all entries are processed. The process of applying entries to revert to a previous snapshot begins by reading all entries from corresponding patch files up until that snapshot. Each entry is then associated with a reference back to the patch file from which it was read. The entries are then processed in reverse chronological order into a hashmap of entries associating each filename with a list of these entries.

Each *Delta*, *Add*, *Delete*, or *Update* entry is appended to the corresponding chain for its source path, while other entries modify the chains themselves. *Copy* and *DeltaCopy* entries copy and append any existing chain of their source path to the chain of their destination path. *Move* and *DeltaMove* operations append any existing chain of their source path to the chain of their destination path, and then delete the key referencing their source path. The pseudocode for this process can be seen below in Listing 4.

```
1. Read all entries from patch files and create structs of entries and associated
data
2. Add these structs for all currently existing files
3. Iterate through these structs from newest to oldest, building hashmap of filename
-> chain of operations:
   - For directory operations: expand to individual file operations
   - For moves: append chain from source to destination, delete source key
   - For copies: copy chain from source, appending to chain for destination
   - For others: append to source filename chain
4. Filter chains to specific file/directory if requested
5. Clean each chain: keep only relevant operations, deduplicate
6. Process chains in parallel:
   - Apply (deltas, updates, adds) sequentially
   - Read "objects" (diff data for deltas, file contents for updates/adds) from disk
   - Apply deltas to file data when applicable
   - Write final result to disk, computing fuzzy / strict hashes if rebasing
7. If rebasing: remove deleted files, clean old patches, and update table of hashes
```

Listing 4: Pseudocode of general entry application process

DirDelete, *DirCopy*, and *DirMove* operations are expanded out into individual file entries which are then also appended to their respective chains. Any keys in the hashmap that represent a subdirectory of the directory entry source at the time of processing is considered relevant. The complete process for this can be seen in Listing 5. It is important to note that within each patch file, directory entries are processed before file entries to allow for later compensating *Move* and *Delete* entries.

```
1. For each directory operation (DirMove/DirCopy/DirDelete):
2. Find all files with matching source prefix
3. For each matching file:
   - Compute destination path
   - Convert directory operation to file operation
   - Add file operation to that file's chain
4. For moves: transfer existing chains; for copies: duplicate chains
```

Listing 5: Pseudocode of directory expansion for entry application

After the hashmap of chains is fully constructed from all relevant entries, chains are then individually deduplicated in a two-step process. First, entries from chains are read in order, and any entries prior to the last *Add* or *Update* entry are deleted. Then, the resulting chains of entries are processed in reverse order to exclude any entries that do not directly relevant paths. Starting with the key of the chain as the final destination of the file, entries are filtered until the next entry that results in a file at that destination. This is repeated until all entries are processed. This process results in each chain referencing all modifications required to reconstruct exactly one file.

All chains are then applied in parallel; reading necessary objects from the corresponding patch files and applying diffs or full file contents, before eventually being written to disk. Engram also supports rebasing the repository to a specific snapshot, in which case file hashes are recomputed at this time and updated in the table of hashes, and old files and patches are deleted from disk.

3 Performance & Benchmarks

Engram's performance was evaluated for generating entries over a range of file counts and file sizes. A Python script was used to generate random directories and modifications for

use as a source data directory. Paramaters for both tests were idential with the same initial seed of "engram", 10% maximum file modification rate (v. file count), 50% maximum file content modification rate (v. file size), and time across 100 runs averaged per measurement. Modifications to the directory included file and directory moves, file and directory copies, new files, file content modifications, and file and directory deletions. Resulting times ranged from ~7-80ms, and can be seen in Figure 1 and Figure 2.

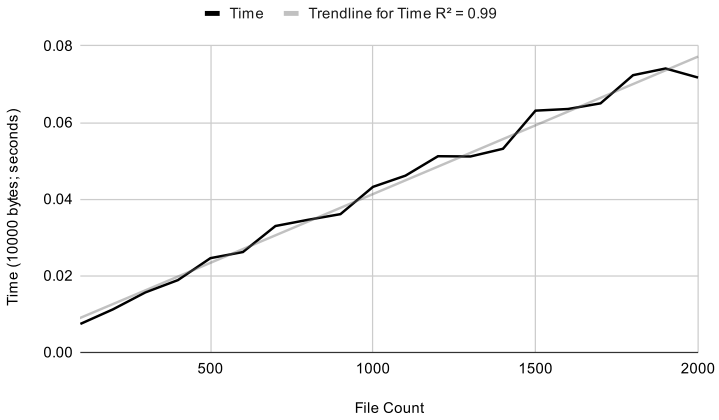

Figure 1: Engram entry generation time v. file count

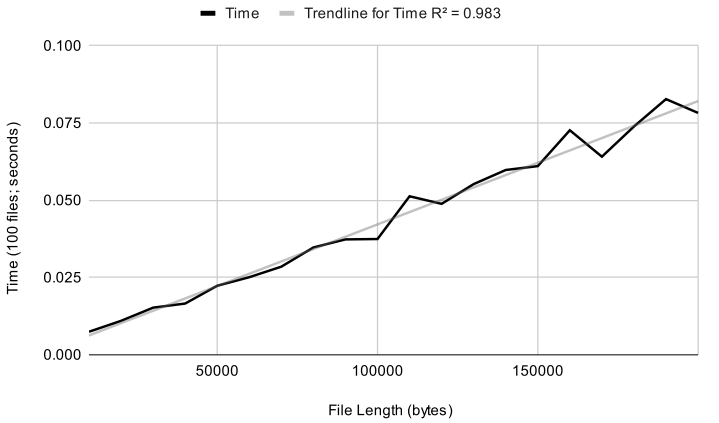

Figure 2: Engram entry generation time v. file length

Approximately linear behavior was observed in both tests, despite the expected $O(n^2)$ cost of the greedy process used in entry generation as compared to file count. This is likely due to the realistically dominminent linear cost of file reads required for hashing.

APPENDIX A

A.1 Backwards Patches

Engram's choice of generating patch files in reverse chronological order has a few key benefits. First, it allows for easy pruning of irrelevant patches. Any of the oldest patch files may simply be deleted without affecting reconstruction of any later snapshot.

A.2 Patchfile Encoding

Entries are compressed and written at the beginning of the patchfile. Each object is then individually compressed and written sequentially. The starting position of each object is encoded at the end of the file using variable width integers preceded by a null byte as seen in Listing 6.

```
<entries><object0><object1>..0x00<object0_pos><object1_pos><object2_pos>..
```

Listing 6: Binary patch file structure for storing entries and relevant objects

This encoding scheme allows for random and parallel access of objects after the final position table is read. The position table may also only be written once the length of all objects are known, and this approach minimizes backwards seeking.

A.3 Hashing

Engram uses GXHash and TLSH for strong and fuzzy hashing respectively. Hashes are stored in a file in the repository known as the hashtable. The hashtable contains a strong hash for all files, and a fuzzy hash for files longer than the minimum TLSH filesize. Internally, hashes for each file are stored sequentially in alphabetical order by corresponding file name. At the end of the hashtable, a run length encoded bitfield is written preceded by a null byte. Each bit in the bitfield corresponds to if a TLSH hash in each hash pair is included or not. The hashtable structure can be seen in Listing 7.

```
<GXHash0><TLSH0?><GXHash1><TLSH1?>..0x00<bitfield>
```

Listing 7: Binary hashtable file structure for storing file content hashes

The choice of GXHash and TLSH for hashes primarily depended on both speed and similarity separation performance. GXHash is one of the fastest available hashing algorithms written in pure Rust, largely due to its use of SIMD and AES intrinsics. [3] Likewise, TLSH has a native Rust implementation, and offers excellent similarity detection while using a fixed hash size. [4], [5]

A.4 Compression / Diff Algorithms

Zstandard is used for all necessary compression, due to its excellent compression ratios while maintaining fast streaming performance. [6] Engram uses a Zstd compression level of three for all applications. DDelta is used for delta patch generation and application. It offers relatively high performance, and a streaming interface unlike other diffing algorithms. [7]

A.5 Rust / Rayon

Rust was chosen as a language due to its combination of memory safety and excellent support for third party libraries, including rayon. Rayon allows for high-level implementation of parallel iterators, which Engram uses heavily. [8]

A.6 Memory Mapped I/O

Engram uses memory mapped I/O to more efficiently process files for hashing and diff application. There appears to be noticible improvements in throughput and read latency when using memory mapping as compared to Rust's standard file interface. The impact of introducing unsafe code using this approach should be further analyzed and may be an important consideration.

A.7 Software Architecture

Engram is implemented using a layer-based architecture. Each layer is only allowed to call functions in the dedicated layer interface of the layer directly below it. The architecture can be seen in Figure 3.

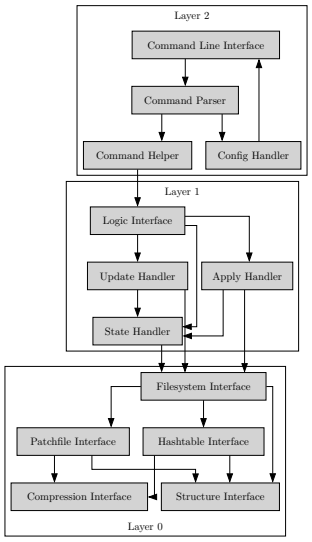

Figure 3: High level architecture diagram for Engram

This approach allows for restrictive control over which peices of code may access the file system, and more precise seperation of logical functions from low and high-level interaces.

Bibliography

[1] "rsnapshot." Accessed: Dec. 07, 2025. [Online]. Available: https://rsnapshot.org/

[2] N. Craig-Wood, "Rclone." Accessed: Dec. 07, 2025. [Online]. Available: https://rclone.org/

[3] O. Giniaux, "GxHash: A High-Throughput, Non-Cryptographic Hashing Algorithm Leveraging Modern CPU Capabilities." Accessed: Dec. 07, 2025. [Online]. Available: https://github.com/ogxd/gxhash

[4] J. Oliver, C. Cheng, and Y. Chen, "TLSH – A Locality Sensitive Hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, Sydney NSW, Australia: IEEE, Nov. 2013, pp. 7–13. doi: 10.1109/CTC.2013.9.

[5] U. Prasad, "Similarity Digests Compared for Malware Detection." Accessed: Dec. 07, 2025. [Online]. Available: https://dzone.com/articles/similarity-digests-tlsh-ssdeep-sdhash-benchmark

[6] “zstd.” Accessed: Dec. 07, 2025. [Online]. Available: https://github.com/facebook/zstd

[7] “ddelta: streaming, uncompressed, less memory-needy version of bsdiff.” Accessed: Dec. 07, 2025. [Online]. Available: https://github.com/julian-klode/ddelta

[8] “rayon.” Accessed: Dec. 07, 2025. [Online]. Available: https://github.com/rayon-rs/rayon