12.2. Data Structures

As we got a basic idea about Design acyclic graph (DAG), let us dive deeper into its storage in Mercurial. DAG is key to the central functioning of Mercurial. We use various kinds of DAGs in storage repos on the disk and the structure in-memory. Let us learn more about what they are and how it works.

12.2.1. Challenges

The first idea of Mercurial was tracked in a mail that Linux Kernel mailing list received by Matt Mackall in 2005. This took place directly after BitKeeper was no more used for the expansion of Kernel. The email had objectives like: simple scalable and efficient.

Matt declared that VCS should deal with trees that has millions of files and can handle many changesets and can get popular among users who are creating revisions over these many years. He also mentioned that technology has limiting factors like:

- Speed: that means the CPU
- Capacity: like how much can the disk and memory hold
- Bandwidth: LAN, WAN, memory and disk
- The seek rate of the disk.

The limiting factors are the disk seek rate and bandwidth WAN and hence must be improvised.

- Storage compression: To save history of files on the disk, what type of compression is most appropriate? While stopping the CPU time from turning into a gridlock, what kind of algorithm is most suitable to yield I/O performance?
- Retrieving arbitrary file revisions: Many VCS will save particular revision in a manner such that a bulk of older versions of files must be read in order to rebuild a newer revision. We wish to command this so that we are sure of recovering older versions and still maintain its speed.
- Adding file revisions: we constantly add new versions but at the same time we do not wish to revise old versions each time we add new one because it would be very slow in a case of multiple revisions.
- Showing file history: we want to view all the changes made in any file recorded on history.we can even make annotations : revising the initial changeset in every line that is presently in the file.

The material has alike scenarios on the project. At this level, the basic functions are revising and making a commit for a new version and discovering any differences in the working file. On the other hand, the latter can be little less speedy for huge trees like NetBeans, Mozilla etc projects that uses Mercurial for VCS

12.2.2. Fast Revision Storage: Revlogs

Matt came up with an idea for solution named revlog that is a short form for revision log. It is a method of effectively storing contents of file revisions (with a little modification as compared to former version). It is expected to be worthwhile in both storage space and time access for using disk seeks, in guidance of usual plot in the last section.

6 bytes	hunk offset
2 bytes	flags
4 bytes	hunk length
4 bytes	uncompressed length
4 bytes	base revision
4 bytes	link revision
4 bytes	parent 1 revision
4 bytes	parent 2 revision
32 bytes	hash

Table 12.1: Mercurial Record Format

There are fixed-length data in the index which of the components are in the table 12.1. It is goof to have fixed-length records as it allows direct time access to the local revision. One can just check the position (index-length x revision) in the file to find the data on index. If we discrete the index from data it will signify that we can read fast the data without going through the seek the disk in all the data file.

The hunk (offset and length) signifies that a part of the file data is read to achieve a compact data for that very revised version. To achieve original data file, reading the base would be a good idea to begin with and then put in deltas along this revision. The plooy is to the conclusion about where to store the new revision base. This conclusion depends on the total delta size as compared to the unzipped length of the version. Note that the file data is compressed by zlib so

that it occupies even very less room on the disk. By restricting the delta chain length doing this, we can confirm that rebuilding the data in a particular revision doesn’t need reading and application of many deltas.

The link revision have revlogs that relates to the most high level revlog and using the local revision number, the parent revised versions are stored.It will make work easier to check data in the applicable revlog. Hash is utilized to store the unique identifier. Instead of 20 bytes for SHA1, we use 32 bytes to keep room for evolution.